University of Alberta

Library Release Form

Name of Author: Andreas Junghanns

Title of Thesis: Pushing the Limits: New Developments in Single-Agent Search

Degree: Doctor of Philosophy

Year this Degree Granted: 1999

. . . .

Andreas Junghanns
Feldstraße 12
09755 Niederwiesa
Germany

Date: . . . .

University of Alberta

Pushing the Limits: New Developments in Single-Agent Search

by

Andreas Junghanns

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 1999

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Pushing the Limits: New Developments in Single-Agent Search** submitted by Andreas Junghanns in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

. . . .

Dr. Jonathan Schaeffer

. . . .

Dr. Richard Korf

. . . .

Dr. Gordon Rostoker

. . . .

Dr. Joseph Culberson

. . . .

Dr. Peter van Beek

Date: . . . .

Meinen Eltern

Abstract

Search is one of the fundamental methods in artificial intelligence (AI). It is at the core of many successes of AI that range from beating world champions in non-trivial games to building master schedules for large corporations. However, the applications of today and tomorrow require more than exhaustive, brute-force search, because these application domains have become increasingly complex. Traditional methods fail to break the complexity barrier caused by the combinatorial explosion that characterizes these large, real-world domains.

This thesis enhances our understanding of single-agent search methods. A puzzle (Sokoban) is used to explore new search techniques for single-agent search. Sokoban offers new challenges to AI research, because it has a much larger search space than previously studied puzzle domains and exhibits a new, real-world-like search-space property. Deadlock, the possibility to maneuver into an unsolvable position, provides traditional search methods with considerable difficulties. This thesis shows the failure of these traditional search methods to solve more than trivial Sokoban problems. The state-of-the-art is significantly improved when traditional methods are changed such that they are able to adapt to each instance. Furthermore, several new techniques are suggested to combat the complexities and challenges exemplified by Sokoban. Most successful is a technique that dynamically gathers knowledge during the search to avoid deadlocks and to improve the search's understanding of the search space. Another technique is described and analyzed that uses the heuristic notion of relevance to focus the search effort. This thesis closes with a suggestion of a framework and a classification for single-agent search enhancements.

Preface

It is quite interesting to ask people what they think of their PhD thesis after it is finished. The reactions range from dismissive hand-waving, to excuses for a number of things. It is rare to meet somebody who is openly proud of their PhD thesis. That is good. It means I am not alone...

I have tried to understand how so much enthusiasm, drive and optimism could turn into impatience and a hope-it-will-be-over-soon attitude. By trying to understand what caused my frustration, I did regain some of the lost excitement for the research. Of course, it always takes too long to finish a thesis. Naturally, the discovery phase is much more fun than documenting what has been found in full detail. We are hungry for the knowledge, but not for the clean-up! The writing phase does not give the same impression of progress – indeed, it seems it will never end. After doing a lot of research and uncovering many things, I feel that I know less now than ever before. How is this possible? We start out seeking the Truth, but inevitably find only deeper questions.

"The more I know, the more I realize how little I know." – Socrates

In the quest to enlarge our circle of knowledge, we inevitably enlarge the frontier, where the questions lie. In the end, the search for answers is a search for new questions. A working title for the thesis was "The Search is the Goal", a play on the Zen Buddhist adage "the path is the goal". This might explain why the feeling of completeness that I was hoping to achieve is missing. To put it in more definite terms, thesis writing is about drawing the line. When enough new questions have been created, it is time to stop. Thus, the thesis in front of you is a work in progress, halted for a moment in time to allow for proper documentation of the results achieved so far. The circle of knowledge does not stop expanding.

I was privileged to have had the opportunity to come to the University of Alberta, its Computing Science Department, and not least, the GAMES research group. Students from around the world come to Edmonton to study some of the hardest problems that games have to offer. The diverse interests and expertise of all the members form a wonderful synergy that leads to high-performance programs and exciting research. I can recall countless discussions in research meetings and at parties where games and puzzles (and how to solve them) were the subject of intense debate.

I have many people to thank, without whom this thesis would not be what it is today. First and foremost, Jonathan Schaeffer and his relentless pursuit of excellence – nothing is good enough. Drawing from his wealth of knowledge and experience has allowed me to solve many hard problems. I thank the members of my examining committee, Peter van Beek, Joe Culberson, Richard Korf, and Gordon Rostoker, for their time and valuable suggestions on how to improve the thesis. Yngvi Björnsson, for being a sounding board for raw or unpolished ideas, and for the many valuable hints and ideas he shared. Darse Billings, for stimulating discussions about games and how they pertain to other aspects of life, as well as his efforts in improving my speaken and wrotten English. Neil Burch, who rekindled my spirits after a long drought of ideas. Tony Marsland, for being instrumental in my coming to Alberta. Other members of the GAMES group that helped produce ideas for this thesis, in so many direct and indirect ways: Mark Brockington, Aske Plaat, Roel van der Goot, Duane Szafron, Denis Papp, Lourdes Peña, and Jack van Rijswijck.

And, last but not least, my family. Manuela, for her patience and support, which cannot be repaid! Anne and Robert, for the joy and inspiration they gave during the final stages of this thesis. To all of you – Thanks!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "To find the way out of a labyrinth," William recited, "there is only one means. At every new junction, never seen before, the path we have taken will be marked with three signs. If, because of previous signs on some of the paths of the junction, you see that the junction has already been visited, you will make only one mark on the path you have taken. If all the apertures have already been marked, then you must retrace your steps. But if one or two apertures of the junction are still without signs, you will choose any one, making two signs on it. Proceeding through an aperture that bears only one sign, you will make two more, so that now the aperture bears three. All the parts of the labyrinth must have been visited if, arriving at a junction, you never take a passage with three signs, unless none of the other passages is now without signs."
>
> "How do you know that? Are you an expert on labyrinths?"
>
> "No, I am citing an ancient text I once read."
>
> "And by observing this rule you get out?"
>
> "Almost never, as far as I know..."

Adso and William in the labyrinth, "The Name of the Rose", Umberto Eco.

1.1 Artificial Intelligence and Search

Research into search methods is a fundamental branch of Artificial Intelligence (AI). Without joining the debate over what intelligence is and how it can be achieved, it seems generally recognized that search-based programs can solve problems that humans would say require intelligence. Games and puzzles are examples of these problems. They have provided artificial intelligence researchers with excellent experimental domains. First, games are closed and well-defined applications where improvements are easily measured. Second, they have supplied researchers with strong motivation and clear goals, such as beating the best humans with an artificial entity.

Fifty years of AI research using games as an experimental test bed has led to some important results:

- Some games are solved. That means the computer knows a strategy that allows it to always achieve the best possible result. Among these games are Go-Moku and Qubic [All94], Nine-Men's-Morris [Gas94] and Connect-4 [All88].

- In several games, programs have surpassed the best humans. In checkers, the program *Chinook* won the World Championship in a regular match and defended its title several times until it retired [SLLB96, Sch97]. The Othello program *Logistello* defeated the world champion 6-0 in a match [Bur97]. The program *Maven* plays Scrabble at such a high level that it loses only a few possible points per game, consistently surpassing human performance.

- In other games, programs are approaching championship caliber that rivals the best humans. In chess, strong programs can beat all but the very best humans. *Deep Blue* even defeated the World Champion Garry Kasparov in an exhibition match [New96]. Gerry Tesaro's *TD-Gammon* plays backgammon on par with the best humans in the world [Tes95].

- For games like bridge [Gin99] and poker [BPSS99], significant progress is being made that may lead to high quality play rivaling the best human players.

These are important success stories for AI research. For some of these games, one could argue that the Turing test has been passed, albeit in a limited domain. However, some of the programs play so well that they would have to start blundering once in a while to appear to be human!

Of course, there are many challenges left. Games such as Go and Shogi still resist the traditional approaches that are successful in most of the mentioned games. We believe it is no coincidence that success in writing programs for games appears to be correlated with our understanding of how to make search work for them. This observation underscores that search is one of the most basic and important tools in AI.

Domains besides two-player games in which search is successfully deployed are optimization tasks, scheduling, and, to a lesser extent, planning. These applications are examples of single-agent search domains. In contrast to adversarial games, such as chess and poker, where opponents try to achieve opposing goals, single-agent search assumes that only one agent is manipulating the world in order to achieve some (optimal) goal. Puzzles belong to the single-agent category as well.

The research into games and puzzles has produced an enormous body of useful techniques and methods for problem solving that has found its way into main stream computing science. However, it has also some serious drawbacks. The artificial nature of games is reflected in their search-space properties. Relatively manageable search-spaces, either small or well structured, have been implicitly assisting the early progress of AI research. However, they do not compare to the complexities of the real-world applications scientists are working on today. Because of the combinatorial nature of most domains and the resulting exponential size of the search spaces, scalability of search methods is of great importance. If the domains used as research vehicles do not keep pace in complexity and relevant properties, the research results are less likely to be useful for practical domains.

The success of search depends on the ability of the program to visit most of the relevant parts of the search space. If the search space is too large and/or heuristic knowledge to focus the search is missing, success is unlikely. Since the search-space size of a problem is fixed, knowledge is needed to focus the search. This is where machines currently fail and humans still have a considerable edge: finding and using knowledge to reduce the problem complexity. Thus, more research on how to focus search on relevant parts of the search space is needed.

Methods that do not adapt to the problem instance, but instead rely on general properties of the domain, can help to improve search efficiency. But, they are limited by the necessity of keeping their knowledge generally applicable. However, humans are capable of learning *during* the problem solving process about how to solve the current problem instance. This suggests developing dynamic methods to glean and use knowledge that pertains to the specific problem instance currently under examination. This specificity can help to break the complexity barrier on a problem-by-problem basis. Specific problem knowledge can remove irrelevant parts from the search with the precision of a scalpel. Of course, instance-dependent knowledge has its price. It has to be found over and over again, and generalizing it is not only little understood, but it would turn this knowledge into a dull, even though larger, machete.

Dynamic knowledge discovery is in fact a form of learning. It is performed at the level of problem instances, but shows all the properties of learning. As the learning progresses, the classification of subtrees as relevant/irrelevant becomes more precise. The result is a more efficient search, reducing the complexity of the search space at hand. However, our current understanding of these dynamic methods is limited at best.

In this thesis, a puzzle game (Sokoban) is used to explore new search techniques for single-agent search. Sokoban offers new challenges, because it has a much larger search space than previously studied puzzle domains and exhibits real-world-like search-space properties. Deadlock, the possibility to maneuver into an unsolvable position, provides considerable difficulties to traditional search methods. This thesis shows that traditional search methods fail to solve more than trivial Sokoban problems. The state-of-the-art is significantly improved when traditional methods are changed such that they are able to adapt to each instance. Furthermore, several new techniques are suggested to combat the new complexities and challenges exemplified by Sokoban. Most successful is a technique that dynamically gathers knowledge during the search to avoid deadlocks and to improve the search's understanding of the search space. Another method that uses the heuristic notion of relevance to focus the search effort is described and analyzed. This thesis closes with a suggestion of a framework and a classification for single-agent search enhancements.

The research presented here leaves a great number of important issues open. The performance of domain-independent solvers is still quite limited. The question is, how can the enhancements suggested here (and, of course, many others already suggested elsewhere) be automatically instantiated for a new domain? Can they be formulated in a domain-independent way? Can we identify the essential properties a domain must have to be amenable to a certain search enhancement? For simple enhancements, such as transposition tables, this is possible. Can we find such ways for other, more complex search enhancements? After all, humans seem to be able to adapt their cognitive processes to a seemingly endless number of new problems. We are only at the beginning...

1.2 Contributions

This thesis enhances our understanding of single-agent search with the following contributions:

- The puzzle game Sokoban is investigated and it is shown that its large search-space size and particular search-space properties offer significant new challenges for AI research. One of these challenges is the possibility of deadlocks: the search can create problem configurations that have no solution. In fact, state-of-the-art single-agent search is shown to be insufficient to even solve Sokoban problems of modest complexity.

- The concept of macro moves [Kor85b] is improved by adding automatic off-line macro-move generation. Significant efficiency gains are the result.

- A new search enhancement is introduced: pattern searches. Small, speculative on-line searches gather dynamic knowledge that helps avoid deadlocks and improve the heuristic estimate of the distance to the solution. The use of this dynamic knowledge allows orders-of-magnitude reductions in search-tree sizes for our Sokoban solver. The necessary properties of the application domain and heuristic function are identified that allow the application of pattern searches. The feasibility of pattern searches for different domains is shown using the example of the 15-puzzle.

- Relevance cuts, a new domain-independent forward-pruning technique is presented. It is theoretically analyzed, and the risks and benefits are studied. The analysis is contrasted with the experimental results. Relevance cuts lead to relatively small search-efficiency improvements in the domain of Sokoban.

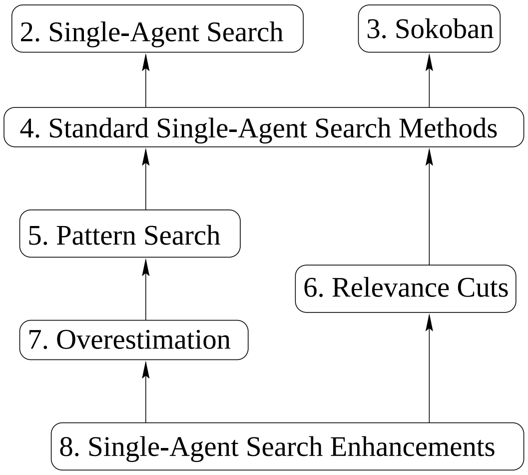


Figure 1.1: Chapter Dependencies

- A traditionally successful method for overestimation (WIDA* [Kor93]) is shown to fail in Sokoban. An explanation for this phenomenon is given. An alternate, domain-dependent method, driven by the dynamic knowledge gathered with the pattern searches, is shown to yield significant improvements in search efficiency.

- A classification of single-agent search enhancements is presented. It reveals interesting insights into the strengths and weaknesses of certain fundamental approaches to enhancing search algorithms.

- Control knowledge and control functions, new concepts in single-agent search, are proposed. The distinction between task and control knowledge allows for a cleaner treatment during design, implementation and tuning of search enhancements.

- A framework for single-agent search enhancements is given. Four basic types of search enhancements are identified.

1.3 Organization

Figure 1.1 contains a graph showing these inter-chapter dependencies. After an introduction to single-agent search in Chapter 2, Chapter 3 introduces the puzzle game Sokoban in detail. Chapter 2 will be useful when reading through parts of Chapter 3, but it is not essential. Readers familiar with single-agent search and/or Sokoban may wish to skip Chapter 2 and Chapter 3, respectively.

Chapter 4 examines the performance of the standard single-agent search techniques that are available in the literature and shows how to enhance macro moves with off-line precomputation. This chapter also lays out the experimental setup used throughout this thesis. It is fundamental to the understanding of either of the following three chapters in terms of methodology and terminology.

Chapter 5 introduces *Pattern Search*, a method that dynamically learns how to avoid deadlocks and improve the lower bound.

Chapter 6 discusses a new forward-pruning technique called *Relevance Cuts*.

In Chapter 7, the possibilities for overestimation are explored. The reader of this chapter should be comfortable with ideas and terms defined in Chapter 5.

Chapter 8 shows how these techniques fit into a framework that extends the traditional view on single-agent search. To get the most from this chapter, the reader should be well versed with single-agent search (Chapter 2), the standard single-agent search enhancements (Chapter 4) and the new search enhancements from Chapter 5 to Chapter 7.

1.4 Publications

Chapter 4 "Using Standard Single-Agent Search Methods" is based on two papers. The first paper, "Sokoban: A Challenging Single-Agent Search Problem" [JS97], was presented at the workshop "Using Games as an Experimental Testbed for AI Research" at IJCAI'97, Nagoya, Japan. The second paper, "Sokoban: Evaluating Standard Single-Agent Search Techniques in the Presence of Deadlock" [JS98c], is a revised and updated version of the workshop paper. It was presented in 1998 at the Canadian AI conference in Vancouver, Canada.

Chapter 5 "Pattern Searches" is based on the paper "Single-Agent Search in the Presence of Deadlock" [JS98b] which was presented at AAAI'98, Madison/WI, USA.

Chapter 6 "Relevance Cuts" stems from the paper "Sokoban: Improving the Search with Relevance Cuts" [JS99b] which was accepted in 1999 for a special issue of the Journal of Theoretical Computing Science. This paper is based on an earlier version, "Relevance Cuts: Localizing the Search" [JS98a], which was presented in 1998 at "The First International Conference on Computers and Games", Tsukuba, Japan.

Chapter 8 "Single-Agent Search Enhancements" is based on the paper "Domain-Dependent Single-Agent Search Enhancements" [JS99a] which was presented at IJCAI'99, Stockholm, Sweden.

Chapter 2

Single-Agent Search

2.1 Purpose of Search

Real-world problems can often be abstracted into *models* where a state of the world is described mathematically. *State-transition rules* describe the conditions for the transitions between *states* in the model and the changes these transitions cause.

For example, the children's toy called sliding-tile puzzle can be modeled in the following way. A state consists of the current location of the tiles and the empty space. The state-transition rules define that any of the up to 4 neighboring tiles can be pushed into the empty space. This simple description allows us to model the "real-world" problem of the sliding-tile puzzle economically.

Single-agent search assumes that only one "agent" is changing the state of the world, in principle, having total control within the rules defined by the model. *Adversarial search* assumes multiple (typically two) agents that both change the world to achieve opposing goals. We will restrict ourselves to single-agent search in this thesis.

State descriptions and state-transition rules (collectively, the "model") implicitly define a *graph* that is called the *problem* or *state space*. The nodes in this graph represent the states and the edges are transitions between states. A *problem* is given by a *start state* of the world and a description of at least one *goal state*. A *solution* would be a path leading from the start state to any of the goal states. A solution path indicates the sequence of transitions needed to transform the start state into the goal state. Some restrictions on that path might be given, such as the shortest possible path in terms of number of transitions.

In a problem description as defined above, finding a solution means finding a path in a graph. Different algorithms have been proposed that attempt exactly that. They follow different strategies.

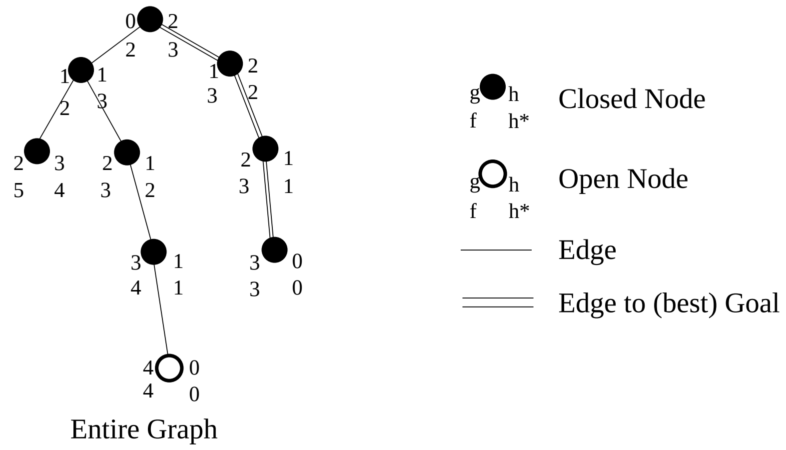


Figure 2.1: Example Search Graph (Tree) With Legend

2.1.1 Notation and Terminology

First we will introduce some notation and terminology to help explain the algorithms in the next sections. The exact *distance*[1] from a state s to the closest goal is usually referred to as $h^*(s)$. The function $h^*(s)$ is generally unknown. If we had a perfect function for h^*, finding an optimal path to a goal would be trivial and search would be unnecessary. The *heuristic function* $h(s)$ estimates $h^*(s)$ and is said to be *admissible*, if $h(s) \leq h^*(s), \forall s \in S$, where S is the set of all states in the search space. In other words, $h(s)$ is a *lower bound* on $h^*(s)$. The function $h(s)$ is called *consistent*, if $h(s_1) \leq h(s_2) + c(s_1, s_2), \forall s_1, s_2 \in S$, with $c(s_1, s_2)$ being the cost to get from s_1 to s_2. Consistency means that each transition having cost $c(s_1, s_2)$ can decrease the heuristic value by at most that cost. Consistent heuristics are necessarily admissible. For any of the goal states $h(s_g) = 0$, where $s_g \in G$ and G is the set of all goal states and $G \subseteq S$. The value $g(s)$ is the cost of all the transitions (or *actions*) performed to reach s from the start state s_0, therefore $g(s_0) = 0$. We will also use the function $f(s)$, which is defined as $f(s) = g(s) + h(s)$. Intuitively, $f(s)$ is the estimate of the total distance from the start state to a goal state via s. $f^*(s)$ is defined as $f^*(s) = g(s) + h^*(s)$. Thus, $f^*(s)$ is the actual cost along an optimal path through the state s. We will call a generated node in a search graph *open* if all of its successors have not yet been generated by the search algorithm and *closed* otherwise.

Figure 2.1 shows an example of a state space which we will use throughout this chapter to illustrate the different search algorithms. Nodes are marked with h and h^*, and with g and f, as indicated in the legend of Figure 2.1.

[1] Cost and distance will be used interchangeably, since we assume cost to be equal to distance. This simplifying assumption does not invalidate the generality of the following statements, since distance could be defined differently.

2.2 Algorithms

Many different algorithms have been proposed to traverse search spaces. We will concentrate on the most important and relevant to this thesis. We will start with three uninformed searches: random walk, breadth-first search and depth-first search. We then move on to informed searches, like A*, that use additional information in the form of heuristic knowledge to guide the search. This section closes with some intuitive explanations of search spaces and heuristic functions.

2.2.1 Uninformed Search

Random Generation and Random Walk

Random walk does what the name suggests: The algorithm walks randomly through the search space, choosing any of the neighbors. This might sound silly, but it can be a good idea if the goal density (the ratio of goal to non-goal states) is high enough, the quality (cost) of a solution is not a major issue, and little or no knowledge about the problem domain is available. Systematic search algorithms can suffer from problems such as space requirements, cycles, transpositions, and infinite paths – problems that are almost no issue in random algorithms. However, random algorithms are easily outperformed by more systematic approaches when searching for good quality solutions, or searching in large search spaces with low goal density, or when high-quality knowledge about the problem is available.

Figure 2.2 shows pseudo code for a random-generation algorithm. Instead of choosing a neighbor, it randomly selects any of the *open* nodes. `OpenStorage` refers to a data structure that can simply hold states visited, such as a list. The routines `Store` and `SelectRandomly` store and retrieve as well as remove states from that data structure. `Empty` tests if the data structure still contains states. `Child` expands a state (creates all possible successors by applying all legal actions) and `Solution` tests if a state is a goal state.

Some humans practice a form of random walk when they try to find, for example, baby food in a big department store that they visit for the first time. Without knowledge of where the products are located, most of their steps take them in a random direction, eventually reaching the shelf with the baby food.

Breadth-First Search

Intuitively, breadth-first search traverses the search space systematically by visiting all the nodes that are closest to the start state before visiting the ones further away, hence *breadth-first*. Figure 2.3 shows an order the nodes of the example graph in Figure 2.1 are expanded.

Figure 2.4 shows the pseudo code for breadth-first search. It starts by storing the start state into a first-in-first-out (FIFO) queue that holds all *open* states, states that are due to be expanded.[2] Until a goal state is found, breadth-first search takes

[2] Note that `Store` for `OpenStorage` in the random walker and `Store` for `OpenFIFO` have slightly

```
RandomWalk( StartState )
{
  Store( OpenStorage, StartState );
  Success = FALSE;
  DO {
    CurrentState = SelectRandomly( OpenStorage );
    IF( Solution( CurrentState ) )
      Success = TRUE;
    ELSE
      FOREACH( Child( CurrentState ) ) DO
        Store( OpenStorage, Child( CurrentState ) );
  } UNTIL( Success OR Empty( OpenStorage ) );
  IF( Success ) RETURN( CurrentState );
  ELSE RETURN( NULL );
}
```

Figure 2.2: Random-Generation Algorithm

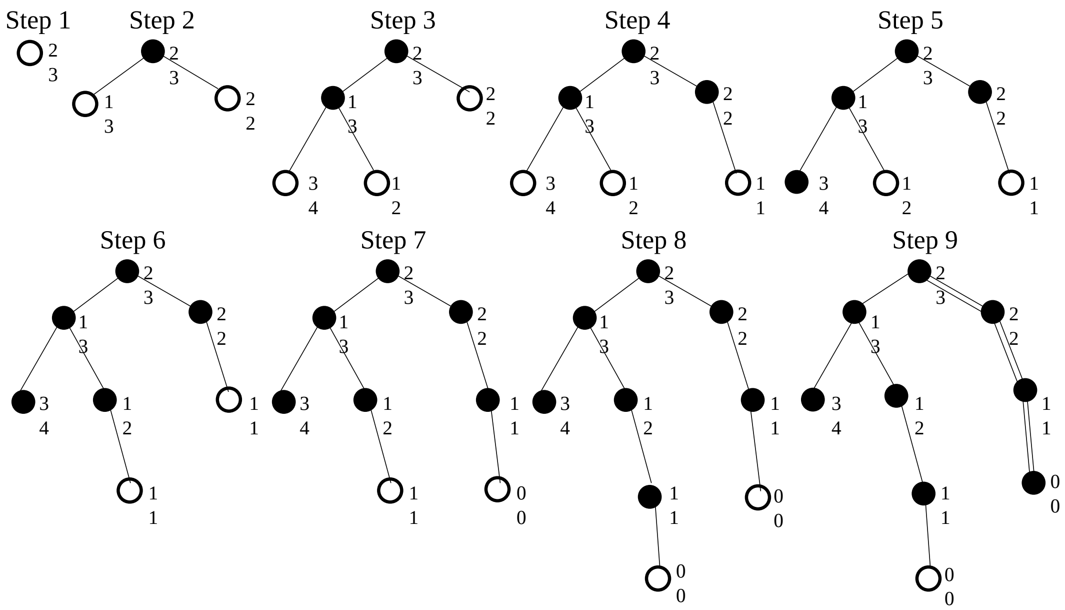


Figure 2.3: Illustration of Breadth-First Search

```
BreadthFirst( StartState )
{
  Store( OpenFIFO, StartState );
  Success = FALSE;
  DO {
    CurrentState = GetFirstIn( OpenFIFO );
    IF( Solution( CurrentState ) )
      Success = TRUE;
    ELSE
      FOREACH( Child( CurrentState ) ) DO
        Store( OpenFIFO, Child( CurrentState ) );
  } UNTIL( Success OR Empty( OpenFIFO ) );
  IF( Success ) RETURN( CurrentState );
  ELSE RETURN( NULL );
}
```

Figure 2.4: Breadth-First Algorithm (for Unit-Cost Actions)

the next state from the queue, expands it and stores all successors at the end of the queue.

Consider an analogy. When we drop our last match in a dark cellar, one would usually kneel down, touching the immediate area on the floor, slowly extending our reach, enlarging the area searched until we touch the match. This is in principle a breadth-first search. This approach seems reasonable, since we expect the match to be close by.

This simple algorithm is guaranteed to find an optimal solution, if all actions have the same (unit) cost. In the case of non-unit-cost actions, having found a solution, we have to continue to expand all states in the queue that have a cost less than the currently best solution. Two changes are necessary to achieve this. First, a simple check of each new state is added before it is put into the queue to see if its cost is less than the current best solution. Second, the algorithm stops only after the queue is empty. Dijkstra's shortest-path algorithm is the generalisation of breadth-first search to this case.

The immediate concern with this algorithm is its space requirement. The queue contains the entire search frontier, all the open states. This can quickly exhaust the memory capacity, even for moderately complex problems. For problems where that is the case, we need an alternative.

Depth-First Search

Depth-first search explores the search space from top to bottom across the graph (like columns in a table). More specifically, before searching the siblings of a node, all its children are searched. Thus, the deepest open nodes are expanded first. Figure 2.5

different semantics, each according to the kind of data structure they are operating on.

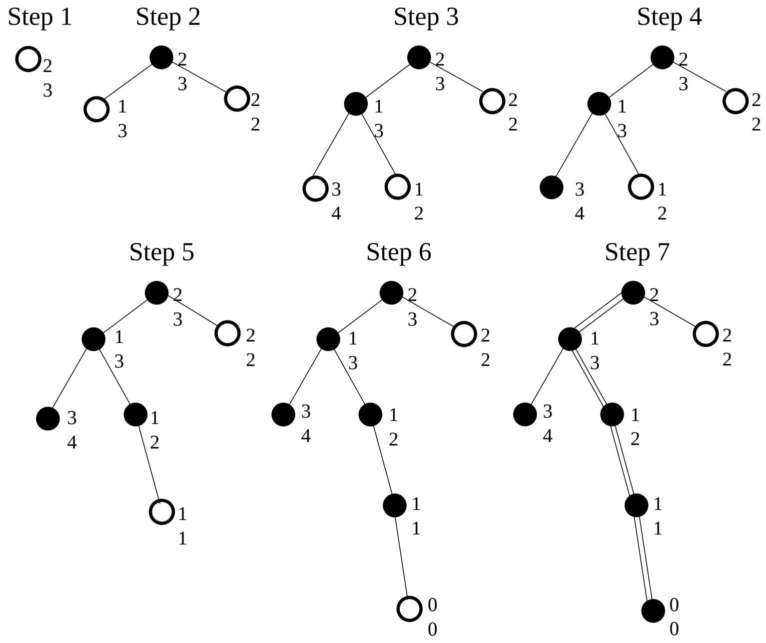


Figure 2.5: Illustration of Depth-First Search

shows depth-first search for our example graph in Figure 2.1.

To achieve the behavior from the sketch of the breadth-first algorithm above, we simply change the queue into a stack (last-in-first-out = LIFO)[3]. Figure 2.6 shows the pseudo code for the depth-first algorithm. If the algorithm stops after the first solution is found, we cannot be guaranteed to have an optimal solution. We will have to explore all children of the start state to make sure. However, there is one observation that can be used to improve the efficiency. Once a state s has a cost $f(s)$ larger than or equal to the currently best solution, exploring its successors will not yield any better solution and we can stop searching this part of the search space, if no actions have negative cost.

The advantage of depth-first search over breadth-first search is in the space requirements. The stack only holds all the neighboring states of the states on the current path. That means that the space needed for the stack is linear in the length of the current search path, which is logarithmic in the size of the tree. On the other hand, breadth-first search stores the search frontier, which, because of the exponential growth of the search tree, grows exponentially.

When Columbus set out from Spain to find India in the West, he did not waste time trying to find it in the immediate vicinity of Spain; he went straight West. We know today that this depth-first approach, because of the size of the goal (we assume it was America), makes perfect sense.

The obvious drawback of this algorithm is the possibility of cycles or infinite paths.

[3]The attentive reader will notice a slight inconsistency here. If the nodes are generated from left to right, as is usually assumed, and immediately placed on the stack, they would be expanded right to left. However, Figure 2.5 shows a left-to-right expansion.

```
DepthFirst( StartState )
{
  Push( OpenStack, StartState );
  Success = FALSE;
  DO {
    CurrentState = Pop(OpenStack);
    IF( Solution( CurrentState ) )
      Success = TRUE;
    ELSE
      FOREACH( Child( CurrentState ) ) DO
        Push( OpenStack, Child( CurrentState ) );
  } UNTIL( Success OR Empty( OpenStack ) );
  IF( Success ) RETURN( CurrentState );
  ELSE RETURN( NULL );
}
```

Figure 2.6: Depth-First Algorithm

If the problem domain allows for either of these, the simple algorithm of Figure 2.6 might fail to find a solution. If the algorithm had a notion of how much effort was spent on a certain path and an estimate about how far a potential solution was away, these two problems could be avoided.

Iterative Deepening

Iterative deepening is an attempt to rectify the problems depth-first search faces with infinite paths or loops, without incurring the excessive space requirements of breadth-first search. It was first introduced in [SA77, Kor85a], albeit for minimax-like algorithms and with a slightly different purpose.

The basic idea is to iteratively deepen the maximum depth a depth-first search can traverse into the search tree. If a certain iteration has finished without finding a goal, the maximal depth is increased and the depth-first search is restarted. At first this might sound like a lot of wasted work, but since the search-tree size grows exponentially with the depth, the size of the tree is dominated by the effort spent in the last iteration. Thus, all previous iterations search a relatively small portion of nodes when compared with the current iteration. Additionally, if a goal is found, it must be an optimal goal, since the previous iteration searched all nodes reachable with less moves then the current iteration allows (given unit-cost actions).

Iterative Broadening

Iterative broadening [Gin93a] is to breadth-first search what iterative deepening is to depth-first search. The number of successors explored at each node in the tree is restricted to a fixed portion (or alternatively a fixed number) of all successors. If no solution is found, the search can be restarted with more successors considered at each node. It is important to note here that the search-tree size growth is only dampened; it still grows exponentially!

Since iterative broadening does not impose depth limits *per se*, loops and infinite paths can lead to problems if not searched in a breadth-first manner. To avoid excessive space requirements, one can use the iterative deepening idea in connection with iterative broadening. However, if a search iteration failed to find a solution, it is hard to decide if the search should broaden or deepen the search efforts. Very little research has been done to investigate mechanisms that might be useful to control such hybrid searches.

Beam Search

Beam search [Win92, Bis92] restricts the number of open states per level in the tree to a constant – the beam into the search tree. Obviously, the search trees are only growing linearly in depth. Even though the idea might seem appealing at first, it comes with its own set of problems that are similar to the hybrid iterative approaches. If a search returns failure, how much wider should the search beam be? Which nodes should be kept at a particular level? These and similar issues are under investigation (For an example see [Zha98].).

2.2.2 Informed Search

All previous algorithms had no more information about a state other than the cost needed to reach it from the start state. Informed algorithms use additional knowledge to estimate how far away a solution is. This domain-dependent knowledge is encoded into a heuristic function. It returns an estimate of the distance to the goal for any arbitrary state. This heuristic function is called admissible if it never overestimates the distance (or cost) from any state in the search space to the closest goal. This estimate is also called a lower-bound estimator.

To get a lower-bound heuristic, one can remove constraints from the original set of rules for the domain and use this simpler problem to come up with an optimistic estimate on the cost to achieve a solution. For example, in the sliding-tile puzzle (see Section 2.6.1), we might choose to ignore the rules that only one tile can be at one square at a time and that a blank has to be beside a tile to be moved. With these two relaxations of the rules of the game, we get the Manhattan distance heuristic (see Section 2.3.3). Of course, the more simplifying (or ignorant) the assumptions are, the greater the error between the lower-bound estimator and the real cost to a goal.

The fewer the simplifications, the smaller the error between h and h^* will potentially be. Taking more details of the domain into account makes the lower-bound estimator more expensive to compute. However, since decreasing the error in the lower-bound estimator means a more efficient search[4], the gains in efficiency can offset the more expensive lower-bound calculations.

Many of the good lower-bound functions used for specific domains, such as the sliding-tile puzzle, are hand-crafted. Often, they result from clever reductions into

[4]This relationship will become more apparent in upcoming sections.

```
BestFirst( StartState )
{
  Insert( OpenSortedList, StartState );
  Success = FALSE;
  DO {
    CurrentState = GetBest( OpenSortedList );
    IF( Solution( CurrentState ) )
      Success = TRUE;
    ELSE
      FOREACH( Child( CurrentState ) ) DO
        InsertSortedOnH( OpenSortedList, Child( CurrentState ) );
  } UNTIL( Success OR Empty( OpenSortedList ) );
  IF( Success ) RETURN( CurrentState );
  ELSE RETURN( NULL );
}
```

Figure 2.7: Best-First Algorithm

functions that are easy to compute or approximate. The Manhattan distance is one such function: a number of table lookups and a summation are sufficient to calculate it. Furthermore, one can incrementally update the Manhattan distance as moves are made. This can result in effective *and* efficient implementations.

However, these good lower-bound functions are application-dependent. Each new application domain requires new efforts to find good heuristics. For large and non-intuitive domains this can be a hard problem. Holte *et.al.* [HPRA96] suggest using hierarchical searches to establish lower bounds on distances to goals. By abstracting the *real* search space (often by simplifying it), a smaller search in an abstract search space can produce a lower bound on the number of steps required to reach a goal in the real search space. Geffner [BG98, BG99] suggests a similar approach for his state-space planner.

Naive (Pure) Best-First Search

Best-first search always expands the *best* open node next. "Best" is defined with respect to some measure, typically the estimated distance to the closest goal state $h(s)$. Thus, at each step a best-first search expands the node that it believes to be closest to a goal. This behavior can be achieved by keeping all open states in a sorted list, ordered by the estimate of the distance to the goal (see pseudo code in Figure 2.7).

Unfortunately, this algorithm is not guaranteed to find an optimal solution. The search might be misled by an optimistic estimator for a path to a non-optimal solution. For example, if the heuristic returns a distance of 1 on the path to a non-optimal goal (see Figure 2.8), the path to an optimal goal is ignored and the suboptimal goal is found first. Best-first search will follow this suboptimal path to arbitrary depth. A closer goal was ignored because of a slightly larger estimate of the distance to the

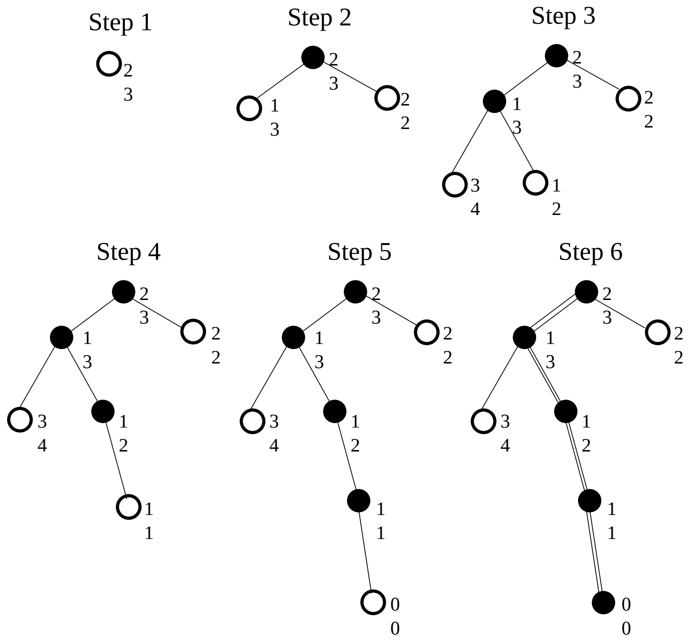


Figure 2.8: Illustration of Best-First Search

goal on an optimal path. This happens because the cost of actually getting to the current state from the start state is ignored.

A*

A* [HNR68] is a best-first algorithm which uses $f(s)$ as the measure for "best". By taking both the actual cost of getting to a state and the admissibly estimated distance to a goal into account, A* is guaranteed to find an optimal solution. Furthermore, A* handles search graphs, not just trees. This can lead to important efficiency gains when identical parts of a search tree (or cycles) are detected and multiple traversal of these parts is avoided. Unfortunately, this comes at a price: every time a node is generated, it must be checked to see if it was already visited or generated. Two lists are used to keep track of open and closed nodes, an OPEN list and a CLOSED list, respectively.

The description of A* in Figure 2.9 shows a simplified version that ignores all the details we have to take care of when we want to connect parent and child nodes in the graph. If a shorter path to a node is found, `PropagateG` is used to update the improved g value in all the successors of that node. Note that the function `Get` does not remove the node from any of the lists. `InsertSortedOnF` sorts the states in the queue according to the f value.

It is interesting to note that by appropriately choosing values for g and h, A* can behave like any of breadth-first, depth-first or the naive best-first algorithms. Setting the cost of an action to 1 and h to 0, A* defaults to breadth-first search. If we set the cost of an action to -1 and keep h at 0, depth-first behavior results. And finally,

```
A_STAR( StartState )
{
  StartState.g = 0;
  Insert( OpenSortedList, StartState );
  Success = FALSE;
  DO {
    CurrentState = GetBest( OpenSortedList );
    IF( Solution( CurrentState ) )
      Success = TRUE;
    ELSE
    {
      FOREACH( Child( CurrentState ) ) DO {
        IF( IsIn( OpenSortedList( Child( CurrentState ) ) ) )
        {
          OldState = Get( OpenSortedList( Child( CurrentState ) ) );
          OldState.g = min( OldState.g, Child( CurrentState ).g );
        }
        ELSIF( IsIn( ClosedList( Child( CurrentState ) ) ) )
          PropagateG( Get( ClosedList( Child( CurrentState ) ) ) );
        ELSE
          InsertSortedOnF( OpenSortedList, Child( CurrentState ) );
      }
      Insert( ClosedList, CurrentState );
    }
  } UNTIL( Success OR Empty( OpenSortedList ) );
  IF( Success ) RETURN( CurrentState );
  ELSE RETURN( NULL );
}
```

Figure 2.9: A* Algorithm

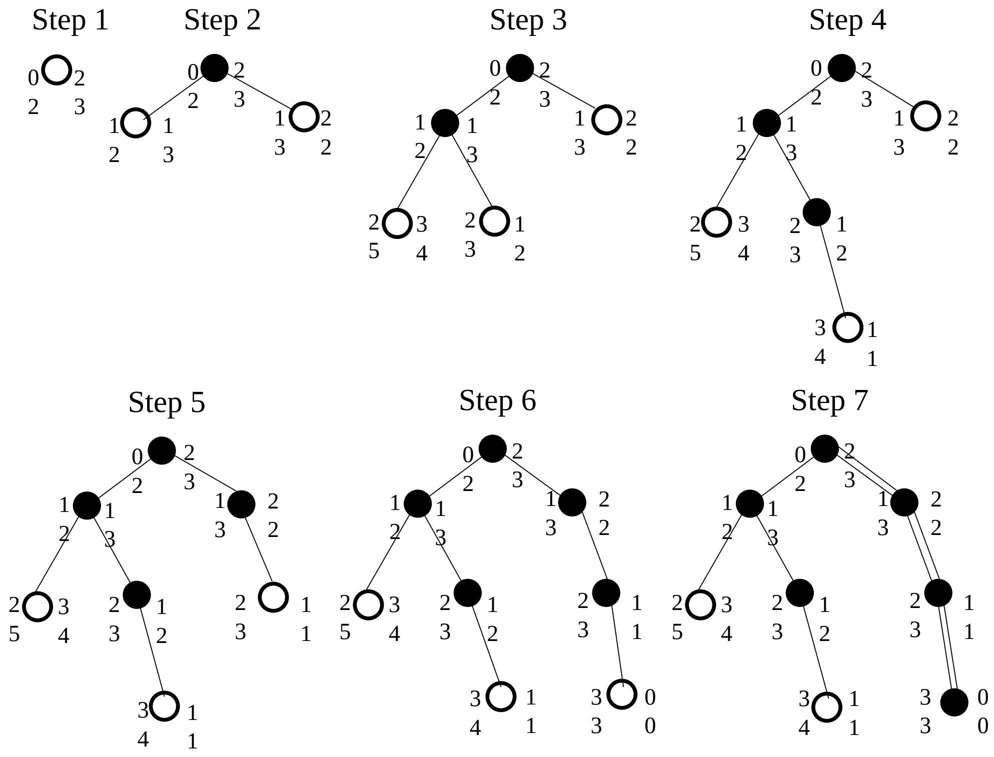


Figure 2.10: Illustration of A*

by setting the cost of actions to 0 (resulting in g being 0) and using the normal h, we achieve the naive best-first behavior.

The maintenance and size of the OPEN and CLOSED lists, with the expensive `Get` and `InsertSortedOnF` operations, are the main drawbacks of A*. Even moderately complex problems can bring the space requirements beyond the acceptable.

IDA*

Korf [Kor85a] applied the idea of iterative deepening to A*. The resulting algorithm (see Figure 2.11), Iterative Deepening A* (IDA*), traverses the search tree in a depth-first manner, iteratively deepening the tree. Each iteration of IDA* tries to find a solution with a path length equal to PathLimit. For the first iteration, PathLimit is set to the heuristic estimate for the start state (h(s)). If the heuristic is admissible, any node s with $g(s) + h(s) = f(s) > PathLimit$ cannot be on a solution path of length PathLimit and can therefore be ignored (pruned from the tree in this iteration). Exhaustively searching the tree during an iteration and not finding a solution is proof that no solution with length PathLimit exists. PathLimit is increased and a new iteration started. Eventually, we will increase PathLimit to a value that is as large as the optimal solution length (cost). During this last iteration we will find an optimal solution.

Is this approach efficient?

```
IDA_STAR( StartState )
{
  PathLimit = H( StartState ) - 1;
  Success = FALSE;
  DO {
    PathLimit ++;
    StartState.g = 0;
    Push( OpenStack, StartState );
    DO {
      CurrentState = Pop( OpenStack );
      IF( Solution( CurrentState ) )
        Success = TRUE;
      ELSIF( PathLimit >= ( CurrentState.g + H( CurrentState ) ) )
        FOREACH( Child( CurrentState ) ) DO
          Push( OpenStack, Child( CurrentState ) );
    } UNTIL( Success OR Empty( OpenStack ) );
  } UNTIL( Success OR ResourceLimitsReached() );
  IF( Success ) RETURN( CurrentState );
  ELSE RETURN( NULL );
}
```

Figure 2.11: IDA* Algorithm

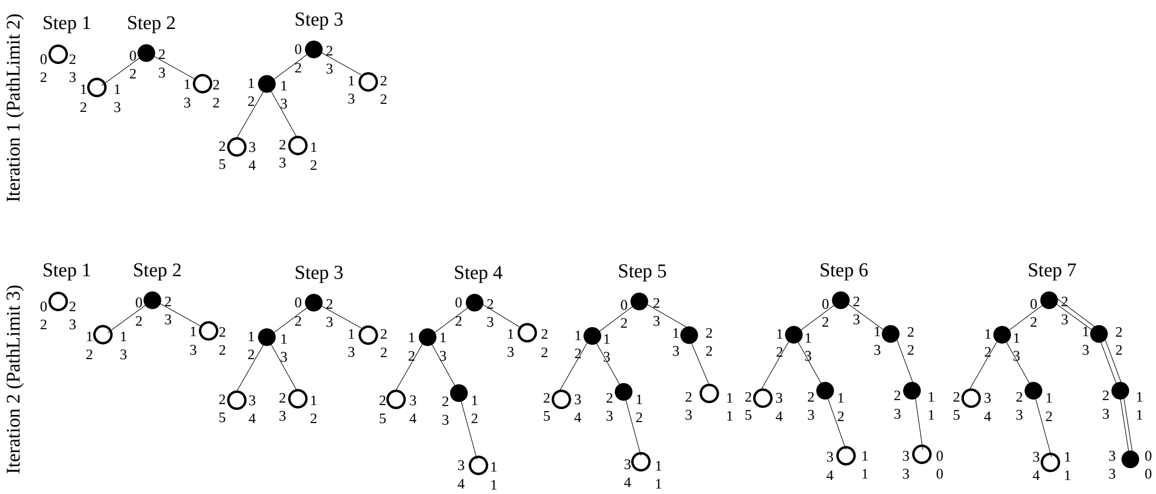


Figure 2.12: Illustration of IDA*

- Each iteration is a depth-first search, restricting the space requirements to logarithmic space in the size of the search tree, whereas A* needed linear space in the size of the search tree.
- No expensive list operations are needed anymore, lists are replaced with a cheap stack.
- With the limit on the solution length, an additional cutoff criterion is given that controls the size of the search tree.
- Since we are dealing with trees that grow exponentially in size, earlier iterations are usually small enough to be virtually negligible in cost compared to the last iteration.

These four reasons make IDA* a viable contender for practical applications in search.

The efficiency of IDA* depends directly on the quality of the heuristic function h. If $h = h^*$ the search would simply walk to the solution. It is important to note that the quality of the heuristic function depends on the average error over all the states in the search space, not just the root node. Even if the root node is estimated perfectly (no error), the search might not be able to find a solution because the heuristic is poor in the rest of the search space. In the worst case, $h = 0$ for all states, IDA* will degenerate to a series of depth-limited depth-first searches.

Depth-First Branch and Bound

IDA* starts with a lower bound on the solution length and increases this lower bound each time it proves that no solution with this lower bound exists. Depth-first branch and bound (DFBB) [LW66] starts with an upper bound on the solution length. The upper bound is used to prune parts of the search space that cannot contain a solution better than the current best solution. That means that whenever DFBB encounters a node s in the search space that has an f-value ($f(s) = g(s) + h(s)$) equal or larger than the best solution found so far, s gets cut off. DFBB traverses the search space in a depth-first manner. Whenever it finds a new goal when attempting to expand it, this goal must be better than the previously found. The cost of the new goal is used to adjust (lower) the upper bound and the search continues. Depth-first branch and bound depends on a high goal density, otherwise it will suffer from the same problems as depth-first search.

Bidirectional Search

Nothing forces us to solve the problem in a "forward" direction [Nil80]. Why not search "backwards", starting from the goal state and attempting to find a path to the start state? Choosing the right direction ("forward" or "backward") can lead to significant savings, since tree shapes might not be symmetric and a forward tree might be larger than the corresponding backward tree. However, so far we are not talking about something new, just that the search direction might be an issue. Backward search would use inverse actions to create all possible predecessors of a node in a

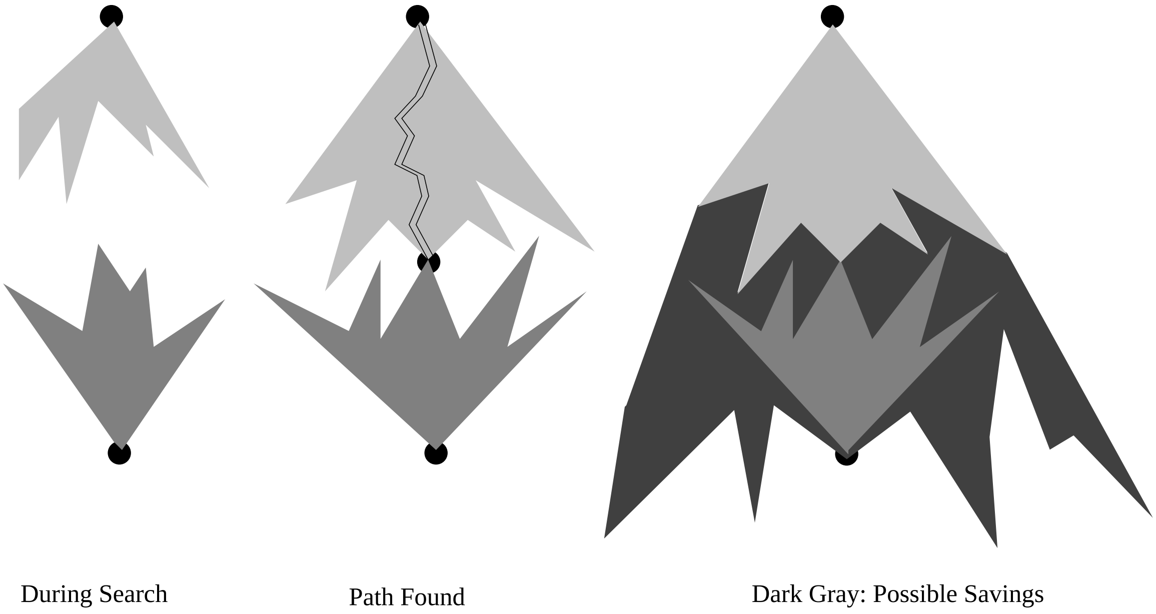


Figure 2.13: Illustration of Bidirectional Search

forward sense, which are the successors in a backward sense. We would also have to exchange goal with start state(s).

But why do we search only in one direction? Could it be beneficial to search from both sides? Bidirectional search [Poh71] proceeds to deepen the tree from both sides until both trees intersect at one node, connecting a path between the start and a goal state. This path is not necessarily an optimal path. The search strategies for the forward and backward search do not need to be the same. Depending on the different search-space properties, different strategies might be chosen for the different directions.

Figure 2.13 shows the search trees for the two directions. As they grow towards each other, they will eventually meet. A unidirectional approach would have to expand a deeper tree with a potentially larger number of nodes than the two smaller trees together. The dark gray area in the third part of the figure shows what the potential savings could look like.

All this sounds rather convincing. The question is why is this approach not widely used? There are several problems with bidirectional search. For a long time it was assumed that it was hard to be able to make the search frontiers meet. However, Kaindl and Kainz [KK97] show that finding a solution was not so much the problem as finding an optimal solution. The search spends a lot of effort making sure an optimal solution is returned. Furthermore, the savings shown in Figure 2.13 are not necessarily achievable, if the unidirectional search is efficient. If, for example, IDA*'s last iteration is small because of good move ordering, the savings achievable with bidirectional search are small.

Furthermore, it is a problem to detect when search frontiers even meet, since at least one frontier has to be kept in memory to find intersections. Since trees grow exponentially, search frontiers and, thus, space requirements do too, a major drawback for bidirectional search.

Backward searches often face another problem. Since the goal state is not necessarily one state, but a set of states, the backward search could potentially have several possible start states. This increases the amount of work to be done and simultaneously decreases potential savings. Other, more practical reasons might include difficulties with reverse actions and the overhead of finding, tuning and coding the additional heuristics.

2.2.3 Choosing the *Right* Algorithm

When presented with the choice of which algorithm to implement for a certain domain, one might be confused by the multitude of different approaches and ideas behind the algorithms. So what drives the selection of an algorithm? What are the properties of the search space that are used to decide which algorithm to use?

The first choice between informed and uninformed algorithms depends on the availability of domain knowledge. Usually, informed searches perform better than uninformed. Therefore, if knowledge is available, informed algorithms are the preferred choice. As long as the search space fits into memory and the overhead of maintaining the OPEN and CLOSED lists is no concern, A* is the choice. However, if memory or list maintenance is a concern, IDA* is the preferred choice.

Now, when would a branch and bound search be useful? Branch and bound searches operate on the fact that we can easily find *a* solution, but want to improve on the quality of that solution. A high density of goal states in the search space is needed if DFBB is used. Otherwise DFBB will degenerate into a depth-first search.

Random walk algorithms are of use when faced with huge search spaces where we have little or no knowledge of the search space and we are looking for just any solution, and not necessarily a high-quality solution. Recent interest in random walk applications was sparked by advances in the satisfiability (SAT) domain, where WALKSAT [SKC94] seems to perform rather well. Other random walk algorithms are heuristically guided, but use the random element to avoid local minima. Genetic algorithms, and certain hill climbers, such as simulated annealing, belong to that group.

2.3 Enhancements

The description of the algorithms in the previous section conveyed only the basic ideas. Most of these algorithms are used in connection with powerful enhancements. Search enhancements potentially increase the search efficiency by orders of magnitude, depending on the problem domain and algorithm. Often, the choice of the right algorithmic enhancement(s) is more difficult and crucial to the performance of the program than choosing the right algorithm. This problem will be discussed in more detail in Chapter 4.

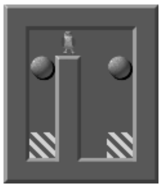


Figure 2.14: Example Problem to Illustrate Transposition Tables and Macros

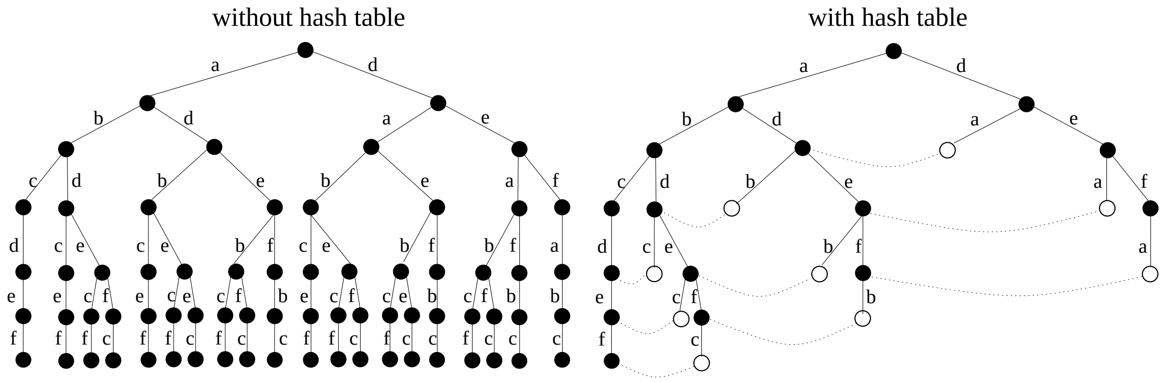


Figure 2.15: Impact of Transposition Tables

2.3.1 Transposition Table

Even though search spaces are generally graphs, most search algorithms treat them as trees. If a state can have several predecessors, this can lead to duplicate work. The search could revisit nodes and even entire subtrees several times. These "transpositions" are detected using a large transposition table [SA77][5] in which useful information about previously visited nodes is stored. Before expanding a node, the transposition table is consulted, and if valid information is found, it is used to potentially curtail the search. Transposition tables are usually implemented as hash tables.

Furthermore, when iterative deepening is used, the transposition table serves to store information that can be used to make subsequent iterations more efficient (see Section 2.3.2 "Move Ordering").

Consider the position in Figure 2.14: Two stones need to make three moves in a row. We will use *a,b,c* for the moves of the left stone and *d,e,f* for the right stone. The moves of each stone have to be made in sequence, but can be interleaved in any way between the two different stones. Figure 2.15 shows what happens if the search is enhanced with a transposition table. Solid nodes represent nodes searched normally, while light nodes represent cutoff nodes because of a transposition table match, with dotted lines connecting identical positions in the tree. For example, the top most light node is reached with the move sequence *d,a* which results in the same position that was previously searched after the moves *a,d*.

[5]Another way of detecting transpositions involves finite state machines [TK93].

Another way of looking at the function of the transposition table is by describing it in terms of the heuristic lower bound and IDA*. Each state s searched in IDA* must have the following property: $g(s) + h(s) = f(s) <= PathLimit$, otherwise the search would not consider this state. When a state was searched exhaustively and the search backs up with failure, we have proven that our heuristic function $h(s)$ was off. We know now that $g(s) + h'(s) = f'(s) > PathLimit$, or $h'(s) > PathLimit - g(s)$, or $h'(s) >= PathLimit - g(s) + 1$. When storing $h'(s)$ in the transposition table, we allow the search to improve on the heuristic value $h(s)$ every time it revisits the state s. The value stored in the transposition table is used to improve the lower bound. This dynamic improvement of the lower bound leads to additional cutoffs in the search in two ways.

- Within an iteration, revisiting a state with the same or larger $g(s)$ allows us to improve the lower bound with a transposition table lookup. The improved lower bound will be enough to cause a cutoff ($g(s) + h'(s) = f'(s) > PathLimit$).

- If the search revisits the state s with a lower $g(s)$, no cutoff can happen, the search will proceed. However, in the next iteration, if we visit node s in the same order, the $h'(s)$ stored in the transposition table is now sufficient to cause a cutoff when reaching s via a non-optimal $g(s)$.

This scheme also handles cycle detection. Rather than storing the new lower bound after we searched the subtree, we update the transposition table *before* descending into the tree. If we ever cycle back into a state that is on the current path, $g(s)$ must be larger than it was previously and thus, a cutoff will occur. No special code is needed to detect cycles.[6]

2.3.2 Move Ordering

Instead of visiting successors of a move in an arbitrary order, one can try to look at "good" successors first.

Move ordering is not used in best-first searches; the algorithm itself provides for a global ordering of the alternatives. In depth- and breadth-first searches, move ordering can lead to efficiency gains because goals are found earlier (left in the tree) rather than later (right in the tree). Reinefeld and Marsland [RM94] comment on the effectiveness of move ordering in single-agent search. For IDA*, ordering moves at interior nodes makes no difference to the search, except for the final iteration. Because the final iteration is aborted once a solution is found, finding a solution early in the final iteration can significantly improve the performance, especially considering that the last iteration is potentially the largest.

The information used to order moves can come from different sources, usually domain-dependent knowledge. Sometimes domain-independent knowledge gathered

[6]Some readers might feel uncomfortable with this statement, because there is the remote possibility of collisions in the hash table that overwrite entries, resulting in undetected cycles. While this is true, this should happen very infrequently and in such rare cases we are willing to go the extra depth until $g(s)$ is large enough to curtail the search.

in the search tree (*e.g.*, tree sizes, tree depths...) can be useful. In the case of iterative deepening, move ordering information is passed from one iteration to the next by means of the transposition table.

The alpha-beta algorithm, used in adversarial search, relies on good move ordering to achieve maximal efficiency by establishing good bounds early (worst case tree size is w^d, best case with perfect move ordering is $w^{d/2}$, where w is the branching factor and d is the depth of the tree). In single-agent search, move ordering can be much more crucial. If a depth-first search had perfect move ordering, it could go straight to the goal. In the worst case, depth-first search spends exponential effort. Of course, perfect move ordering does not exist in search, since it would entirely obsolete search *per se*. However, the better the move ordering the more efficient the search, if the overhead of achieving this ordering does not offset the gains.

2.3.3 Pattern Databases

Lower-bound functions provide the search with guidance in the form of cost estimates for reaching a goal from a position in the search. These functions usually ignore some of the domain constraints to allow for efficient implementations. A common approach is to decompose the total cost of solving the problem into solving independent subtasks. These subtasks usually consist in moving physical objects to goal squares. For example, for the sliding-tile puzzles, the distance of each tile to its target square is summed to produce a lower bound on the total number of steps required to solve the entire problem. This heuristic is called *Manhattan distance*. The Manhattan distance assumes that every tile can directly move to its goal without detour.

Lower-bound functions following this approach can be very efficiently computed and are even amenable to incremental updates during the search because of the independence of the subgoals.

However, the challenge of these puzzles and real-world problems lies in the interactions of the subgoals. Neglecting them creates poor lower bounds. An improvement to the sliding-tile puzzle's Manhattan distance, called linear conflicts, was proposed by Hannson *et al.* [HMY92]. It uses the observation that one of two neighboring squares that are in each others way to reach their goal square optimally has to make at least two non-optimal moves off the optimal path. Identifying two such squares allows us to increase the lower bound by 2.

Linear conflicts contain the core idea used for pattern databases [CS96]. Instead of looking at one of the (physical) elements or subgoals at a time, combinations of these elements (patterns) are used. For each of these patterns a precomputation determines the minimum cost to get all the elements of the pattern to their respective goals. The precomputation takes interactions of these elements into account and stores the costs in a database that can be queried during the search. The more elements are taken into account the more accurate the lower bound. One could even call the Manhattan distance a one-tile pattern database and the Manhattan distance plus linear conflicts a two-tile pattern database.

Culberson and Schaeffer [CS96] show results for the sliding-tile puzzle and Korf [Kor97] applies the technique to Rubik's Cube. The improvements in lower-bound

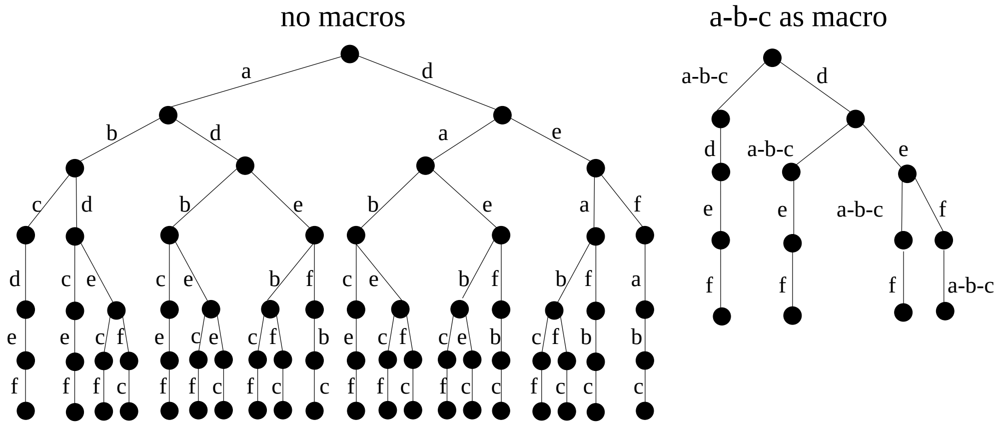


Figure 2.16: Impact of Macro Moves

quality lead to significant gains in the search efficiency.

Cazenave [Caz99] suggests an interesting improvement on the idea of pattern databases for the domain of Go. A core pattern is annotated with external conditions. The core patterns are defined by the physical arrangement of stones. External conditions are logical properties of the board around the core pattern that help to determine the state of the core pattern. The use of external conditions reduces the number of total patterns, because a large number of essentially irrelevant details are abstracted into a few rules.

2.3.4 Macro Moves

The search algorithms discussed so far treat all the moves equally. After making a move, all legal moves are considered as successors. These algorithms are therefore considering all sequences of moves even though their order does not matter.

Consider trying to solve the problem of driving to work in the morning. When trying to devise a plan to get from home to work, all the algorithms are considering sequences such as: leave-the-house, mow-the-lawn, open-garage, get-in-car, exit-car, mow-the-lawn, get-in-car... All actions are legal, but not necessarily related. The method of macro moves [Kor85b] is an attempt to group related atomic actions into higher level composed actions: macros. This can result in impressive search-space reductions. Special attention has to be paid to the impact these macro moves can have. They might influence the correctness and/or the completeness of the search as well as the ability of the algorithm to find optimal solutions.

Figure 2.16 shows the impact of the move sequence *a-b-c* being treated as a macro in the position of Figure 2.14. The effect on the search-tree size is visible, instead of exploring every possible combination of interchanging moves *a, b, c*, and *d, e, f*, the search visits less nodes and even the depth of the tree is reduced.

James [Jam93] builds on an idea from Iba [Iba89] and dynamically creates macros by "tunneling" peaks in the search-space landscape. Figure 2.17 shows what happens. Iba suggested tunneling from one valley in the cost landscape to the next, leading

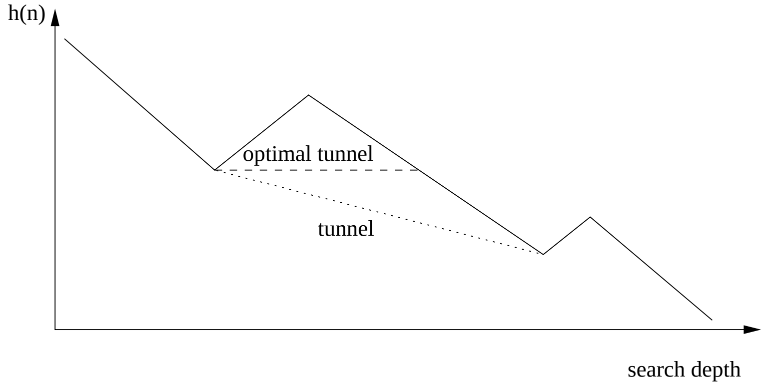


Figure 2.17: Tunneling: The Dynamic Creation of Macros

to long macros with many preconditions that are hard to match. James observed that difficulty and improved on the tunneling idea by suggesting the shortest possible tunnel that “drains” the water into the next valley. This results in fewer preconditions and the increased chance to match a macro.

It is important to note that tunneling changes the search space by creating macros that behave as shortcuts. They detour the error of the heuristic estimate, rather then decreasing it.

2.3.5 Summary

Enhancements to search algorithms can improve search efficiency dramatically. Depending on the application domain, those savings can be several orders of magnitude for every one of those enhancements. Each enhancement is implicitly benefiting from properties of the search space. Transposition tables work only if the underlying search tree is in reality a graph. Macro moves assume that treating several moves as one does not change the rest of the problem and ultimately its solvability (or completeness[7]). For related results see Culberson and Schaeffer [CS94].

Each new enhancement will have a limited scope of domains to which it applies. The No-Free-Lunch (NFL) theorem, that we will be talking about later, backs this

[7]Whereas completeness is an important theoretical consideration, we feel it has little or even no practical value. “Complete” algorithms use exponential time to ensure completeness. Since we face time constraints in practice, this theoretical completeness is of little value. Furthermore, it is always easy to construct a complete algorithm. After exhausting a predetermined time limit, any theoretically complete algorithm can be executed. This two-phase approach is also theoretically complete, but with a constant run-time overhead. In practice it is only important how many problems can be solved within the time limit. Complete algorithms have the nice property that they can prove that no solution exists. But again, if they could do that in a reasonable time frame, the more efficient algorithm would stop without finding a solution even earlier and the second phase of our hypothetically complete algorithm will prove that the problem has no solution. Because the first phase searches less of the search space, it will use less time than the second phase. Our two-phase algorithm will therefore take at most twice the time for this proof than the complete algorithm of the second phase would have used on its own.

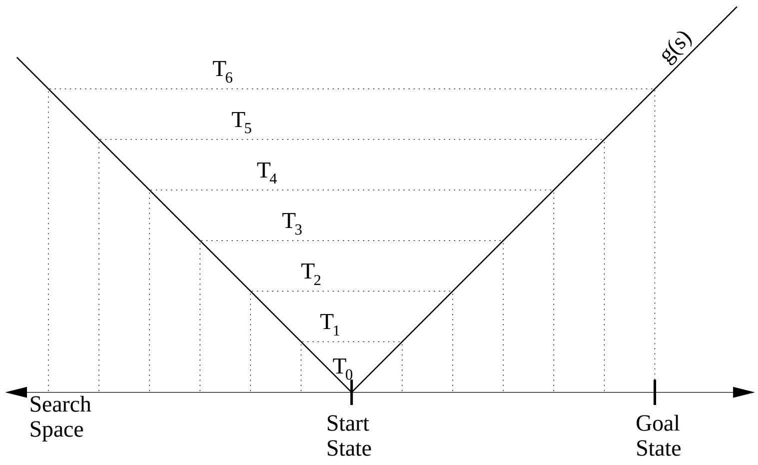


Figure 2.18: Search Space Without Heuristic Knowledge

intuitive statement up. There is no magic bullet, neither in the form of an algorithm nor as an enhancement, that could be used to solve every possible domain more efficiently.

2.4 Heuristic Functions and Search Spaces

This section is an attempt to underpin the formal concepts introduced earlier. Examples are used that will help the reader develop a more intuitive feeling for search spaces and how heuristic knowledge can help to guide algorithms through them.

Search spaces are multi-dimensional structures that are very hard to visualize. We will therefore use simplified search spaces here. Figure 2.18 shows a one dimensional search space along the horizontal axis of the graph. The function $g(s)$ shows the distance from the start state. Since search spaces are discrete structures, $g(s)$ should be a step function. For simplicity, we are using continuous functions here.

A breadth-first search would expand nodes in waves; after all nodes at a certain depth T_i are visited, the next depth is started until the last level, here T_6, is reached. It now depends on the order of expansion, when the goal state is found. If the search is "lucky", it visits states to the right first, finding the goal state without expanding states to the left of the start state.

Depth-first search is faced with a different problem. Assume our search space is much larger than shown. If depth-first search starts searching to the left, it will not find the solution until it has visited *all* the states to the left. However, starting to the right, it would get lucky. Note that in practice, with the many dimensions of a real search space, getting "lucky" would mean making the right decision many times – an unrealistic hope.

Figure 2.19 shows the same search space with good heuristic knowledge. In addition to $g(s)$, the distance from the start state, $h(s)$, the lower-bound estimate of the distance to the goal, is available. The knowledge is consistent with the location of the

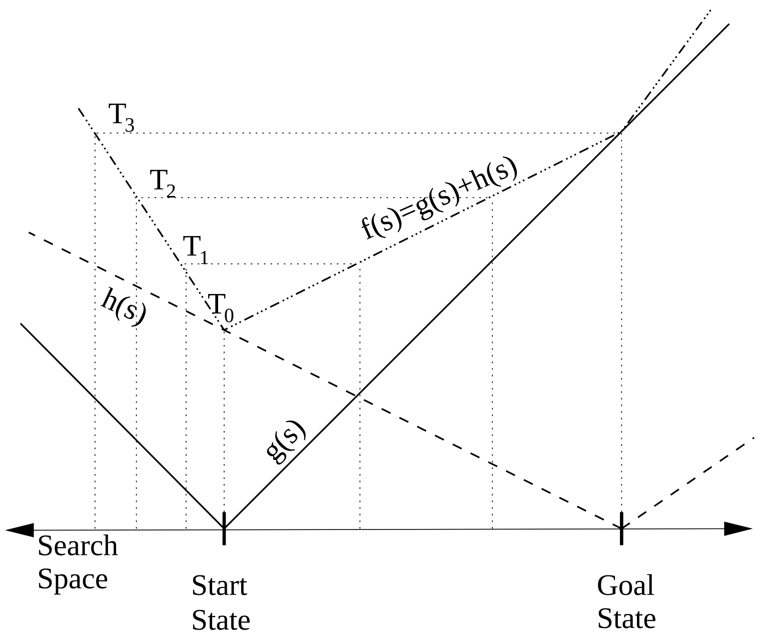


Figure 2.19: Search Space With Good Heuristic Knowledge

goal state, $h(s)$ decreases towards the goal state. A*, which uses $f(s)$ for guidance, will expand states in the search space in similar "waves" as breadth-first search does. Each new wave T_{i+1} is larger then the previous wave T_i. However, A* needs fewer waves and each wave-front is smaller than in breadth-first search, because heuristic knowledge allows A* to prune portions of the search space which would be visited by breadth-first search.

While Figure 2.19 shows an example with good heuristic knowledge, heuristic knowledge can be misleading. Figure 2.20 shows such an example. The path away from the goal looks initially better than the path to the goal state. Due to misleading knowledge more states are expanded in each wave. However, as long as the heuristic knowledge is admissible, A* will not expand more nodes than breadth-first search.

Figure 2.21 shows the ideal case. If the search knew the correct distance to the goal at each state in the search, it could directly go to the goal. As discussed earlier, this is not an interesting case for search, but it can show what better heuristic knowledge will asymptotically lead to.

Since Figures 2.18, 2.19, 2.20, and 2.21 show trivial, one-dimensional search spaces, they cannot convey the exponential growth of consecutive waves. Figure 2.22 shows two search graphs with the waves shown in different shades of gray. One could imagine these graphs to show the search space from the top. The exponential growth of each larger wave is now visible.

Improvements in the heuristic knowledge that is available to the search algorithms usually lead to two kinds of efficiency gains:

- Fewer waves: With an increase in the heuristic value of the start state, the difference between $h(start)$ and $h^*(start)$ decreases, leading to fewer iterations

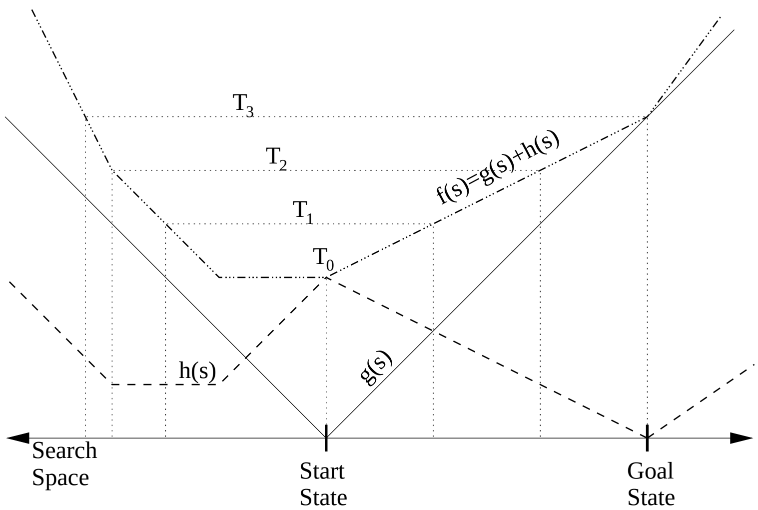


Figure 2.20: Search Space With Misleading Heuristic Knowledge

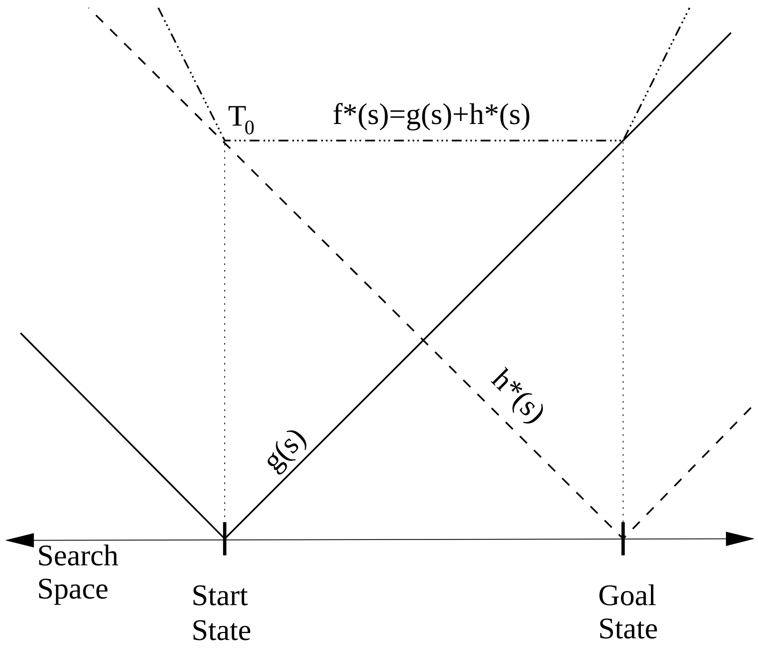


Figure 2.21: Search Space With Perfect Knowledge

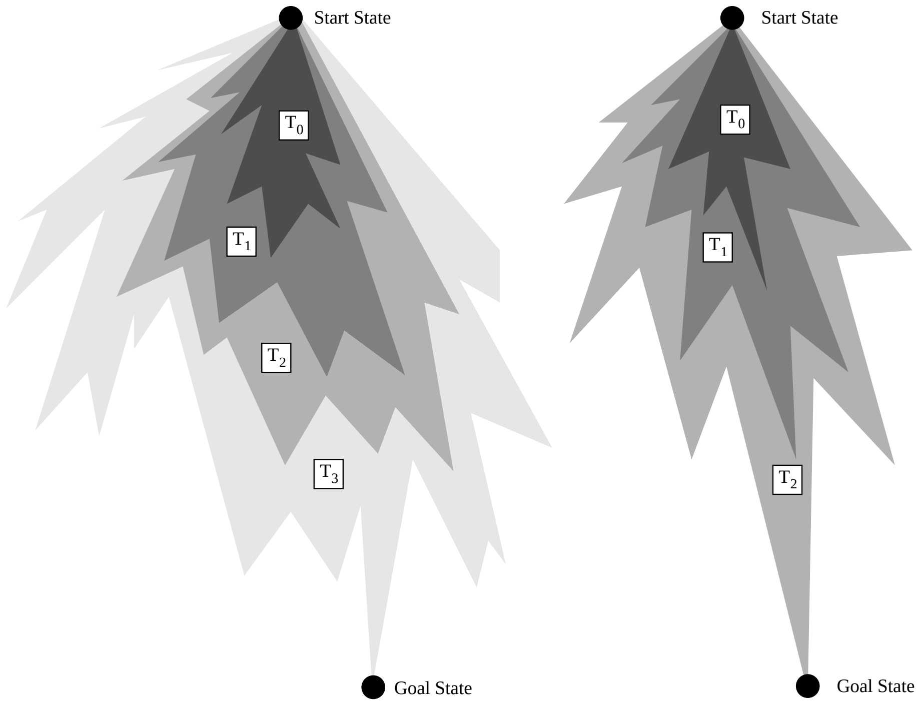


Figure 2.22: Examples of Search-Space Waves

(IDA*) or fewer waves (A*).

- Smaller waves: Improvements to the lower bound for many states in the search space lead to more cutoffs in the tree which in turn results in smaller waves.

For some problems the solution length is known. That is equivalent to an improved lower bound for the start state and helps to remove the initial iterations for IDA*, because it can start with the correct threshold.[8] However, the last iteration is still as large as it was before. Because of the exponential growth of the search tree, the last iteration dominates the effort of the search and thus, the savings are relatively small.

It is more important to improve the lower-bound function on average for the entire search space to remove large portions of the last iteration. These savings are potentially much larger. Figure 2.22 shows a combination of the two when comparing the left with the right figure: on the right side there are fewer iterations searched and there are fewer states in each of the iterations.

2.5 No-Free-Lunch Theorem(s)

Throughout the computing science literature, the quest for the Holy Grail can be found: a universal problem solver, universal function optimizer and alike. Some

[8]This trick has one side effect worth mentioning: The first action away from the start state can lead to a large reduction in the heuristic value, potentially larger than the cost of the action. The resulting heuristic is therefore not consistent.

people even claim to have found such a tool. Others are more modest in their claims; they restrict their statement to a certain class of problems, such as search problems. The last wave of such claims could be observed during the advent of evolutionary algorithms. This was, unfortunately, surely not the last.

2.5.1 Bad News

As seductive as the thought of a one-size-fits-all algorithm is, such an algorithm does not exist. Wolpert and Macready [WM96] prove with their No-Free-Lunch (NFL) theorems that all algorithms that search for an extremum of a cost function perform exactly the same when averaged over all possible cost functions. A "universally best" search algorithm would have to outperform all other algorithms on average. Wolpert and Macready show that if an algorithm A outperforms algorithm B on some cost functions, then B must outperform A on others. Culberson [Cul96] uses adversarial arguments to come to the same conclusion.

What does that mean? If we look at all possible cost functions (or, for that matter, search spaces), there exists no algorithm that can outperform all other algorithms. Worse still, all algorithms, even totally random searchers, will perform the same on average on all possible search spaces.

Extensions of these theorems in [Cul96] even show that learning does not work over all possible instances. Not even adaptive (learning) algorithms that try to extrapolate from what they have seen so far to guess into the future will work better, when averaged over all search spaces. Heuristic knowledge can also be only problem specific and not absolutely general. The claim of "universal" can clearly be rejected, in all cases.

2.5.2 Good News

However, not all is lost. By restricting our algorithm to domains where we have knowledge available, the knowledge can help to increase performance in that domain as compared to a knowledge-poor algorithm. Given the above reasoning, we trade the performance increase for "our" problem with a performance decrease in some other domain, but we are happy with that trade.

That means that algorithms and algorithmic enhancements work for certain problem domains. What are these domains, what makes them suited for a certain algorithm and not for another? Usually, the knowledge we are encoding in our algorithms reflects search-space properties that can be exploited by the search. For example, if we know that our search space is a graph and not a tree, the use of a transposition table can yield performance improvements.

The general strategy for tackling a domain is to look for certain search-space properties and exploit that knowledge for efficiency gains. Therefore, the question about the generality of a search enhancement (or conversely, how domain dependent is *this* enhancement) is not the proper question to ask. One should rather ask, what the properties of those search spaces are that the enhancement in question relies on to yield a performance increase.

2.6 Example Domains from the Literature

The following subsections introduce the domains used most often in research on single-agent search in the literature. The goal here is to introduce the reader to the general complexities, special properties, and issues of the search spaces of those domains.

2.6.1 Sliding-Tile Puzzles

The sliding-tile puzzles are a family of the commonly known toys, where a (usually square) matrix of tiles has to be ordered. In a 4x4 matrix of tiles, there are 15 tiles and one blank square. The tiles can only be moved into the blank. Other studied variations are the 24-puzzle (5x5), the 8-puzzle (3x3), and even the 19-puzzle (4x5, and really MxN).

A state of the 15-puzzle can be described with the location of all tiles. Each state can have a maximum of 4 legal moves if the blank is in the middle 4 squares, 3 if it is at the edge and 2 moves if the blank is in a corner. However, since one move led into the current position, the move unmaking it does not need to be considered. Therefore, the resulting effective branching factors are 3, 2 and 1, for the respective positions of the space. Edelkamp and Korf derive 2.13 as the asymptotic branching factor for the 15-puzzle [EK98].

For the 24-puzzle, Korf [Kor96] reports average solution lengths (for randomly generated instances) of over 112, for the 19-puzzle 71.5 and for the 15-puzzle 52.6. The 8-puzzle is small enough to be enumerated exhaustively [Sch67, Rei93]. The search spaces are almost 10^{25}, 10^{18}, 10^{13} and 10^{5} for the 24-, 19-, 15- and 8-puzzle, respectively.

The state-of-the-art systems solving sliding-tile puzzles use IDA* with transposition tables and improved Manhattan distance as the admissible heuristic. Improvements of the Manhattan heuristic are derived from the fact that there might be conflicts among different tiles when trying to push them straight to their respective goals (linear conflicts) [HMY92]. Taking this idea even further, Culberson and Schaeffer [CS96] suggest to use pattern databases that record the optimal number of moves required to push subsets of tiles to their goal positions. A unified view on this issue is that each of these approaches allows more and more of the real constraints to be used in the lower-bound calculation. Whereas the Manhattan distance assumes no restrictions in the tile movements, linear conflicts take the movements of up to 4 tiles into account and treat the rest as non-existent. Pattern databases consider even more tiles (constraints).

Current state-of-the-art search techniques and computers allow us to solve any random instance of the 15 and 19-puzzles within a reasonable amount of time. The 24-puzzle is still presenting a considerable challenge though.

2.6.2 Rubik's Cube

Rubik's Cube, the famous combinatorial puzzle invented by Erno Rubik in the late 1970s, is also used in the literature to investigate search algorithms and their enhancements [Kor85b, FMS+89, Pri93, Kor97].

Rubik's Cube has a search-space complexity of about 10^{19} and a median solution length of 18. The longest solution is believed to be no longer than 20 moves [Kor97]. Edelkamp and Korf [EK98] calculate the asymptotic branching factor to be 13.35, if a move can be more than a 90 degree twist. Programs to solve Rubik's Cube problems are very similar to programs solving sliding-tile puzzles. IDA* is used as the search algorithm and large pattern databases are used to achieve a good lower-bound estimator. Korf reports solving 10 random instances of the Rubik's Cube optimally [Kor97].

2.6.3 Mazes

Rao, Kumar and Korf [RKK91] introduce another domain into the literature: mazes. The task is to find optimal routes between two points in the maze. The complexity of mazes can be adjusted arbitrarily by scaling. Transpositions can be simulated by allowing mazes with a graph structure (holes in walls). Whereas Rao, Kumar and Korf [RKK91] used mazes of size 120 x 90, Kainz and Kaindl [KK96] used mazes of size 2000 x 2000.

The domain of mazes is interesting because of the property of the lower bound. When the Manhattan distance is used, the ratio between the correct distance h^* and the estimated distance h can be large. IDA* performs rather poorly, since many iterations are performed, without finding a solution. A*, because it keeps the entire graph in memory, should be a better choice.

2.7 Summary

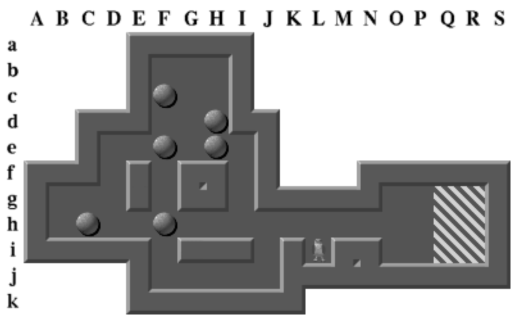
There are a wide variety of strategies for efficiently traversing search spaces. Uninformed searches traverse the search space blindly in a systematic fashion. Informed algorithms exploit knowledge about the search space to search more efficiently. Search strategies and algorithmic enhancements are chosen to exploit specific properties of the underlying search tree or graph.

The NFL theorems provide us with the arguments as to why different search strategies and enhancements are needed for different problem domains. Thus, algorithms should not be judged by obscure performance measures, that were proven not to exist, but should be qualified by the search-space properties they depend on. An interesting followup question then might be, if these properties are common among the domains we are interested in solving.

Chapter 3

Sokoban

3.1 The Game



He-Ge, Hd-Hc-Hd, Fe-Ff, Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh-Rg,
Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qi-Ri,
Fc-Fd-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Qg,
Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh-Rh,
Hd-He-Ge-Fe-Ff-Fg-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Pi-Qi,
Ch-Dh-Eh-Fh-Gh-Hh-Ih-Jh-Kh-Lh-Mh-Nh-Oh-Ph-Qh

Figure 3.1: Sokoban Problem #1 With One Solution

Sokoban is a popular one-player puzzle game. The rules and structure of the game are simple. Figure 3.1 shows a sample Sokoban problem. The playing area consists of rooms and passageways, laid out on a rectangular grid of size 20x20 or less. Littered throughout the playing area are *stones* (shown as circular discs) and *goals* (shaded squares). There is a *man* whose job it is to push each stone to a goal square. The man must push from behind the stone and can only push one stone at a time. At any time, a square can only be occupied by one of a wall, stone or man. The initial challenge is to push all of the stones onto goal squares. To increase the difficulty one can try to find more efficient solutions by reducing the required number of stone pushes and man moves.

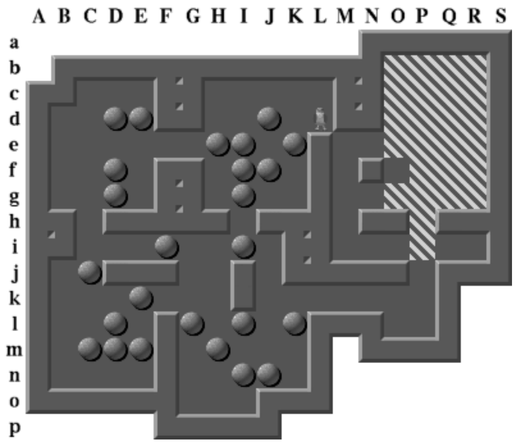


Figure 3.2: Sokoban Problem #39

To refer to squares in a Sokoban problem, we use a coordinate notation. Assuming the maximum sized 20×20 problem, the horizontal axis is labeled from "A" to "T", and the vertical axis from "a" to "t", starting in the upper left corner. In our notation we focus on stone pushes. For example, in Figure 3.1 *Fh-Eh* pushes the stone on *Fh* left one square. We use *Fh-Eh-Dh* to indicate a sequence of pushes of the same stone. A push, of course, is only legal if there is a valid path by which the man can move behind the stone and push it. Thus, although we only indicate stone pushes (such as *Fh-Eh*), implicit in this are the man's moves from its current position to the appropriate square to do the push. For example, for the move *Fh-Eh* the man would have to move from *Li* to *Gh* via the squares *Lh,Kh,Jh,Ih,* and *Hh*.

3.1.1 History and Test Suite

The game was apparently invented in the early 1980s by *Thinking Rabbit*, a computer games company in the town of Takarazuka, Japan. The game design is said to have won first prize in a computer games contest. Because of the simplicity and elegance of the rules, and the intellectually challenging complexity of the composed problems, Sokoban quickly became a popular pastime.

Several versions of the game appeared over the years, among which are PC, Macintosh and Unix versions. *XSokoban* is a popular version for Unix running X windows and can be downloaded at http://xsokoban.lcs.mit.edu/xsokoban.html. There exists a quasi-standard set of 50 problems, ordered roughly easiest to hardest in difficulty for a human to solve. According to Hiramatsu [Hir98], this set of 50 problems is derived from a PC version by Spectrum Holobyte from 1984. Similar problem configurations can be found in the problem collections "Sokoban 2" from 1984 and are now included in "Sokoban Perfect". Some of the problems have been altered slightly, probably to fit into a 19x16 format.

The test suite we are using in this thesis consists of 90 problems including the 50 standard problems plus 40 more of varying degree of difficulty. These 90 problems

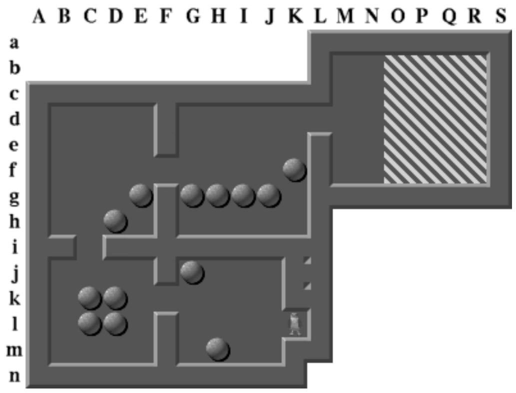


Figure 3.3: Examples of Deadlocks

were downloaded from the XSokoban web-site. Problem 1, shown in Figure 3.1, is the easiest of the set of 90. Figure 3.2 shows maze #39. The shortest recorded solution to date needs 674 stone pushes. However, the solution length is not a reliable indication for how hard a problem is to solve. One can easily think of problem configurations that require even more pushes to solve, but are conceptually simple.

An Internet high-score file is maintained that shows who has solved which problems and how efficient their solution is (http://xsokoban.lcs.mit.edu/xsokoban.html). Thus solving a problem is only part of the satisfaction; improving it is equally important.

3.1.2 Deadlock

A player new to the game will quickly discover that the constraints given by the rules of Sokoban offer some unique challenges. If a stone is pushed into a corner, it is permanently immobilized, and can never be pushed to a goal. Therefore, the problem becomes unsolvable.

Figure 3.3 shows a variety of simple stone configurations that cannot be solved. For example, the stone on *Hm* cannot reach a goal despite having legal moves, because it can never be pushed off the bottom wall. New players will soon understand that certain squares in the maze are tabu for stones. We call these squares *dead squares*. Stones that can never be pushed to a goal are *dead*, and a problem configuration containing a dead stone is said to be *deadlocked*, or simply a *deadlock*.

The two stones on *Dh* and *Eg* are also dead. Even though neither of the stones sits on a dead square, they interact in such a way that the man cannot push the stones to the goals. The four-stone group (*Ck, Cl, Dk, Dl*) in the lower left part of the maze is also a deadlock – the man can only push one stone at a time, but that is impossible in this configuration. The group of stones in the upper right corner shows a more complicated deadlock. None of the five legal moves allows the man to get "behind" the stones to push them out.

In all the examples in Figure 3.3 the deadlocks are local. In general, deadlocks can be arbitrarily complex and far reaching. Figure 3.4 shows an example of how

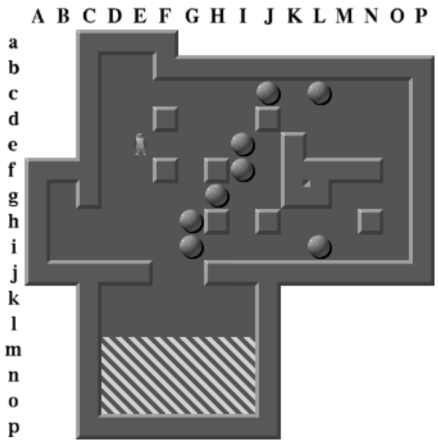


Figure 3.4: A Large Deadlock in Maze #8

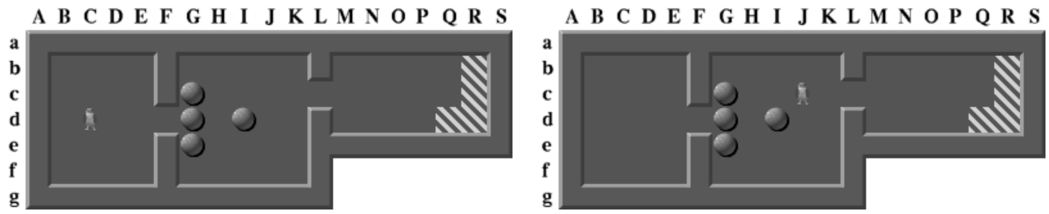


Figure 3.5: Position of the Man Matters

large and involved these deadlocks can become. They can potentially include all the stones in the maze.

3.1.3 Position of the Man

The preceding five-stone deadlock in Figure 3.3 identifies an important issue: the position of the man. Figure 3.5 shows two identical constellations of stones with the man in two different parts of the maze. The position on the left is a deadlock, whereas the maze on the right is solvable.

Furthermore, the stone on *Gd* needs a different number of moves to reach a goal, depending on the position of the man. If the man is on the right side of the stone, the stone must be pushed into the left room first before the man can reposition itself behind the stone to push it towards the goal. Therefore, the position of the man affects deadlocks as well as the number of pushes required to reach a solution.

The interactions between the stones and the man can be quite complicated, and avoiding deadlocks becomes the main challenge of the game.

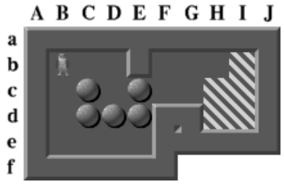


Figure 3.6: Parking

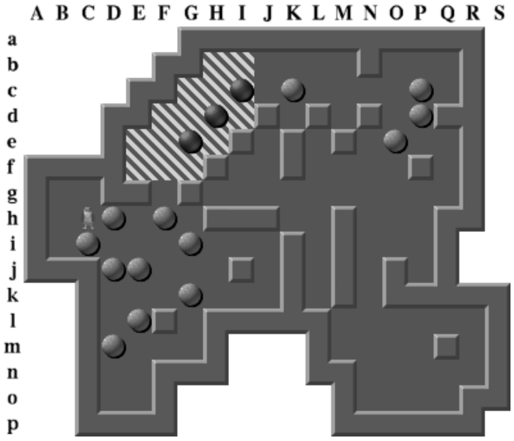


Figure 3.7: Sokoban Problem #50

3.1.4 High-Level Themes and Strategies

The beginner will soon find that there are a few general principles and high-level strategies for solving Sokoban problems. We want to briefly introduce some of them here, to facilitate later discussions.

Most of the problems appear *crowded* in the beginning. Problem #39 in Figure 3.2 is an example. To make progress, the stones have to be reorganized to simplify the maneuvering of stones into the goal area. This reorganization often requires stones to be *packed* into a small space without creating a deadlock. Packing is an important skill that a Sokoban player must acquire early on.

Often stones simply need to be moved out of the way safely until other tasks are accomplished. We call this maneuver *parking*, and it is demonstrated in Figure 3.6. Before any of the stones can be pushed to a goal square, one stone has to be *parked* at the square *Gb*. To understand this, the reader should try to think about filling in both goal squares *Ib* and *Id*. These scenarios can be arbitrarily complex. In problem #50 (see Figure 3.7), many stones have to be moved through the goal area and then parked and packed in a remote area of the maze before they can finally be pushed to the goals.

Other problems in the standard suite introduce the player to the important concept of *goal-room packing*. There are several potential problems to consider. A poorly placed stone may cause other goal squares to be inaccessible. It could also cause a

Figure 3.8: Sokoban Problem #38

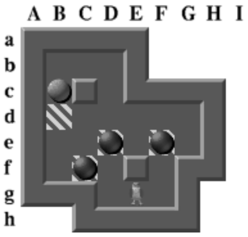


Figure 3.9: Hiroshi Yamamoto's Masterpiece

deadlock by cutting off vital paths for the man, because the goal area is needed for the man to reach certain parts of the maze. Problem #38 (see Figure 3.8) is an excellent example of these kinds of problems.

One can find several problems that live or die on *communication channels* for the man being accessible to certain regions. Inaccessibility of areas can form subtle deadlocks that require a lot of higher level reasoning by the player to be avoided.

3.1.5 Creativity, Art and Challenge

Playing our test suite, one could easily get the impression that Sokoban can be inordinately beautiful and intellectually stimulating. But it is much more than that. Sokoban is also an art. For some people it is creative work and especially in Japan it is very serious fun. The results are wonderfully elaborate and challenging designs. There are designs with only a few stones that shine with elegance and beauty because they combine simplicity and challenge. The game of Sokoban has so many "levels" that there is no end to discovery. If solving problems should ever become monotonous, there is always the possibility of creating new ones.

Figure 3.9 shows the winning design of the last Sokoban contest held in 1996. The designer Hiroshi Yamamoto succeeded in putting many of the complications of a good Sokoban puzzle into a small space.

For some humans simplicity is not a necessary element of beauty. Masato Hiramatsu created the intellectual monster shown in Figure 3.10. It is an excellent example of the level of reasoning that humans are capable of. The understanding

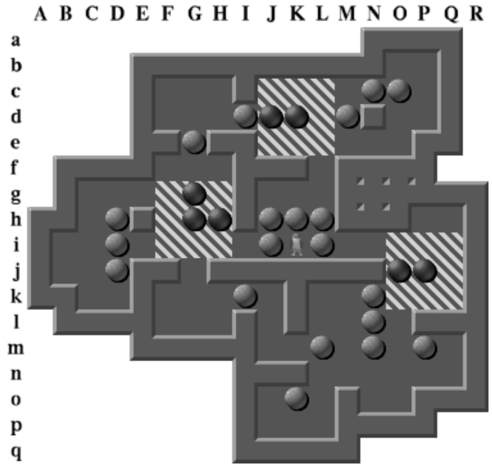


Figure 3.10: Masato Hiramatsu's Creation

Figure 3.11: Michael Reineke's Christmas Tree

of intricate details and the ability to abstract them into subproblems and in the end combine those subsolutions to solve the entire problem, taking all the interrelated special cases into account, is the hallmark of human intelligence. Creating such problems goes beyond...

The challenge in Sokoban can be combined with fun as well. An excellent example is Michael Reineke's Christmas tree shown in Figure 3.11.

The examples shown here can only skim the surface of Sokoban, only playing the game can give an indepth understanding of the beauty of the game.

3.2 Why Is Sokoban Challenging?

Many of the academic applications used to illustrate single-agent search, such as sliding-tile puzzles and Rubik's Cube, have some or all of the following properties:

- Given a solvable start state, every move preserves solvability.
- These domains also have small branching factors and moderate solution depths, resulting in moderate-sized search spaces.

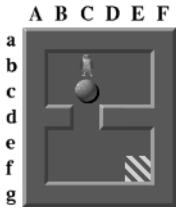


Figure 3.12: Example of Necessary Irreversible Moves

- Furthermore, simple and effective lower-bound estimators are available to guide the search.

Sokoban has none of these desirable properties, nor is a good lower-bound function known. This section examines these differences in more detail.

3.2.1 Deadlock

In most of the single-agent search problems studied in the literature, all state transitions preserve the solvability of the problem, though not necessarily the optimality of the solution. That is because all state transitions (moves) are reversible – there exists a move sequence which can undo a move. For example, a tile just pushed in a sliding-tile puzzle can be pushed back, and any rotation on a Rubik's Cube can be undone. Sokoban has irreversible moves (*e.g.* pushing a stone into a corner), and these moves can lead to states that provably have no solution. In effect, a single move can change the lower bound on the solution length to infinity. If the lower-bound function does not reflect this, then the search will spend unnecessary effort exploring a subtree that has no solution.

The presence of deadlock states in a search space creates a serious dilemma for real-time search algorithms. While we are searching, even irreversible moves are reversible via backtracking in the search space. This situation changes if we have to commit to a move in the real world before the search has found a solution, because of constraints on time or other resources. We may inadvertently move to a deadlock state – a part of the search space without solution. Since many of these deadlock scenarios cannot be determined without search, a real-time algorithm will have a difficult time allocating resources to guarantee that a solution will be found.

The simple problem in Figure 3.12 demonstrates that irreversible moves may be necessary to solve a problem. Therefore, simply avoiding irreversible moves is not feasible.

The existence of irreversible moves reveals an important property of the underlying search space: It is a *directed graph*. The traditional domains used to examine single-agent search map onto undirected graphs. This distinction leads to a rather significant difference. In a domain with an underlying undirected graph, a move of cost c can only change the distance to the goal by at most c. In domains with a directed graph search space, a legal move can decrease the distance to a goal by at most c, but can increase it by an arbitrary amount. In the extreme that is infinity, meaning deadlock.

3.2.2 Search-Space Size

The large size of the search space for Sokoban is due to potentially large branching factors and long solution lengths compared to previously studied domains. The number of stones ranges from 6 to 34 in the standard problem set. With 4 potential moves per stone, the branching factor could be over 100. The solution lengths range from 97 to over 650 stone pushes, ignoring man moves. The trees are bushier and deeper than in previously studied problems, resulting in a search space that is many orders of magnitude larger.

Note that there are different definitions of an optimal solution to a Sokoban problem: minimizing the number of stone pushes, minimizing the number of man moves, or minimizing some ratio between pushes and moves. For a few problems there is one solution that optimizes both stone pushes and man moves, but in general they conflict.

Calculating an upper bound for the search-space complexity for Sokoban reveals the startling size of the search task. For simplicity, let's restrict the size of the problem configurations to mazes of size 20 x 20. Requiring walls around the perimeter leaves an internal area of at most 18 x 18 = 324 squares where stones can move. Maximizing the possible arrangements of stones in this area requires $(18 * 18)/2 = 162$ stones. This leads to

$$\binom{324}{162} = \frac{324!}{(324-162)! * 162!} \approx 10^{96}$$

possible stone configurations. Since the man can be on any of the empty squares, we need to multiply this number by 162, resulting in a number of the order 10^{98}. When considering equivalent man positions for the task of minimizing stone pushes, the size of the search space is somewhere between 10^{96} and 10^{98}.

In these calculations we assume that there are no dead squares and that the maze is as large as possible with no other walls. In practice that is not the case. In our test suite the average number of squares is 113, of which 77 squares are not dead, and there are 16 stones on average. Table 3.1 shows the search-space size for each maze considering only the non-dead squares. This number assumes that the search will not generate moves onto dead squares, a reasonable assumption.

The median search-space size for all 90 problems using only non-dead squares is roughly 10^{18} – far less then the initial estimate of 10^{98}. However, the search-space size is not necessarily a good indicator of the difficulty of the problem, since it does not reflect the *decision complexity* [All94]. If a problem is over-constrained or under-constrained, it might be easy to solve or prove that no solution exists, respectively. The hard problems can be expected to be in the middle zone. Since the problems in the test sets are composed by humans for humans, we can assume that they are generally challenging and have a high decision complexity. The property of a sudden increase in difficulty at certain constraint levels is called a *phase transition* ([CKW91] is an excellent reference).

#	stones	squares	non-dead squares	search space sizes	#	stones	squares	non-dead squares	search space sizes
1	6	56	41	10^{8}	46	14	97	68	10^{16}
2	10	70	46	10^{11}	47	16	85	73	10^{18}
3	11	56	43	10^{11}	48	34	94	84	10^{25}
4	20	112	77	10^{20}	49	12	81	57	10^{14}
5	12	71	54	10^{13}	50	16	134	96	10^{20}
6	10	60	41	10^{11}	51	14	72	54	10^{14}
7	11	64	43	10^{11}	52	18	132	101	10^{22}
8	18	109	85	10^{20}	53	15	133	76	10^{17}
9	14	83	60	10^{15}	54	16	135	82	10^{19}
10	32	172	116	10^{31}	55	12	128	72	10^{15}
11	14	93	68	10^{16}	56	16	123	82	10^{19}
12	15	104	66	10^{16}	57	16	130	90	10^{19}
13	16	118	78	10^{18}	58	15	135	92	10^{19}
14	18	121	85	10^{20}	59	16	122	81	10^{18}
15	15	104	77	10^{17}	60	13	121	77	10^{16}
16	15	81	55	10^{15}	61	20	131	82	10^{21}
17	6	87	53	10^{9}	62	16	126	86	10^{19}
18	11	105	70	10^{14}	63	17	140	94	10^{20}
19	15	123	84	10^{18}	64	16	117	82	10^{19}
20	18	151	96	10^{21}	65	15	130	80	10^{18}
21	13	94	64	10^{15}	66	18	144	89	10^{20}
22	27	167	116	10^{28}	67	20	121	82	10^{21}
23	18	127	104	10^{22}	68	15	132	84	10^{18}
24	22	157	114	10^{25}	69	18	139	82	10^{20}
25	19	140	88	10^{21}	70	18	130	84	10^{20}
26	13	80	58	10^{14}	71	18	135	77	10^{19}
27	20	122	92	10^{22}	72	16	132	84	10^{19}
28	20	112	85	10^{21}	73	14	139	88	10^{18}
29	16	107	59	10^{16}	74	16	126	73	10^{18}
30	18	119	78	10^{19}	75	17	130	77	10^{19}
31	20	110	85	10^{21}	76	17	130	88	10^{20}
32	15	73	59	10^{15}	77	14	126	80	10^{17}
33	15	93	62	10^{16}	78	8	90	66	10^{12}
34	14	93	62	10^{15}	79	12	100	68	10^{15}
35	17	150	101	10^{21}	80	12	110	80	10^{16}
36	21	124	92	10^{22}	81	12	95	72	10^{15}
37	20	130	92	10^{22}	82	12	85	65	10^{14}
38	8	49	40	10^{9}	83	10	102	66	10^{13}
39	25	142	105	10^{26}	84	12	104	75	10^{15}
40	16	107	77	10^{18}	85	15	145	78	10^{18}
41	15	94	67	10^{16}	86	10	75	49	10^{12}
42	24	118	98	10^{25}	87	12	111	74	10^{15}
43	9	88	61	10^{12}	88	23	133	114	10^{26}
44	9	95	64	10^{12}	89	21	155	104	10^{24}
45	17	98	68	10^{17}	90	25	181	133	10^{29}
					AVG:	16	113	77	10^{18}

Table 3.1: Search-Space Sizes for the Test Suite

3.2.3 Lower Bound

In general, it is hard to admissibly estimate the number of stone pushes needed to solve a Sokoban problem.[1] The tighter the bound, the more efficient a single-agent search algorithm can be. The stones can have complex interactions with elaborate maneuvers often being required to reposition stones. For some problems, without a deep understanding of the problem and its solution, it is difficult to obtain a reasonable bound. For example, in problem #50 (see Appendix A), the solution requires moving stones *through* and *away* from the goal squares to make room for other stones. Our best lower-bound function returns 100 stone pushes (see Section 4.3), whereas the best known human solution requires 370 moves. This is clearly an enormous gap, and an imposing obstacle to an efficient IDA* search.

Several ideas come to mind when trying to design a good lower-bound function. Trivially, one could use the number of stones not on a goal; or, with a little more sophistication, one could compute the sum of the distances of each stone to its respective closest goal. Unfortunately, neither of these two heuristics is accurate.

Each goal can accept only one stone, so instead of using the goal closest to each stone, we can try to find a *matching* of stones to goals. Since we are looking for a lower bound (*i.e.* an admissible heuristic) we need to find a minimum cost matching of stones to goals, where the cost is the number of pushes required to get a stone from its current position to a specific goal.

This lower-bound heuristic is expensive to compute ($O(n^3)$), which is yet another distinction from simpler domains (for example the Manhattan distance used in the sliding-tile puzzles). Despite the expense of computing this lower bound, it is still of rather poor quality. None of the complex interactions of stones that can increase solution lengths dramatically are taken into account. The resulting differences between h and h^* can be large, causing IDA* to fail since its efficiency depends on reasonably small errors. We will discuss this lower-bound estimator and possible enhancements for Sokoban in more detail in Section 4.3.

3.2.4 Conclusions

Sokoban is a difficult search application for many reasons:

1. the branching factor is large and variable (potentially over 100),

2. the solution may be deep in the search tree (some problems require over 500 moves to solve optimally),

3. solutions are inherently sequential, subgoals are often interrelated and thus cannot be solved independently,

4. it has a complex lower-bound estimator, and

[1]We chose to solve problems minimizing stone pushes. Solving for man moves instead of stone pushes would require a different lower-bound estimator than we are currently using. In our opinion, it would be harder to find and most likely of poorer quality.

5. the search space is a directed graph that contains states with no solution.

However, humans can successfully solve Sokoban problems. They apply higher-level reasoning, pattern matching, detect exceptions and special cases, learn from previous examples, combine partial solutions, and are able to find the exact reason for why a particular strategy failed. As a testbed for artificial intelligence techniques, Sokoban offers a significant challenge to researchers, since many of the core problems of artificial intelligence need to be addressed to build a program that rivals the best human performance in solving Sokoban problems.

3.3 Related Work

Unbounded Sokoban has been shown to be NP-hard [DZ95] and P-SPACE complete [Cul97]. Dor and Zwick [DZ95] show that Sokoban is an instance of a motion planning problem, and compare the game to other motion planning problems in the literature. For example, Sokoban is similar to Wilfong's work with movable obstacles, where the man is allowed to hold on to the obstacle and move with it, as if they were one object [Wil88]. Sokoban can be compared to the problem of having a robot in a warehouse move a number of specified goods from their current location to their final destination, subject to the topology of the warehouse and any obstacles in the way. When viewed in this context, Sokoban is an excellent example of using a game as an experimental test-bed for mainstream research in artificial intelligence.

There are a number of other known Sokoban solvers in existence. It is interesting to see the different approaches people have taken.

3.3.1 Mark James

In 1993, Mark James wrote his Master's thesis at the University of Calgary on the automatic creation of macro moves [Jam93]. He used Sokoban to show the limitations of his suggested methods which worked well in other domains (see Section 2.3.4). His Sokoban program was able to solve problem #1 using over 2 hours of CPU time. No other problems where solved.

3.3.2 Andrew Myers

Andrew Myers' program appears to be an interesting approach, and it has solved nine problems. Myers [Mye97] writes that his:

> ... program uses a breadth-first A* search, with a simple heuristic to select the next state to examine. A compact transposition table stores the states. When the solver runs out of memory, it discards some states below the 10th percentile in moves made. This feature allows the program to handle levels [problems] like level 51. The solver tries to minimize both moves and pushes. It does not support macro moves.

The heuristic estimates both the number of stone pushes and the number of man movements needed to complete the puzzle. The number of pushes is estimated more quickly but less accurately, taking advantage of the usual clustering of the goal spaces in one area of the board. The estimate has two parts: the number of moves and pushes needed to push a ball to the nearest goal square, and the number of pushes needed to push a ball to each goal square from the nearest *non-goal* square. In addition, the estimator compensates for the ball that is optimal for the man to push next. The estimate is summed quickly, using approximately 700K of precomputed tables. The estimate does not consider linear conflicts, which would probably help. The heuristic is not monotonic; a conservative, monotonic estimate is used to discard suboptimal states.

Deadlocks are automatically identified for 3x3 regions, and also for certain goal locations that can never be filled. A goal location can only be filled if in one of the four directions, the two immediately adjacent squares can be made empty. If an immovable ball is placed in either square, the state is deadlocked. An optional deadlock table allows easy specification of complex deadlock conditions by hand. However, the program does not attempt to automatically fill the deadlock table.

3.3.3 Stefan Edelkamp

Stefan Edelkamp, working on his PhD at the University of Freiburg in Germany, has developed a program that can solve 13 problems of our test suite [Ede98]. His program attempts to solve for minimal number of stone pushes and uses a sophisticated decomposition algorithm to reason about the presence of static deadlocks with minimal lookahead. An elaborate data structure is used to store and match minimal deadlock patterns.

3.3.4 Meiji University

At Meiji University, Japan, A. Ueno, K. Nakayama and T. Hikita developed a strong Sokoban solver based on A*, but using non-admissible heuristics [UNH97, Hik99]. The program uses a heuristically driven deadlock search; no conflicts of potential solutions are exploited. The solutions found are neither move- nor push-optimal. The program can solve 25 of the 90 problems.

3.3.5 Sokoban Laboratory

Sokoban Laboratory is a program developed in Japan to facilitate the construction of Sokoban problems. It also contains a solver, which solves 55 problems of the test suite, using a heuristically driven best-first search. Their solutions are non-optimal, for either pushes or moves. The program is based in part on the Sokoban solver developed at Meiji University. It appears to be a team effort of several people, either contributing directly, or making code of their solvers available: K. Takahashi, A. Ueno, K. Nakayama, T. Hikita, Y. Murase, Y. Oki as well as deepgreen.

3.3.6 Deepgreen

The best overall program we have heard about so far is by an author who calls himself or herself *deepgreen*. The program can solve 62 of the 90 problems [dee99]. No details are known about this program at the moment. However, deepgreen is in communication with the authors of the other strong Japanese programs and we assume the program builds on previous efforts of the strong Japanese Sokoban community.

The collaborative approach to solving Sokoban problems make the Japanese efforts unique. Each new program can build on over 10 years of team effort. Sharing source code and ideas accumulates a wealth of knowledge that is unparalleled.

Chapter 4

Standard Single-Agent Search Methods

This chapter investigates the power and limitations of state-of-the-art single-agent search techniques. We will consider the choice of algorithm and the plethora of search enhancements available to increase search efficiency as given in artificial-intelligence text books. We implemented those techniques in the program *Rolling Stone* to obtain experimental results that allow us to evaluate them for the domain of Sokoban.

The next section contains a clarification of what the problem is that we are trying to solve. Each of the following sections is then devoted to single-agent search and its enhancements as discussed in Chapter 2. Starting with the choice of algorithm and moving on to the lower-bound function, transposition table, move ordering, deadlock tables and macro moves, this chapter discusses and explains how the standard techniques from the text books can be applied to the domain of Sokoban and, more importantly, what their strengths and limitations are.

To evaluate these methods, one central experiment is used throughout this (as well as the following) chapter(s). The program is given a fixed amount of search effort per problem – 20 million nodes. The program tries to solve each of the 90 problems within the search constraints. With each enhancement discussed and added to the program, more of the 90 problems can be solved. This "evolutionary" approach to performance evaluation has its pitfalls. Therefore, a section is included that estimates the value of each of the enhancements in a different way.

The conclusion of this chapter is that even though text books stress the importance of choosing the correct algorithm, this is usually a trivial task. In contrast, finding, implementing and tuning the right combination of search enhancements is far more difficult and important for performance. Furthermore, even though standard search enhancements can give some impressive search tree reductions, they are far from being sufficient to solve even moderately difficult problems in the domain of Sokoban.

4.1 Problem Definition

As discussed before, there are several possible ways to solve Sokoban problems. The difference lies in what one tries to optimize: man moves, stone pushes or a (weighted) combination of both. For real-world domains, the optimization would try to minimize cost. That cost is usually dependent on the actions performed: an airplane's operating cost, the time to load or unload a truck – any number of costs can be imagined. Since Sokoban is a game, real costs do not exist and we have the choice of what costs we associate with each action.

Rolling Stone is designed to find solutions that optimize the number of stone pushes; the number of man moves is not considered. Expressed in terms of cost, we associate a cost of 1 with a stone push and a cost of 0 with a man move. This choice was deliberate, because we felt that a good lower-bound estimator for man moves was harder to design and implement than for stone pushes.

When comparing human solutions to those found by *Rolling Stone*, this becomes immediately obvious: the number of man moves is usually higher. Those solutions contain sequences of pushes that can be optimized for man moves by simply rearranging the stone pushes. One could add a post-processing phase that takes a solution and tries to reorganize the stone pushes to reduce the number of man moves. This post-processor could be compared to a scheduler of tasks provided by the planner (*Rolling Stone*) minimizing the resource "man moves".

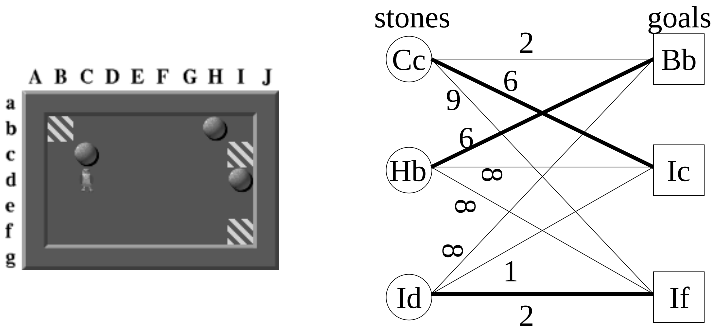
Our initial attempts to solve Sokoban optimally could solve only a few small problems. Relaxing the optimality criterion allows us to use more aggressive approaches that enable us to solve more problems. The tradeoff is between solving a few problems optimally and solving many more problems nearly optimally. We believe strongly that optimality is of little practical value if it means that only a small percentage of the posed problems can be solved. For humans, the satisfaction comes from finding *any* solution to a Sokoban problem; few are interested in or capable of finding optimal solutions.

4.2 Search Algorithm

When choosing the algorithm to solve problems from the Sokoban domain, we consider some of the crucial search-space properties:

- there are few goal nodes, and they are located deeply in the tree,
- heuristic information, in the form of a lower-bound heuristic, is available and
- the search space is large.

These properties dictate an informed search that finds sparsely distributed goals in a huge search space: IDA*. As discussed in Section 2.2.3, the choice of algorithm is rather trivial.



stones	goals on		
on	*Bb*	*Ic*	*If*
Cc	2	**6**	9
Hb	**6**	∞	∞
Id	∞	1	**2**

Figure 4.1: Minmatching Example

4.3 Lower Bound Heuristic

To design a lower-bound heuristic for Sokoban that estimates the number of stone pushes required to get all the stones to the goals, we could ignore the distance to goals and the stone-man interactions. These simplifications result in a lower-bound estimator that counts the number of stones that are not on goals. Let's call this lower-bound function *Count*. If we still ignore stone-man interactions, but take distances into account, we get a computationally inexpensive lower bound summing the distances of all the stones to their respective closest goal. Let's call this function *Closest*. Of course, these extreme simplifications are unlikely to lead to a high-quality lower-bound estimator. Allowing more of the constraints of the problem to be taken into account results in a better lower-bound estimator, albeit at a possibly higher computational cost.

4.3.1 Minimum Cost Matching

The fundamental observation leading to the lower-bound heuristic used in *Rolling Stone* is the following: Only one stone can go to any one goal. For each stone, there is a minimum number of pushes required to maneuver that stone to a particular goal. This distance (or cost) assumes no adverse interactions with other stones in the maze, basically pretending the maze is empty. The problem is to find the assignment of stones to goals that minimizes the sum of these distances.

Since there are as many stones as there are goals, and every stone has to be assigned to a goal, we are trying to find a *minimum cost* (distance) *perfect matching* on a complete bipartite graph. Edges between stones and goals are weighted by the distance between them, and assigned infinity if the stone cannot reach a goal. We will call this heuristic *Minmatching* for short.

Figure 4.1 shows an example of the lower-bound calculation. The table lists the distances from the three stones to each of the three goals in the maze. The bold entries represent a minimum cost matching. It is important to note here that the Minmatching solves one important problem. Even though the stone on *Cc* and the stone on *Id* both have goals close by, they have to be pushed to a goal further away. While the functions *Count* and *Closest* would return 3 and 5 respectively,

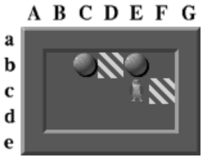


Figure 4.2: Minmatching Detects Deadlock

Minmatching returns 14! This larger lower bound allows the search to eliminate large parts of the search space.

4.3.2 Deadlock Detection

Figure 4.2 shows an example where the Minmatching algorithm detects a deadlock. None of the stones can reach the goal on *Fc*, the goal on *Db* is over-committed. This example shows how powerful this lower bound is, compared to the more naive approaches.

4.3.3 Underlying Algorithms

Minimum cost perfect matching for a bipartite graph can be solved using minimum cost augmentation [Kuh55]. Given a graph with n nodes and m edges, the cost of computing the minimal cost matching is $O(n * m * log_{(2+m/n)}n)$. Since we have a complete bipartite graph, $m = n^2/4$ and the complexity is $O(n^3 * log_{(2+n/4)}n)$. Clearly this is an expensive computation, especially if it has to be performed for every node in the search.

However, there are several optimizations that can reduce the overall cost. First, when we find a matching that reduces the minimum cost by the cost of the move, we know we can not do better and we can abort immediately. Second, during the search we only need to update the matching, because each push results in only one stone changing its distances to the goals. This requires finding a negative-cost cycle [Kle67] involving the stone pushed. Finally, we are looking for a perfect matching, which considerably reduces the number of such cycles to check.

Even with these optimizations, the cost of maintaining the lower bound dominates the execution time of our program.

4.3.4 Entrance Improvement

How can we be more efficient in computing the Minmatching? Consider Figure 4.3: Both stones need to go through the entrance square *Ce* to enter the goal area. Whenever two stones (S_1, S_2) must go through one square (let's call that square E) to get to their goals (G_1, G_2), the assignment of stones to goals does not matter, since the sum of the distances is a constant ($distance(S, G)$ denotes the distance from the square S to the square G):

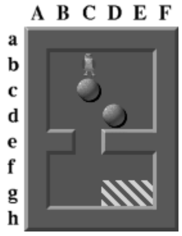


Figure 4.3: Both Stones Are Forced to Pass Through *Ce*

$$\begin{aligned}
distance(S_1, G_1) + distance(S_2, G_2) &= (distance(S_1, E) + distance(E, G_1)) \\
&\quad +(distance(S_2, E) + distance(E, G_2)) \\
&= (distance(S_1, E) + distance(E, G_2)) \\
&\quad +(distance(S_2, E) + distance(E, G_1)) \\
&= (distance(S_1, G_2) + distance(S_2, G_1))
\end{aligned}$$

Most of the problems that we are interested in solving follow similar principles. They have goal areas with single entrances. This observation can lead to significant speedups when worked into finding negative-cost cycles. However, even after this improvement, our lower-bound estimator is still more expensive to compute than most of the lower-bound functions used in the single-agent search literature (such as the Manhattan distance used for sliding-tile puzzles). Note that the entrance trick only improves the efficiency of the computation, but does not improve the quality of the lower-bound estimate.

4.3.5 Position of the Man

Simply using the distance of the stone to the goals ignores an important issue: The position of the man with respect to the stone to be pushed.

What is the distance of a stone to a particular goal? One could assume the man is able to travel from anywhere in the maze to anywhere else. However, the maze, even with only one stone in it, restricts the man's movements. If a stone's path leads through an articulation point[1] in the maze, the man's movement is restricted by that one stone.

Consider the maze in Figure 4.4. Even though the stone is only three squares away from the goal, the man is on the wrong side of the stone to be able to push the stone with three pushes to the goal. To reposition itself to the left side of the stone, the man needs to push the stone two pushes away from the goal, swing behind it, and then push it to the goal. The capability to detect this and improve the lower bound is called the *backout conflict*.

[1]Articulation points (or squares) are squares that divide the maze into at least two disjoint parts.

Figure 4.4: Distance Depends on the Position of the Man

Figure 4.5: The Stone Needs to Be Backed Out

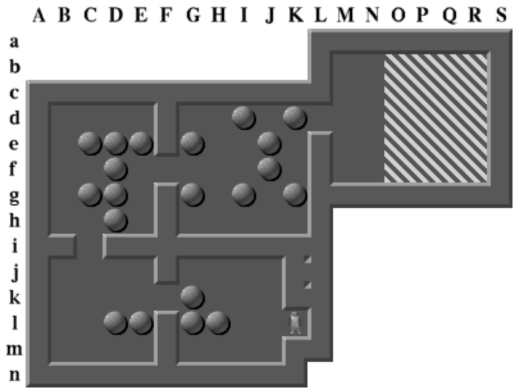


Figure 4.6: Backout Conflict Improves Lower Bound for Problem #4

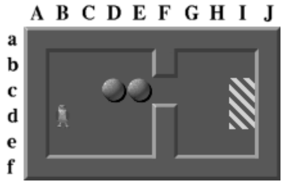


Figure 4.7: Example of Linear Conflicts

Figure 4.5 shows how this idea carries over to stones that are not on articulation squares yet, but that are forced to move through them. Problem #4 in Figure 4.6 is an excellent example of the effectiveness of this lower-bound enhancement. Four stones have to be backed out of their current room to reposition the man behind them: *Ge*, *Gg*, *Dh* and *El*. For each of these stones the lower bound is off by 6. The lower bound is increased by 24, from 331 to 355, resulting in a large reduction in search-tree size.

As already discussed in Section 2.2.2, this improvement is possible because further problem constraints are used in the lower-bound calculation. Whereas previously, the man was allowed to ignore the placement of the stones, with the backout-conflict enhancement the man cannot simply jump over stones anymore. That results in larger and more realistic distances that stones have to travel between squares and therefore in an increased lower-bound estimate.

4.3.6 Linear Conflicts

The *linear conflicts* enhancement is used to improve upon the Manhattan-distance lower bound in the sliding-tile puzzles. There, if two neighboring tiles are in each other's way (their paths directly conflict), an evasive maneuver of at least one of the tiles is necessary to allow the other to pass. This allows for an increase of the lower bound by two.

Consider Figure 4.7: The minimum matching lower-bound estimator would return a value of 10. That assumes that both stones can use an optimal path from their current location to the goal they are targeted to. But can they? No. Similar to the linear conflicts in the sliding-tile puzzles, one stone has to move off its optimal path to allow the second stone to pass or to allow the man access, depending on which stone gets pushed aside. Either the stone on *Dc* has to move down one square to allow the man to push the stone on *Ec* or, alternatively, the stone on *Ec* has to be pushed down to allow the stone on *Dc* to pass. In either case, two extra pushes are required.[2] That means the minimum cost matching is off by at least two in this case. Whenever two stones are on these two squares, we can increase the Minmatching lower bound by two without violating admissibility. We call this increase a *penalty* of two.

[2]However, if we would take man moves into account, we would have to break the linear conflict such that we minimize man moves as well. In that case, pushing *Dc-Dd* to break the conflict takes the fewest man moves with an equal number of stone pushes. Since we simplified our objective to only minimize stone pushes, we can ignore that issue in the program.

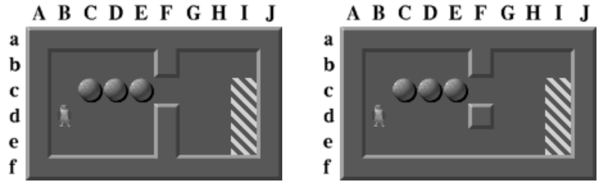


Figure 4.8: Complications With Linear Conflicts

All this is very well, but... there are two problems that we want to draw the reader's attention to that spoil the beauty[3] of the idea of linear conflicts and, unfortunately, will come up in a later chapter again.

First, consider the left maze in Figure 4.8. There are two linear conflicts of the kind we have seen in Figure 4.7. Identifying two linear conflicts does not automatically permit us to increase the lower bound by 4. By pushing the stone on square *Dc* down, only one non-optimal push is necessary to push all stones to goals. However, if the middle stone was blocked, say by a wall on square *Dd*, this maneuver would be impossible and both end stones in the chain of linear conflicts would have to be pushed. The penalty of 4 would be justified.

The second problem is shown in the right maze of Figure 4.8. Note the additional entrance. None of the linear-conflict reasoning holds anymore, because the stone moving down to allow the others to pass can now move towards its goal using the new entrance. It is therefore important to know if a stone is forced to use one direction from a square to reach *all* goals in the maze. To make matters even more complicated here, once a stone is pushed towards the lower entrance it can be the only one moving there without being penalized. Only one stone can enter through the lower entrance because there is only one goal to be filled that is reachable from it. That means we still have to break the linear conflict with the middle stone, otherwise one non-optimal push is required. But we digress...[4]

4.3.7 Dynamic Updates

The distances used for the Minmatching are precomputed before the search starts. They represent the number of stone pushes required to push a stone from any square in the maze to any other square. These distances are optimistic distances in that they assume no interference with other stones. The only restrictions are the walls in the maze.

[3]If the reader senses a little sarcasm here, she is right. It has been a recurring theme while working in the domain of Sokoban to find neat and beautiful ideas that looked so innocently promising – on the surface. After intense programming and debugging efforts (because the results where not favorable or seemed otherwise wrong) exceptions or special cases where found that had to be dealt with.

[4]If the reader feels slightly lost in all this discussion, that is understandable. Even after several years of active research we are still not able to say that we fully appreciate the depth and subtlety that Sokoban provides us with. We could not resist the temptation to introduce the reader to some of these wonderfully intricate features here!

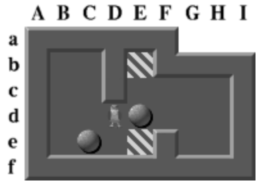


Figure 4.9: Example for Dynamic Distance Updates

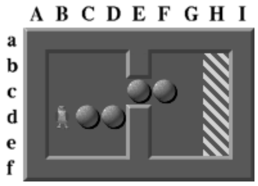


Figure 4.10: Limitations of the Lower Bound Estimator

When a stone is pushed into a corner, it becomes fixed. If that corner square is not a goal, the stone will never be able to reach a goal square and the position is a deadlock. The lower bound will detect this deadlock because of the infinite distance from the corner square to all goal squares.

However, if the corner square is a goal, the position is not necessarily a deadlock. Fixed stones on goal squares can be treated as walls. They potentially change distances, because walls are obstacles. Consider Figure 4.9. In the left maze, the stone on *Ed* can be pushed up. The distance from square *Ed* to square *Eb* is 2. However, in the right maze, since the stone on square *Ee* is fixed, the distance from *Ed* to *Eb* is no longer 2; it is 6.

Rolling Stone therefore recalculates distances whenever stones are pushed onto a fixed goal square. Note also that after a stone is fixed, other squares beside it can potentially become fixed.

4.3.8 Limitations

As the example of the linear conflicts shows, dynamic interactions of groups of stones (and possibly the man) are not reflected in the lower-bound estimator. While linear conflicts usually result in penalties of two, larger penalties resulting from other stone interactions are possible. This is dramatically illustrated with the deadlock in Figure 4.10: The position of the stones and the man create a deadlock that the lower-bound estimator cannot detect. The search could potentially explore a large tree exhaustively just to prove that there is no solution to this problem. It would do so without understanding why no solution exists.

#	MM	+BO	+LC	ALL	UB	Diff	#	MM	+BO	+LC	ALL	UB	Diff
51	118	118	118	118	118	0	34	152	152	154	154	168	14
55	118	120	118	120	120	0	71	290	290	294	294	308	14
78	134	136	134	136	136	0	40	310	310	310	310	324	14
53	186	186	186	186	186	0	35	362	362	364	364	378	14
83	190	190	194	194	194	0	36	501	501	507	507	521	14
48	200	200	200	200	200	0	41	201	219	203	221	237	16
80	219	225	225	231	231	0	45	274	282	276	284	300	16
4	331	355	331	355	355	0	19	278	282	280	286	302	16
1	95	95	95	95	97	2	22	306	306	308	308	324	16
2	119	129	119	129	131	2	20	302	444	304	446	462	16
3	128	128	132	132	134	2	18	90	106	90	106	124	18
58	189	197	189	197	199	2	21	123	127	127	131	149	18
6	104	104	106	106	110	4	13	220	220	220	220	238	18
5	135	137	137	139	143	4	31	228	228	232	232	250	18
60	148	148	148	148	152	4	64	331	367	331	367	385	18
70	329	329	329	329	333	4	25	326	364	330	368	386	18
63	425	425	427	427	431	4	90	436	442	436	442	460	18
73	433	437	433	437	441	4	49	96	104	96	104	124	20
84	147	149	147	149	155	6	42	208	208	208	208	228	20
81	167	167	167	167	173	6	61	241	241	243	243	263	20
10	494	506	496	506	512	6	28	284	284	286	286	308	22
38	73	73	73	73	81	8	68	317	319	319	321	343	22
7	80	80	80	80	88	8	39	650	650	652	652	674	22
82	131	131	135	135	143	8	46	219	223	219	223	247	24
79	164	164	166	166	174	8	67	367	375	369	377	401	24
65	181	199	185	203	211	8	23	286	424	286	424	448	24
12	206	206	206	206	214	8	32	111	111	113	113	139	26
57	215	215	217	217	225	8	16	160	160	162	162	188	26
9	215	227	217	229	237	8	85	303	303	303	303	329	26
14	231	231	231	231	239	8	89	345	349	349	353	379	26
62	235	237	235	237	245	8	24	442	516	442	518	544	26
72	284	284	288	288	296	8	15	94	94	96	96	124	28
77	360	360	360	360	368	8	33	140	150	140	150	180	30
54	177	177	177	177	187	10	26	149	163	149	163	195	32
56	191	193	191	193	203	10	11	197	201	201	207	241	34
76	192	192	194	194	204	10	75	261	261	263	263	297	34
47	197	197	199	199	209	10	29	124	122	124	122	164	42
8	220	220	220	220	230	10	74	158	172	158	172	214	42
27	351	351	353	353	363	10	37	220	242	220	242	290	48
86	122	122	122	122	134	12	88	306	334	308	336	390	54
44	167	167	167	167	179	12	52	365	365	367	367	423	56
17	121	201	121	201	213	12	30	357	357	359	359	465	106
59	218	218	218	218	230	12	66	185	185	187	187	325	138
87	221	221	221	221	233	12	69	207	217	209	219	443	224
43	132	132	132	132	146	14	50	96	96	96	100	370	270

Table 4.1: Lower Bounds

4.3.9 Results

Table 4.1 shows the effectiveness of our lower-bound estimate. The table shows the lower bound achieved by minimum cost matching (MM), inclusion of the backout enhancement (+BO), inclusion of the linear conflict enhancement (+LC), and the combination of all three features (ALL). The upper bound (UB) is obtained from the global Sokoban score file. Since this file represents the best that human players have been able to achieve, it is an upper bound on the solution length. The table is sorted according to the last column (Diff), which shows the difference between the lower bound with all the enhancements (ALL) and the upper bound (UB). Clearly, for some problems (notably problem #50) there is a huge gap. Note that the real gap might be smaller, as it is likely that some of the hard problems have been non-optimally solved by human players. Furthermore, if the difference is 0, the optimal solution lenght is known.

Using the IDA* framework and this sophisticated lower-bound function, the search cannot find even one solution to any of the 90 problems with a limit of 20 million search nodes. Even increasing our effort limit 50 fold to 1 billion nodes did not yield a single solution.

Judging the numbers of Table 4.1, one should keep in mind that the efficiency of the search depends on the overall quality of the lower-bound estimator for the entire search tree, not just the root node as shown in the table. This is one of the reasons why we cannot solve any of the problems, even though for some problems the lower bound of the root node matches the upper bound given by the human solution. Usually this would represent the ideal case in which the search should excel and easily find the solution. However, even though our lower-bound estimator seems to deliver reasonable results for the root nodes of the problems, the average error throughout the search tree is higher and leads to large and inefficiently searched trees. Examples are deadlocks that are created by the search, but not detected by the lower bound. The search, led by the poor lower-bound estimator, will explore large parts of the search space where no solution can be found.

4.4 Transposition Table

4.4.1 Implementation

Transposition tables are a standard tool to accomplish two different tasks: to avoid cycles and duplicating work by detecting nodes previously visited. Our implementation uses unique 64 bit hash keys that are used to create an index into a large hash table. The hash table used for the results reported here has 2^{18} entries. It is organized as a two-level table.[5] The replacement scheme keeps the entry searched deepest in the first level and stores the most recent entry in the second level of the table.

The hash keys incorporate only the exact stone positions. To match an entry, the keys must be identical. Since the position of the man is of importance, a second test

[5] See [Bre98] for a description and evaluation of two-level transposition tables.

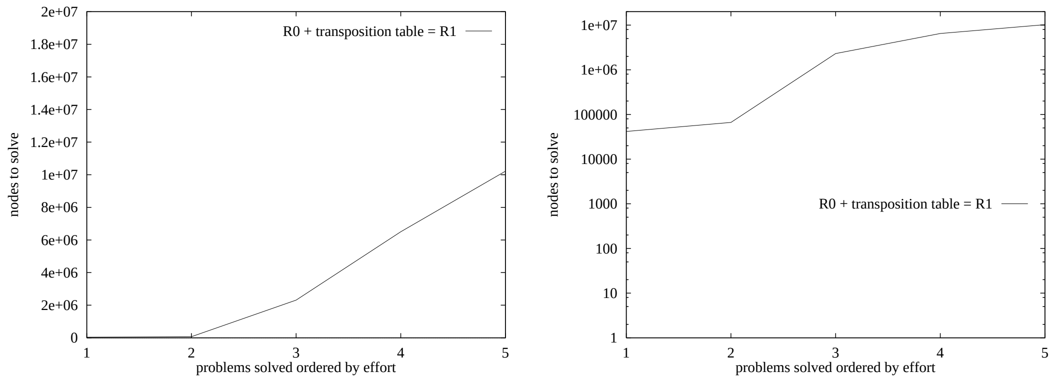


Figure 4.11: Adding Transposition Tables (Linear and Log Scale)

is performed. The man squares of both positions must be connected by a legal man path. This simplification is possible because we only optimize stone pushes. If we insisted on identical man positions, we would get fewer successful matches from the transposition table.

The value of an entry in the transposition table is composed of two things: the matching frequency and the amount of work saved when matched (the size of the tree that is cut off). The two-level strategy reflects both. An entry that was searched deeply is most likely going to save a lot of work if matched again. More recent entries have a much better chance to be matched again. By keeping both kinds of entries, the transposition table is used more effectively than using a single replacement scheme in a one-level transposition table.

4.4.2 Results

Adding transposition tables to IDA* allows the search to solve 5 problems in our test suite, when given a limit of 20 million nodes. 20 million nodes is roughly two to four hours CPU time on current fast machines. Figure 4.11 shows the effort needed to solve those problems ordered by search-tree size on a linear and a logarithmic scale. The vertical axis shows the number of nodes searched to solve the problems. The horizontal axis shows the number of problems solved. We will use this kind of graph throughout the thesis and refer to them as *effort graphs*. The keys of the effort graphs refer to different versions of *Rolling Stone*. In Figure 4.11, “R0” refers to IDA* plus Minmatching lower bound including enhancements. “R1” is a version that adds transposition tables to “R0”.

A second experiment was performed to evaluate the power of matching equivalent man positions instead of exact man positions. Using exact man positions, the number of successful hits dropped below 10% and with the same effort limit of 20 million nodes, only problem #1 could be solved. The node numbers for problem #1 increase from 41,640 to 297,498, roughly 5-fold. Problem #78 was solved with 66,309 nodes before. It cannot be solved anymore when matching exact stone positions. The number of nodes increases by at least 2 orders of magnitude.

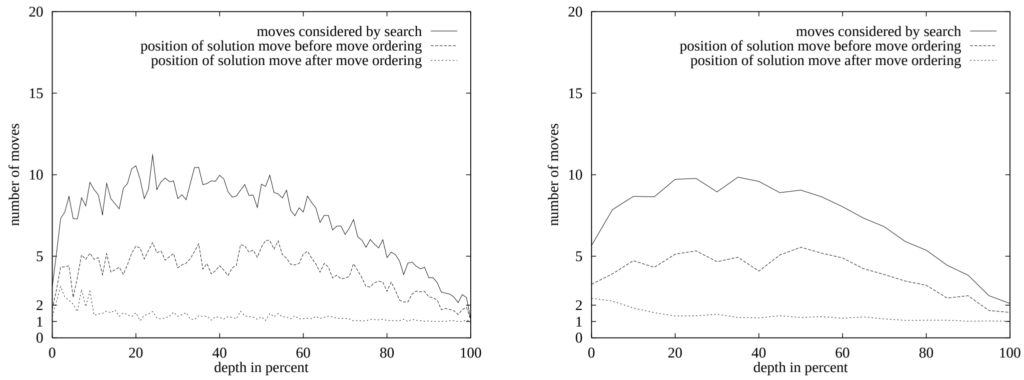


Figure 4.12: The Effect of Move Ordering (Averaged Over 1% and 5% Depth)

4.5 Move Ordering

We have experimented with ordering the moves at interior nodes of the search. One could argue that our inability to solve problem #51 is caused by bad move ordering. For this problem, we have the correct lower bound – it is just a matter of finding the right sequence of moves.

4.5.1 Implementation

We are using a move ordering scheme that we call *inertia*. Looking at the solution for problem #1 (Figure 3.1 on page 35), one can observe long runs where the same stone is repeatedly pushed. Hence, moves are ordered to preserve the *inertia* of the previous move in the following way:

1. Inertia moves are considered first.
2. Then all the moves are tried that decrease the lower bound (optimal moves), sorted by distance of the stone pushed to the goal it is targeted to, with close stones first.
3. Then all the "non-optimal" moves are tried, sorted like the optimal moves.

Since the exact distance to the goals can be arbitrary (see Section 4.3.4 "Entrance Improvement"), the actual distance used for the sorting is the distance to the entrance, except if the stone is already inside the goal area.

4.5.2 Results

Figure 4.12 shows the effect of move ordering.[6] The vertical axis shows the number of moves. The horizontal axis shows the depth of the tree in percent of the solution

[6]The data was compiled from all the positions on solution paths for all the solutions known to the best version of *Rolling Stone* used in this thesis.

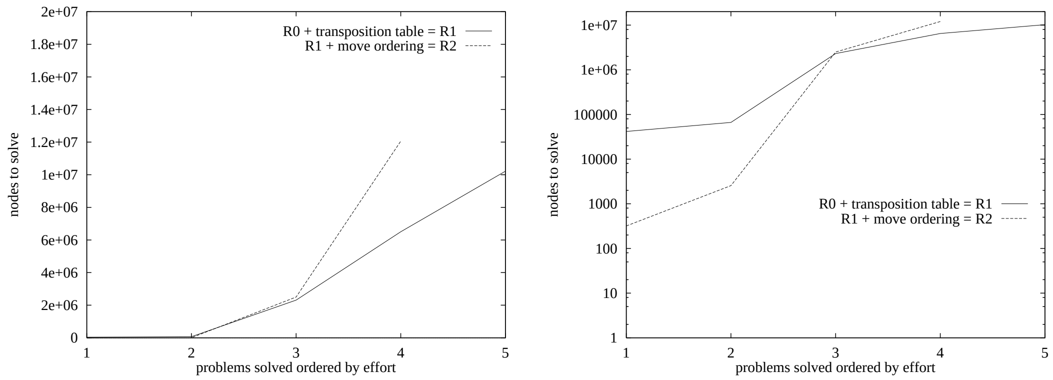


Figure 4.13: Adding Move Ordering (Linear and Log Scale)

length. The left and right graphs show the same data. The left graph clusters the data points for each 1% of tree depth, the right graph averages 5% of the data points. The upper curve indicates the average number of moves considered by the program plotted over the depth of the tree.[7] The effective branching factor is changing with the depth of the tree. In the beginning, the problem is constrained because most of the stones are still outside the goal area. As stones are being pushed to goal squares, more room becomes available for the man and other stones to maneuver, hence the increasing branching factor. Eventually, after more stones are pushed to the goal squares where they are fixed, the number of moves decreases, approaching 1.

The middle curve shows where the solution move is located in the move list after the move generation and before the move ordering. Not surprisingly, solution moves are on average in the middle of the move list. The third and lowest curve shows that after move ordering solution moves are closer to the front of the move list. The earlier the solution moves are considered, the more efficient the search is. Specifically, the last iteration will be smaller. Move ordering becomes more accurate with decreasing distance to the goal. In fact, after about 20% into the depth of the tree, the move ordering is close to perfect. In the beginning, with many complications in the maze, seemingly good moves might actually lead to deadlocks. Many of the problems in our test suite are designed in such a way that an initial "knot" has to be solved by "adding space". This can most often only be achieved with non-optimal moves. After the knot is untangled, a "mop-up phase" is entered during which stones are simply pushed to the goals. This is where our heuristic excels.

Figure 4.13 shows an additional curve in the effort graph. It shows the effect of adding move ordering to the lower bound and the transposition tables (R2). Surprisingly, one problem cannot be solved anymore and two others need more nodes to be solved. This result is not favorable for move ordering. However, we will see later that after other features are added, move ordering is a valuable contribution. Our move ordering heuristic leads to "compression". Stones close to the goals are pushed

[7]Some of the legal moves are discarded immediately because they lead to trivially provable deadlocks. These moves are not included in the graph. See Section 4.6 for more details!

closer and closer together, even though pushes away from the goals are necessary first. Compression makes deadlocks more likely. With additional enhancements that we will add later, these deadlocks become less likely and the advantages of the move ordering can work to its full potential.

4.6 Deadlock Tables

Many trivial deadlocks occur in the search. Initially, we hand-coded tests for some of the simple and common deadlock patterns into the move generation routine. This quickly proved to be of limited value, since it missed many frequently occurring patterns, and the cost of computing the deadlock test grew as each test was added. Instead, we opted for a more "brute-force" approach.

Pattern databases are successfully used to improve lower bounds in the sliding-tile puzzles [CS98, CS96] and Rubik's Cube [Kor97]. We implemented a special case of pattern databases for Sokoban. In an off-line computation, all deadlock patterns in a 5x4 square were found and stored in a database which can be queried during the search. If a move is considered for generation, the pattern of stones, walls and empty squares that is about to be created is looked up in the deadlock table. If the pattern is a deadlock, the move is not inserted into the move list.

4.6.1 Construction

An off-line search was used to enumerate all possible combinations of walls, stones and empty squares for a fixed-size region. For each combination of squares and their contents, a search was performed to determine whether or not a deadlock was present. This information was stored in the deadlock tables. The deadlock tables are implemented as decision trees. Interior nodes represent subpatterns, with links to three successors. These successors represent the parent's subpattern plus one more square's content specified as either *empty*, *wall* or *stone*. Each level in the decision tree contains different subpatterns of the same shape. The leaf nodes in the tree represent the status of a pattern: deadlock or alive. For implementation details see Appendix C.1.

For our experiments, we built two differently shaped deadlock tables for regions of roughly 5x4 squares (containing approximately 22 million entries). The two tables differ in the order the squares in the maze are queried. With two different ways to create patterns, more potential deadlocks can be found, since conflicts with goal squares can sometimes be avoided.

4.6.2 Verification and Compression

Each of our deadlock tables was verified by a separate run with a different program to ensure correctness.

Since the information in the tree is encoded in its structure and leaf node values, identical subtrees can be collapsed into one. Compression ratios of almost 5:1

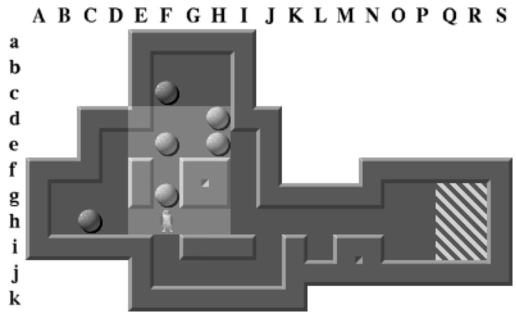


Figure 4.14: Example Coverage of the Deadlock Tables

are achieved using this subtree collapse. This type of compression does not create runtime overhead during the search, since the lookup is still on a non-compressed structure. With this more compact data structure, cache coherence may even be improved because less memory is used.

4.6.3 Usage of the Deadlock Tables

When a push *Xx*-*Yy* is considered for generation, the destination square *Yy* is used as a base square in the deadlock table and the direction of the stone push is used to rotate the region, such that it is oriented correctly. If the push *Fh*-*Fg* is made in the maze of Figure 4.14, then a deadlock table could cover the 5x4 region bounded by the squares *Hh*, *Hd*, *Ed* and *Eh*. Note that the table can be used to cover other regions as well. To maximize the usage of the tables, reflections of asymmetric patterns along the direction the stone was pushed are considered as well.

4.6.4 Limitations and Open Problems

A 5x4 region may sound like a significant portion of the 20x20 playing area. However, many deadlocks encountered in the test suite extend well beyond the area covered by our tables. Unfortunately, it is not practical to build larger tables.

Most of the effectiveness of the deadlock table is lost if a deadlock-table pattern covers a portion of the board containing a goal node. Once a stone is on a goal square, it never needs to move again. Hence, the normal conditions for deadlock do not apply.

Furthermore, for a deadlock to be in the table, all the conditions for the deadlock must be present within the region covered by the deadlock table. In the example of the push *Fh*-*Fg* in the maze of Figure 4.14 this is not given; conditions (such as connectivity and reachability for stones and man) extend beyond the area of the deadlock table. That is by far the most limiting factor of precomputed tables that are restricted to a certain area.

For the game of Go, Cazenave suggests [Caz99] using external conditions for patterns to improve their effectiveness dramatically. It remains to be investigated which

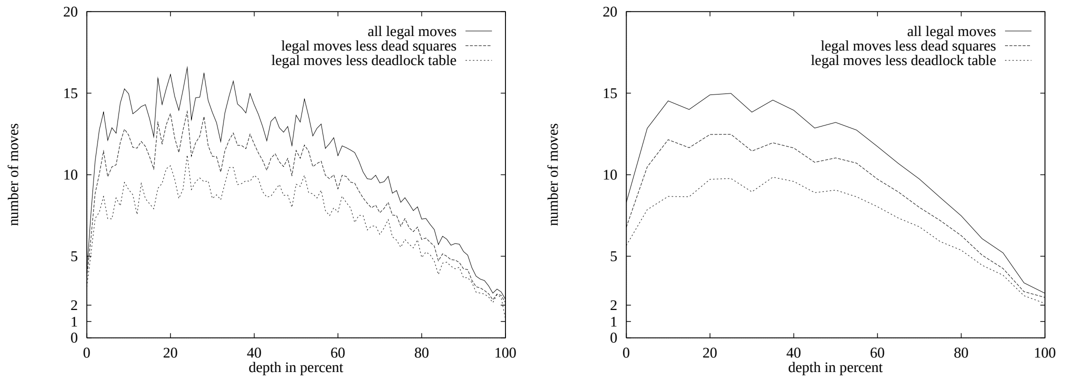


Figure 4.15: Effect of Deadlock Tables (Averaged Over 1% and 5% Depth)

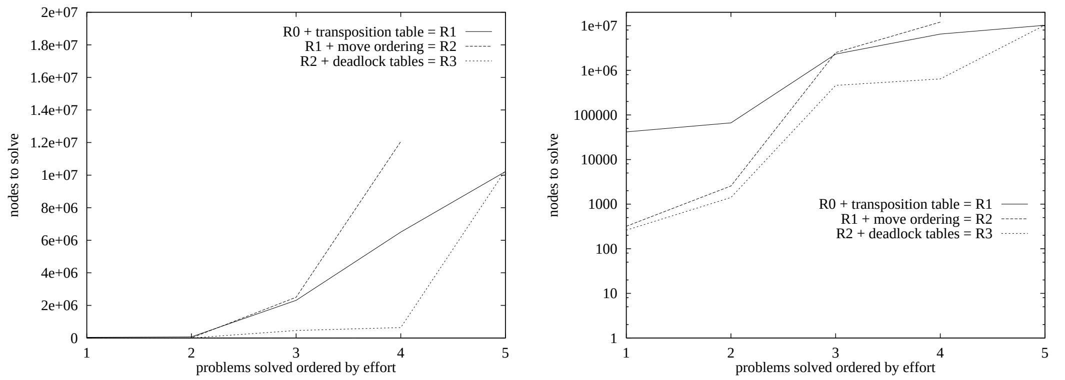


Figure 4.16: Adding Deadlock Tables (Linear and Log Scale)

conditions can express the properties of Sokoban mazes sufficiently well to generalize deadlock patterns.

4.6.5 Results

Figure 4.15 shows the number of moves in the move list over the depth of the tree. Positions on paths to solutions were chosen to avoid pathological cases. The top curve shows how many legal moves those positions have, averaged over all test positions at certain depths in the tree (1% and 5% clusters as before). The second curve shows how many legal moves exist that are not directly pushing stones onto dead squares. Note that this simple test reduces the effective branching factor by about 20%. The third curve shows how many moves are actually considered after screening moves with the deadlock tables. The savings are similar to the simple dead-square checking. On average, we can save about two moves per node that the search does not need to consider. That is equivalent to decreasing the branching factor by 2. These curves also show that the average number of moves varies considerably with the depth of the tree.

In Figure 4.16, we add another entry to the effort graph to indicate the effect of

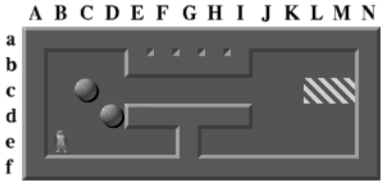


Figure 4.17: A One-Way Tunnel

adding deadlock tables to the program (R3). Now, we can solve 5 problems again, regaining the one lost with move ordering, reducing the search-tree size by orders of magnitude. It is rather illuminating to see that such an impressive reduction in the branching factor does not allow us to solve more problems.

4.7 Macro Moves

Macros are a potentially powerful tool to reduce search spaces by combining several actions into one super-action – a macro. The benefits can be dramatic. To achieve maximal savings with macro moves, they cannot simply be added to the move list. In that case, all iterations but the last would increase in node count since the branching factor is increased. Adding a macro move reduces the search tree only if at least one other atomic (non-macro) move is deleted from the move list. This way the effective branching factor is essentially the same (or less if more than one move is deleted), but the depth of the tree is reduced. Here we are discussing the specific macros used in our implementation.

4.7.1 Tunnel Macros

A tunnel is the part of a maze where the maneuverability of the man is restricted to a width of one. Figure 4.17 shows one such construct: The squares *Ec* to *Ic* are part of a tunnel. Since there can only be one stone in a tunnel without creating an immediate deadlock, tunnels cannot be used to store more than one stone.

One-Way Tunnel Macros

If a tunnel is composed of articulation squares, as in Figure 4.17, we call the tunnel a *one-way tunnel*. If a stone is pushed into a one-way tunnel, it is forced to move all the way through to the other side. There is no reason why one would delay those moves; the man cannot get to the other side of the tunnel since the stone in the tunnel cuts the man off.

When the move generator creates a move into the tunnel, in our example the push *Dc*-*Ec*, this push is substituted with the macro *Dc*-*Kc*. Note that the end square is not just *Jc*, but *Kc* – pushing the stone through and out away from the entrance of the tunnel. Of course, the push *Jc*-*Ic* is equally substituted with the macro *Jc*-*Cc*.

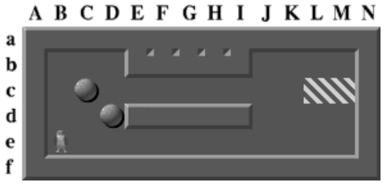


Figure 4.18: Two-Way Tunnels

Before substituting a move with a tunnel macro, we have to check if the tunnel is empty, otherwise the tunnel-macro move is illegal. If this test fails, not only is the substitution not executed, but the initial move is deleted from the move list, because it would create a deadlock and should not be considered by the search. Thus, the tunnel-macro substitution is also preventing some deadlocks.

Two-Way Tunnel Macros

One-way tunnels cannot be used as "storage" for stones. Once the stone is inside, it has to be pushed all the way. What if the man can come back from the other side and push the stone out again? That means the tunnel cannot be a one-way tunnel; the end points of the tunnel must be connected by at least one more path (the tunnel squares are not articulation points). Figure 4.18 shows two such tunnels. The following discussion uses the upper tunnel because the lower one is composed of dead squares.

The upper two-way tunnel in Figure 4.18 consists of 5 squares: *Ec,Fc,Gc,Hc,* and *Ic*. Since a two-way tunnel could be used to park a stone (pushing it in, making other moves in other areas of the maze and later coming back to push it out), we have to allow for at least one stop of the stone inside the tunnel. Since we are interested in solutions with the fewest stone pushes, parking the stone at the entrance it was pushed in is the most sensible strategy if, for example, we just want to push the stone out of the way. Therefore, the push *Dc-Ec* into the tunnel is not changed, because it is valid if we want to park the stone. However, if we want to continue pushing the stone through the tunnel, the only purpose could be to push it all the way out the other side. Thus, the push *Ec-Fc* is substituted with *Ec-Jc*. Note that this time we have to stop directly outside the tunnel, since the man could go around a different path to change the stone's direction right after it leaves the tunnel.

The substitution of moves with tunnel macros does not affect any other move that was generated. However, since another stone might be parked in the two-way tunnel already, before adding a macro, we have to verify the validity of the macro move. If it is not valid, we not only cancel the substitution, but also the move itself (it leads to deadlock) – thereby cutting down on the effective branching factor.

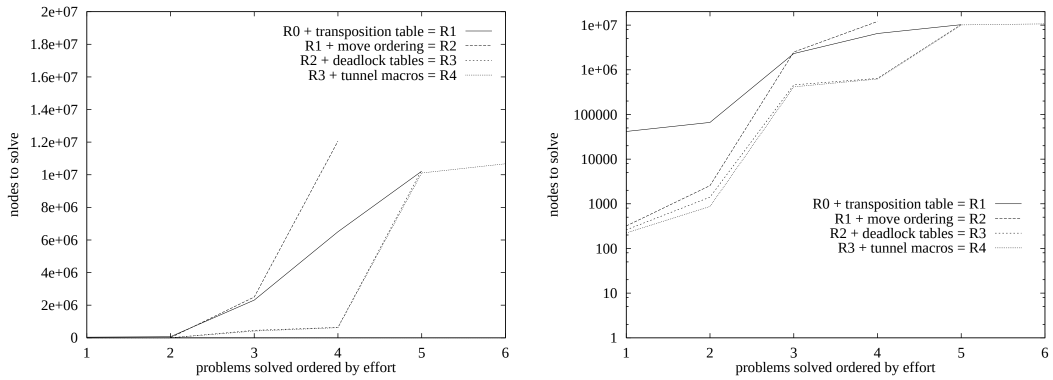


Figure 4.19: Adding Tunnel Macros (Linear and Log Scale)

Results

In Figure 4.19 the effect of tunnel macros is visible: 6 problems can now be solved (R4), one more than in the previous version. The savings for previously solved problems are not as large as for the addition of deadlock tables.

4.7.2 Goal Macros

Precomputation

Many of the Sokoban problems have the goal squares crowded together in rooms. These *goal areas* are accessible through a few squares which we call *entrances*. One can decompose the problem of solving a maze into

- how to get all the stones to the entrances, and
- how to pack them into the goal areas.

Most of the time these two parts can be solved independently, thus reducing the search space enormously. Problem #1 is a good example. As soon as a stone reaches the goal area at the right side of the maze, the stone should be pushed directly to its final destination.

We achieve this in principle by

1. defining the goal area and marking its entrances,
2. precomputing the order in which goal squares are filled without introducing deadlock in the goal area and
3. creating a structure to hold that information to be retrieved during the actual (IDA*) search.

The details of the implementation used in *Rolling Stone* can be found in Appendix C.

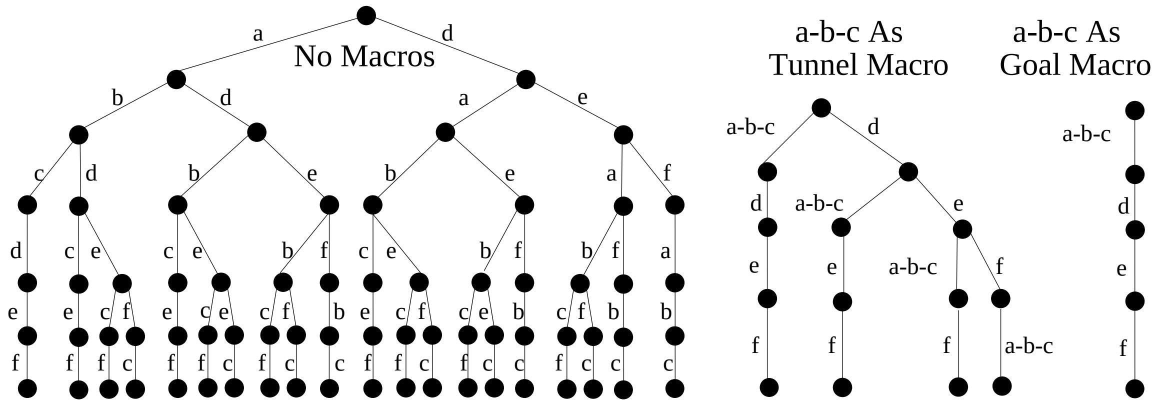
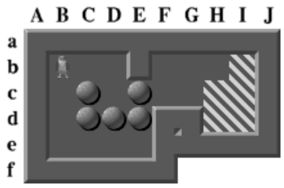


Figure 4.20: Comparing Tunnel and Goal Macro Effects



Ec-Fc-Gc-Hc-Hb-Gb, Ed-Ec-Fc-Gc-Hc-Ic-Id, Gb-Hb-Ib, Cc-Dc-Ec-Fc-Gc-Hc, Dd-Dc-Ec-Fc-Gc, Hc-Hd Gc-Hc-Ic, Cd-Cc-Dc-Ec-Fc-Gc-Hc

Figure 4.21: Parking in a Goal Area

Move Substitution

During the search, if a move is generated that pushes a stone into the entrance of a goal area, that move is substituted with the goal-macro move. Depending on the precomputation, this could be one or many goal macros. All other moves are deleted from the move list; only the goal-macro moves are considered. If we can put a stone "away", nothing else should matter at the moment. That is different from the tunnel macros, where no other move was affected.

By cutting alternative pushes, the effect of goal macros is even more dramatic than the effect of tunnel macros. Figure 4.20 shows the difference in tree-size reduction. While tunnel macros yield large savings on their own, if we can introduce a goal macro, the savings are larger.

Limitations and Open Problems

The goal macros in their current implementation have limitations. One underlying assumption is that no stone will leave the goal area once inside. Problems like #50 cannot be solved without pushing stones through the goal area. A second, even stricter assumption is that once a stone is inside the goal area, it will never move again. This does not allow for parking inside goal areas. Sometimes it is necessary to park a stone in a key position inside the goal area until later in the solution when it can finally be pushed to its final goal square. Figure 4.21 shows one such problem (assume Fc is the entrance square). Before any other stone can be pushed onto a goal square, a stone has to be parked at Gb. The stone on Gb can be pushed to its goal square only after the square Id contains a stone. These interactions violate the assumption that a stone will never move again after being pushed into the goal area and onto a goal square.

The problem of goal-macro generation in Figure 4.21 is handled correctly in our implementation. The goal-macro generation fails and goal macros are disabled. That allows the search to solve the problem, however without the benefits of goal macros.

Note that we could not solve the goal-macro generation for this problem with the current algorithms, even with a different goal area that was smaller, say only containing goal squares. In that case the first stone would have to leave the goal area after entering it, violating the assumption of a stone never leaving the goal area after it enters.

Another limitation, unrelated to the problem just described, is that a goal area containing several man entrances is often a traffic area for the man; certain parts of the maze need to stay connected to allow the man to push stones in a certain way outside the goal area. Even though we can solve the problem of packing stones inside the goal area, they might obstruct the man from other areas of the maze. Problem #38 is an example of such a case.

However, the toughest problem is when stones have to travel through the goal area to enter again later from a different entrance. Problem #50 is one such problem. Since only a limited number of stones can be entered through the lower entrance, stones have to be pushed through the goal area to the other side, parked in the lower right part of the maze until we can finally push them back into the goal area.

These open problems show that the goal macro creation is still far from being solved satisfactorily. Interactions between the goal area and the outside parts of the maze make it difficult to create good goal macros. However, their positive impact in the problems where they work is so large that any high performance Sokoban program needs to implement them in one form or another.

Results

Figure 4.22 shows the dramatic effects of goal macros (R5). Instead of solving 6 problems, we can now solve 17! The savings for individual problems are again several orders of magnitude. For example, the search nodes for problem #55 drop from over 20 million down to a mere 333 – almost 5 orders of magnitude! On average, the searches are a factor of 20 smaller with the goal macros. These are lower bounds, since unsuccessful searches are stopped at 20 million nodes.

4.7.3 Goal Cuts

We are cutting all alternative moves when we substitute goal macros. The reason being that if we can push a stone to its final destination, it will not affect other moves

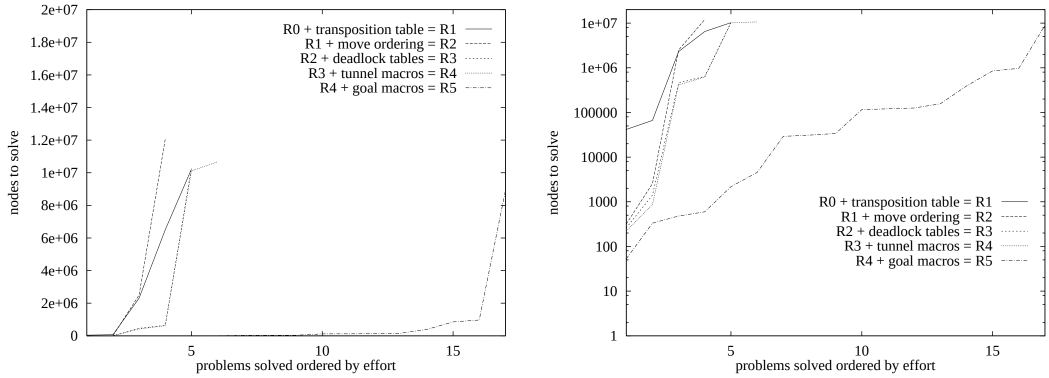


Figure 4.22: Adding Goal Macros (Linear and Log Scale)

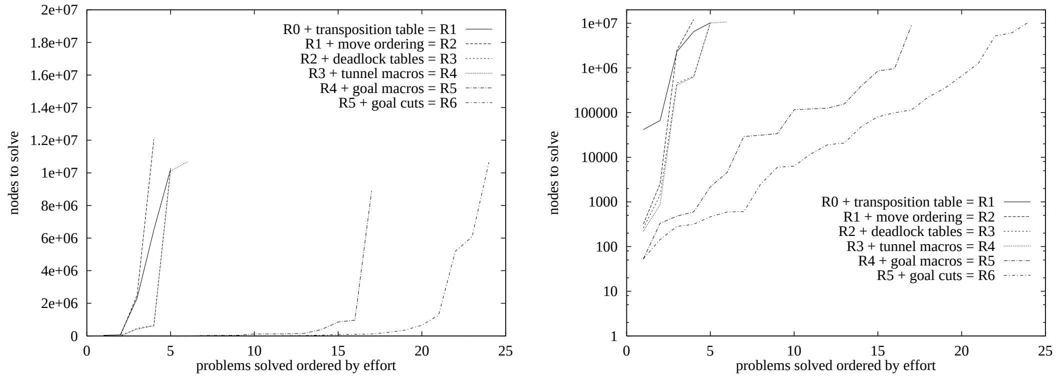


Figure 4.23: Adding Goal Cuts (Linear and Log Scale)

and we can ignore them. Could we not apply the same reasoning to the move that pushed the stone to the square from which it will be "macro"-pushed to the goal square? Goal cuts do exactly that recursively further up the tree: if a stone is pushed to a goal with a goal macro at the end without interleaving other stone pushes, all alternatives to pushing that stone are deleted from the move list.

Currently, we have implemented a scheme that will cut moves only after a stone push towards its macro move was explored. The search backs up the cut information, instead of statically trying to deduce that such a move exists in a certain position. This could potentially lead to missed opportunities for additional cuts if other moves are explored before the one that leads to the goal cut. Since the move ordering will sort moves that are close to goals towards the front of the move list, lead-off moves to goal macros are likely considered early in the move list.

Results

Figure 4.23 shows savings of around one to two orders of magnitude in search-tree size for the version using goal cuts (R6). Now, 24 problems can be solved with a search node limit of 20 million. Problem #65 was not solved without goal cuts. Now it is solved with just over 600 nodes – the search tree is over 4 orders of magnitude smaller. On average, the search trees are at least a factor of 6 smaller.

4.7.4 Correctness, Completeness and Optimality

Tunnel macros preserve correctness, completeness and optimality of the original IDA* algorithm. A solution with goal macros is still correct, but might not be optimal. For all the reasons discussed in 4.7.2, extra moves might have been necessary to find a solution. For the same reasons, completeness is not guaranteed.

4.8 Experimental Results

Table 4.2 shows the numbers for the effort graphs that where presented throughout this chapter. There are a few entries worth pointing out. Enabling all search enhancements allows problem #1 to be solved with fewer nodes than the length of the solution. Macro moves and good move ordering allow this efficient search. For example problem #4, enabling goal macros allows the search to solve it with just under 600 nodes. Previously, it was not possible to find a solution with 20 million nodes. That is an efficiency gain of at least 6 orders of magnitude.

Each search enhancement is able to potentially save orders of magnitude in search-tree size. However, some search enhancements yield overlapping savings. That means that if two features can each save 50 percent of the search tree, together they may reduce the search tree by less than 75 percent. Savings of individual search enhancements are rarely additive.

Comparing search enhancements the way we did throughout this chapter may be misleading. If a search enhancement is introduced late, when others are already present, it is harder to save on top of an already trimmed down tree. Therefore, comparing the impact of search enhancements would be unfair to the ones introduced at a later point. To exclude this effect, we ran an experiment where we turned off one feature at a time. All the other search enhancements were enabled. The results will tell us how unique the savings are that a certain search enhancement can achieve.

Figure 4.24 shows that goal macros are indeed the most valuable search enhancement that we have for Sokoban; without goal macros only 6 problems can be solved. That is a loss of 18 problems! Note that if we turn off goal macros, goal cuts are also disabled. The next most important search enhancement is the transposition table. Turning it off allows us to solve only 9 problems. With either of these two features missing alone, the search efficiency goes down dramatically and other search enhancements cannot substitute for the loss.

The version with goal cuts disabled solves 7 problems less, and the average tree size is about 6 times larger. Turning off move ordering reduces the number of problems solved to 20, losing 4. The trees grow an average of 4 times and all the problems need more nodes to solve. This shows that despite the findings in Section 4.5.2 move ordering is a valuable enhancement. These savings come only from reducing the last iteration. Surprisingly, turning tunnel macros off is not a great loss – we can still solve

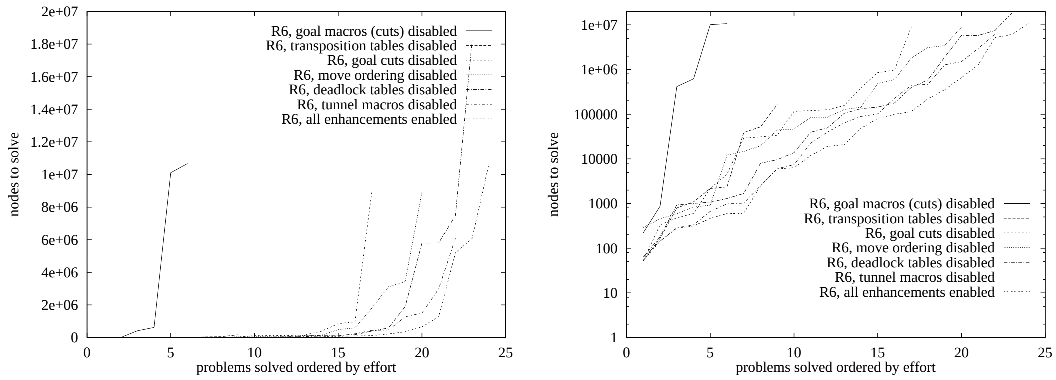


Figure 4.24: Turning One Search Enhancement Off at a Time

22 problems, 2 less than the full version. The trees are about twice the size without tunnel macros. Turning off deadlock tables loses one problem and most problems are between 2 and 50 times more expensive to solve. Table 4.3 shows all the numbers for this experiment.

4.9 Summary and Conclusions

Sokoban is a hard problem; even fixed-size Sokoban shows exponential behavior. Each additional problem becomes exponentially harder for the search to solve. To solve one or two more problems with the same amount of effort (search nodes), large portions of the search tree have to be pruned. Reducing the search tree by 50 percent is usually not enough to solve more problems; one to two orders of magnitude are needed to make significant progress.

In many search domains, an increase in search efficiency by 25% might be an interesting result. In Sokoban, even performance improvements of 50% are irrelevant. The research in single-agent search has so far focused on "simple" domains. Sokoban shows that more powerful search techniques are needed.

The basic text-book approach of IDA*, even equipped with a good lower-bound estimator, cannot even solve one problem. Using state-of-the-art techniques, such as transposition tables, move ordering and deadlock tables produces a program that can solve 5 problems of the standard 90 problem test suite. Simple tunnel macros can increase the number of solved problems to 6.

To make significant progress beyond the first 6 problems solved, the idea of macros has to be carried to its extreme. Goal macros represent the solution to the subproblem of how to arrange the stones in goal areas. The success of goal macros, the immense reduction of the search tree, can be attributed to successfully splitting the solution to a Sokoban problem into two parts: How to get the stones to the goal-area entrances and how to push them from there to their final goal square. Despite the shortcomings of the current implementation of goal macros, their impact on the program's performance is the largest of all the search enhancements introduced into our program.

#	IDA* + MM	IDA* + MM + TT	IDA* + MM + TT + MO	IDA* + MM + TT + MO + DT	IDA* + MM + TT + MO + DT + TM	IDA* + MM + TT + MO + DT + TM + GM	IDA* + MM + TT + MO + DT + TM + GM + GC
1	> 20,000,000	41,640	319	261	223	53	53
2	> 20,000,000	> 20,000,000	> 20,000,000	640,680	620,030	2,176	316
3	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	29,148	2,493
4	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	597	597
5	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	1,275,146
6	> 20,000,000	10,214,381	12,061,182	10,294,734	10,107,621	4,546	283
7	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	126,023	48,209
9	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	659,972
17	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	10,672,805	120,747	11,910
21	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	10,643,971
38	> 20,000,000	2,311,000	2,500,678	460,089	415,485	33,812	19,083
43	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	6,084,369
49	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	8,895,883	5,189,494
51	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	390,690	80,504
55	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	333	144
62	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	6,337
65	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	604
78	> 20,000,000	66,309	2,555	1,408	871	480	465
79	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	156,203	5,964
80	> 20,000,000	6,500,890	> 20,000,000	> 20,000,000	> 20,000,000	115,574	114,930
81	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	853,607	221,690
82	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	971,093	99,236
83	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	31,096	20,847
84	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	354,295
	>480,000,000	>399,134,220	>414,564,734	>391,397,172	>381,817,035	>151,732,061	24,840,912

MM – Minimum Matching, TT – Transposition Table, MO – Move Ordering,
DT – Deadlock Tables, TM – Tunnel Macros, GM – Goal Macros, GC – Goal Cuts

Table 4.2: Effort Graph Data

Table 4.3: Turning One Feature Off at a Time

#	GM (GC) Off	TT Off	GC Off	MO Off	TM Off	DT Off	All On
1	223	53	53	291	63	63	53
17	10,672,805	> 20,000,000	120,747	85,367	2,979,182	13,796	11,910
2	620,030	1,093	2,176	833	337	1,076	316
21	> 20,000,000	> 20,000,000	> 20,000,000	8,927,624	> 20,000,000	> 20,000,000	10,643,971
3	> 20,000,000	165,274	29,148	12,050	2,493	7,992	2,493
38	415,485	> 20,000,000	33,812	19,582	22,559	49,657	19,083
4	> 20,000,000	2,369	597	1,791,400	1,025	1,686	597
43	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	6,084,369	18,216,241	6,084,369
49	> 20,000,000	> 20,000,000	8,895,883	> 20,000,000	> 20,000,000	7,482,856	5,189,494
5	> 20,000,000	> 20,000,000	> 20,000,000	3,112,231	1,275,146	1,901,783	1,275,146
51	> 20,000,000	> 20,000,000	390,690	46,493	89,038	144,393	80,504
55	> 20,000,000	166	333	920	144	180	144
6	10,107,621	799	4,546	455	286	1,296	283
62	> 20,000,000	52,076	> 20,000,000	595,161	7,391	104,691	6,337
65	> 20,000,000	2,179	> 20,000,000	> 20,000,000	667	932	604
7	> 20,000,000	> 20,000,000	126,023	44,486	63,857	133,389	48,209
78	871	38,951	480	594	984	1,011	465
79	> 20,000,000	> 20,000,000	156,203	142,828	6,061	9,630	5,964
80	> 20,000,000	> 20,000,000	115,574	85,633	228,536	178,001	114,930
81	> 20,000,000	> 20,000,000	853,607	127,657	443,344	588,833	221,690
82	> 20,000,000	> 20,000,000	971,093	487,011	101,712	5,797,306	99,236
83	> 20,000,000	> 20,000,000	31,096	14,694	39,993	39,666	20,847
84	> 20,000,000	> 20,000,000	> 20,000,000	> 20,000,000	466,263	397,257	354,295
9	> 20,000,000	> 20,000,000	> 20,000,000	3,431,973	1,503,297	5,797,980	659,972
	>381,817,035	>300,262,960	>151,732,061	> 98,927,283	> 53,316,747	> 60,869,715	24,840,912

MM – Minimum Matching, TT – Transposition Table, MO – Move Ordering,
DT – Deadlock Tables, TM – Tunnel Macros, GM – Goal Macros, GC – Goal Cuts

They allow 11 more problems to be solved, for a total of 17. Goal cuts extend the idea of the goal macro and can push the number of solved problems to 24.

Even though the reductions in search-tree sizes are impressive and result in an increase in the number of problems solved from 0 to 24, we should not forget that the best version of the program can still only solve 24 of the 90 problems. Even though this set is challenging for humans, many problems not yet solved should be well within reach of a computer program.

We have seen that rather impressive search-tree size reductions result in small increases in the number of problems solved. If we want to increase the number of problems solved significantly, we will have to trim the search trees radically.

We can identify two main inefficiencies in the program:

- The lower bound does not capture dynamic interactions of stones that block each other and/or the man. If we could find a way to capture this information and be able to improve the lower bound with it, the search should improve dramatically.
- The mazes are large and often contain parts that are virtually non-interacting. However, the search will consider moves in any of those separate parts in any order. Had the program knowledge about which moves are not influencing the currently attempted subgoal, legal, but irrelevant, moves could be ignored. This could lead to a reduction in the branching factor that can potentially remove large portions of the search tree.

We will discuss methods that address these points in the next two chapters.

Chapter 5

Pattern Search

5.1 Introduction

In the previous chapter, we concluded that the standard techniques are insufficient to make further progress in the domain of Sokoban. Additional search enhancements are needed to enable us to solve significantly more problems from the test set. Since large portions of the search are wasted searching problem configurations with deadlocks present, we speculated that the detection of these deadlocks could lead to significant efficiency gains. The techniques suggested in this chapter are a direct attempt to rectify this problem.

In this chapter, we introduce a new search enhancement that dynamically finds deadlocks and improves lower bounds. *Pattern search* is a real-time learning algorithm that identifies the minimal conditions necessary for a deadlock, and applies that knowledge to eliminate provably irrelevant parts of the search tree. By speculatively devoting a portion of the search effort to learning properties about the search space, the program trades off search-tree size *versus* acquired knowledge.

In the game of Sokoban, the additional knowledge gained by the pattern searches improves the program's search efficiency. The average growth rate of the tree is roughly a factor of 600 times smaller per IDA* iteration. This results in 48 solved Sokoban problems, and significant progress towards solving many more.

We start by introducing the general concepts by looking at deadlock detection and later in this chapter show how to generalize these concepts and methods to the more general case of lower-bound improvements.

5.2 Basic Idea

After making a move, establishing the presence of a deadlock can be quite involved. The deadlock may consist of as few as one or as many as all the stones in the maze. We will describe how to prove the presence of deadlock by showing that the conditions needed to prevent deadlock are not present.

In general, proving that a subset of stones in a maze (a *pattern*) of stones creates a deadlock requires a search to verify that no possible solution path exists. A *pattern*

```
PatternSearch( From, To ) {
  clear TestMaze;
  StonePath = {To};
  FOR( i = 1; i <= MAX_PATTERN_SIZE AND NOT EffortLimit(); i ++ ) {
    IF( stone s on a square in StonePath )
      add closest s to TestMaze
    ELSE IF( stone s on a square in ManPath )
      add closest s to TestMaze
    ELSE BREAK;
    solution = PIDA*( TestMaze, SolLength, ManPath, StonePath );
    /* Test for a deadlock */
    IF( solution == NO AND NOT EffortLimit() ) {
      GeneralizeAndAddPattern( TestMaze, infinity );
      BREAK;
    }
    /* Test for a lower bound increase */
    IF( solution == YES ) {
      lb = LowerBound( TestMaze );
      IF( SolLength > lb )
        GeneralizeAndAddPattern( TestMaze, SolLength - lb );
    }
  }
}
```

Figure 5.1: Pseudo Code for Pattern Searches

search consists of repeated IDA* searches of patterns with more and more stones. A pattern search may result in the discovery of a deadlock pattern which can be used throughout the search to assign the correct lower bound of infinity to any state containing that deadlock. For maximal reusability it is of interest to find the minimal pattern of stones that causes the deadlock.

Detecting deadlocks is only a special case of a more general problem. Stones are interacting in such a way that the total number of pushes required to get them to goals is more than the lower-bound function estimates. Whereas deadlocks are corrections of the lower bound to infinity, the general case is smaller increases of the lower bound, so called *penalties*.

5.3 Basic Algorithm

In the following, we will refer to two different mazes:

- the *original maze*, which is the maze with all the stones of the current IDA* position, and
- the *test maze* which will be used for the pattern searches.

The pattern search will perform small searches in the test maze with a subset of stones from the original maze to determine if the last move introduced a deadlock. In principal, the algorithm performs the following 4 steps:

1. Start by putting only the last stone moved into the test maze.
2. Try to solve the problem.
3. If no solution is found a deadlock is detected, exit.
4. If a solution is found add a stone that is on a square that is needed for the solution.
5. Goto 2.

More specifically, a pattern search iterates on the number of stones in the test maze. If we make a move A-B, we might introduce a deadlock. If this deadlock was not present before the move, then the moved stone, now on square B, must be part of the deadlock pattern. This is the initial stone included in the test maze and PIDA*[1] is called to solve it. PIDA* either returns with failure (no solution, hence deadlock), or it finds a solution. In the latter case, we are interested in the set of squares that are used by the stone and the man during the solution. We call these sets of squares the *StonePath* and the *ManPath*, respectively. These sets of squares are preconditions for the solution to work. The ManPath and the StonePath are used to determine which stone from the original maze to include next in the test maze. Stones in the original maze that are on one of the squares in ManPath or StonePath conflict with the test-maze solution. The stone in StonePath closest to square B (the square the stone was moved to in the original maze) is added next to the test maze. If such a stone does not exist, the stone that is on ManPath closest[2] to square A is used. If no such square exists, the pattern search returns without finding a deadlock.

After including the next stone, PIDA* is called again. It returns with a solution and the two conflict sets. If no deadlock was found, then the conflict sets are used again to add another stone to the test maze. The pattern search terminates in either of three cases:

- the effort limit is reached (usually a predetermined number of nodes),
- a deadlock was detected (all frontier nodes have a heuristic value of infinity or have no moves), or
- no more stones conflict with the solution found.

See Figure 5.1 for the pseudo code describing the pattern search.

[1]PIDA* is a special version of IDA*. See Appendix C.3 for details.

[2]*Closest* is always with respect to the distance of either the stone or the man to the conflicting stone. These distance measures are possibly different due to the more restricted movements of the stones.

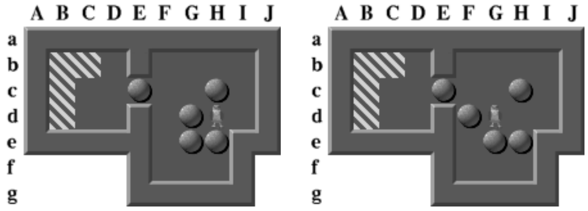


Figure 5.2: Deadlock Example

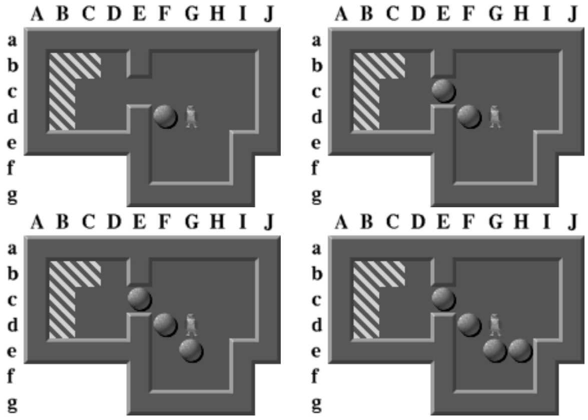


Figure 5.3: Sequence of Test Mazes as Passed to PIDA* (a, b, c, and d)

5.4 Example

Figure 5.2 shows a simple position, before and after the move *Gd-Fd*. The question is whether or not this move introduces a deadlock. Figure 5.3 shows how the test maze is built. Since the last move ended up on square *Fd*, the test maze is initialized with a single stone on *Fd* (Figure 5.3a). A PIDA* search finds a 5-push solution, and returns a ManPath (*Gd-Ge-Fe-Fd-Gd-Gc-Fc-Ec-Dc-Cc*) and a StonePath (*Fd-Fc-Ec-Dc-Cc-Bc*). Since a solution was found, we continue the pattern search.

The original maze has a stone on one of the squares (*Ec*) that the stone moved over. Now this stone is included in the test maze (Figure 5.3b). PIDA* will solve the test maze with the two stones and again return a ManPath (*Gd-Gc-Fc-Ec-Dc-Dd-Cd-Cc-Dc-Ec-Fc-Gc-Gd-Ge-Fe-Fd-Gd-Gc-Fc-Ec-Dc-Cc*) and a StonePath (*Ec-Dc-Cc-Cb Fd-Fc-Ec-Dc-Cc-Bc*). This time, there are no stones in conflict with the StonePath. However, there is a conflict with the ManPath on square *Ge*. Therefore, the stone on *Ge* is added to the test maze (Figure 5.3c) and another search is started. A solution will be found, requiring a fourth stone to be added (Figure 5.3d).

The fourth call to PIDA* will return no solution and announce a deadlock with this pattern of four stones. Identifying the critical stones has been driven by whether or not they conflict with a potential solution. The irrelevant parts of the maze (such as the stone on *Hc*) are ignored.

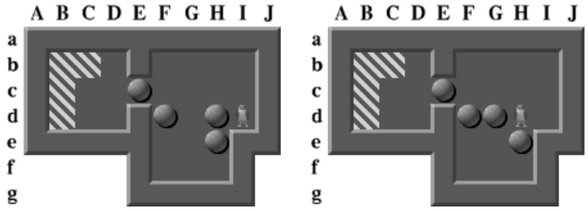


Figure 5.4: Penalty Example

5.5 Minimizing Patterns

The fewer stones in a deadlock pattern, the more likely it will match an arbitrary position and be used to eliminate futile branches of the search. A *minimal deadlock pattern* is a deadlock pattern from which no stone can be removed without making the remaining pattern solvable. The attentive reader will have noticed that only three stones are needed to guarantee the deadlock in Figure 5.3; the stone on *Ec* is unnecessary. Before saving the deadlock pattern, our program will attempt to minimize the number of stones in it.

The deadlock minimization routine takes an *N*-stone pattern and considers each of the possible *(N-1)*-stone subpatterns. Each of the *(N-1)*-stone subpatterns is searched to verify whether removing the other stone preserves the deadlock. If the deadlock still exists, the removed stone was not part of a minimal deadlock set and is removed from the deadlock pattern.

In general, there might be several different minimal deadlock sets. We experimented with different ways of minimizing deadlock sets, but concluded that the greedy and straightforward removal of stones is the most cost-effective way. Often the cost of minimization is greater than the cost of finding the deadlock pattern itself.

5.6 Deadlocks and Penalties

The presence of a deadlock pattern in a position means that the lower bound increases to infinity. Can we find patterns that allow us to increase the lower bound by an amount less than infinity?

Assume there are three stones in the test maze and PIDA* starts its first iteration but fails to find a solution. Hence PIDA* proved that this pattern cannot be solved with the number of moves that the heuristic lower bound indicated. In other words, the lower bound is wrong.

A pattern search will fail to find a deadlock after the push *Hd-Gd* in Figure 5.4. However, this pattern search will discover that it requires 2 iterations (4 moves) more to solve this problem. Hence the lower bound is off by 4. The pattern just discovered can be minimized and used throughout the IDA* search to improve the lower-bound calculations.

5.7 Specializing Pattern Searches

Our program *Rolling Stone* uses three specialized pattern searches. Specialization is a means of improving the efficiency of pattern searches, even though they might miss a few patterns. By decreasing the cost of the individual pattern search, more pattern searches can be executed. All three pattern searches are designed to find different types of deadlocks and/or penalties. Whereas deadlock searches are geared towards finding deadlocks involving many stones, penalty searches are designed to find penalties with fewer stones. Area searches are aimed at cheaply finding deadlocks caused by inaccessible areas. We believe that our attempts at specialization are only a start. Further progress is certainly possible. For more details see Appendix C.3.

Deadlock Search: The deadlock search follows the generic outline of a pattern search as described above. However, a deadlock PIDA* search is allowed to take a few shortcuts. For instance, the definition of a goal node is more liberal. A position where the man can reach all squares in the maze (the stones do not block parts of the maze) is considered unlikely to contain a deadlock. These shortcuts reduce the cost of the deadlock searches and allow them to include more stones. However, deadlock searches are less likely to find increases in lower bounds.

Penalty Search: After a deadlock search fails to produce a cutoff (either by proving deadlock or finding a large enough penalty), a penalty search is executed. Penalty searches are not allowed to take shortcuts. Therefore, they have a chance to find penalty patterns that the deadlock search missed. Penalty searches are more expensive (no shortcuts) and usually include less stones with the same effort limit.

Area Search: If even the penalty search fails to discover a large enough penalty to cause a cutoff, a third and final pattern search is executed. Instead of using the solution conflicts to find the next stone to include, area searches use heuristics to determine the stone(s) most likely to be involved in a penalty pattern. They try to prove that an area inaccessible to the man and adjacent to the last stone moved is enclosed by a deadlock pattern. To that end, prior to calling the PIDA* search, all the stones are included that are surrounding the area that is inaccessible to the man. The area PIDA* searches are as liberal as the deadlock PIDA* searches. If the search cannot find a large enough penalty to cause a cutoff, more stones are included that surround other inaccessible areas, this time not directly beside the man.

5.8 Parameters and Control Function

Pattern searches can be costly. There are three main factors involved in their cost: the frequency of the pattern searches, the bound on the size of a pattern search (the effort limit), and the bound on the deadlock-pattern size (number of stones allowed).

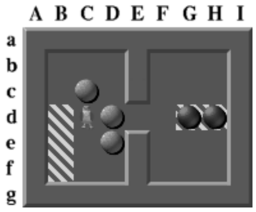


Figure 5.5: Example for Control Function

Frequency of Pattern Searches: We cannot afford to do a pattern search at every node in the IDA* search. We use some non-trivial heuristics (the control function) to decide when to invest in a pattern search. A pattern search is executed if any of the three front and two side squares of the stone pushed contains either a stone or area the man cannot reach. The directions are with respect to the orientation of the last move. Otherwise it is unlikely we have introduced a penalty or deadlock.

Figure 5.5 shows an example. Assume the move *Cd-Cc* was the last, then none of the squares ahead (*Bb, Cb*, and *Db*), nor the side squares are occupied by stones or are inaccessible to the man. The move *Cd-Cc* is unlikely to have introduced a deadlock. Now, assume *Cd-Dd* was the last move (instead of *Cd-Cc*). *Ed*, the square ahead of the stone moved to square *Dd*, is not accessible to the man. Furthermore, there is a stone on *De*, just to the side of the stone moved onto square *Dd*. Either of these two conditions is sufficient to trigger a pattern search.

The transposition table stores whether or not a pattern search was performed to avoid multiple pattern searches at the same node.

Size of the Pattern Search: Pattern searches are restricted to a maximum effort of 1000 nodes. If this limit is reached, the search is aborted. However, whenever a pattern search is successful in finding a penalty, it is allowed to continue searching for another 1000 nodes.

1000 nodes per pattern search seemed to achieve most of the benefits for a still reasonable overhead. Increasing the effort limit did decrease the number of IDA* nodes, but the additional overhead outweighs the benefits. Smaller pattern searches cannot find large enough penalties.[3]

Pattern Size: Pattern searches are stopped once they have included all but two[4] stones from the original maze. This is an artificial limitation, but we have

[3]However, the number 1000 is still chosen quite arbitrarily. We will see this kind of "magic" number many times. They are the result of educated guesses and sometimes tuned by experiments. However, the tuning depends on many other variable search parameters and the test suite used. Truly optimizing these "magic" numbers is at least computationally prohibitively expensive, if not impossible. We will get back to the issue of tuning in a little more depth in Section 8.6.

[4]This is another one of these magic numbers.

not fully explored the tradeoffs of finding larger deadlock patterns *versus* the effort required to find them. Because large patterns are also less likely to match again later in the search, the benefits of large patterns are small. Furthermore, the searches become exponentially more expensive the more stones are present. Therefore, it seemed prudent to limit the pattern size.

Controlling these three parameters is vital for the success of the pattern searches. Too many and too expensive pattern searches can quickly create a large overhead, easily offsetting the savings achieved with the pattern knowledge.

5.9 Storage and Matching

To incorporate the deadlock and penalty patterns into the regular IDA* search, we need to save the patterns found and use them to match positions in the search. The pattern matching is complicated by the fact that one needs to match not only the stones, but also the man position. With each pattern of stones, the squares which the man in the test maze cannot reach (non-reachable squares) are stored. To match a pattern, the current position must have stones in the same places as the pattern and the man must not be on any one of the non-reachable squares stored with the pattern.

As seen with the linear conflicts, patterns can overlap. To ensure admissibility, each stone can only be used once for a pattern that is included in the total penalty. Therefore, we have to optimize which of the overlapping patterns to include to maximize the total penalty. First, all penalty patterns are collected that are overlapping. Then, for each of these sets of conflicting patterns a search is used to find the subset of patterns that maximizes the penalty that can be achieved.

Consider Figure 5.6. Four penalty patterns are matched for the position shown at the top. The penalties for the patterns from left to right (a to d) are: 2, 2, 8, and 4. What is the maximal penalty that can admissibly be given to that position? Since all patterns overlap with pattern c, none can be included if pattern c is. However, the patterns a, b, and d could not all be included at the same time either, because the patterns a and b overlap as well. Including pattern d and either one of a or b could only lead to a penalty of 6. Therefore, only using pattern c results in the largest admissible penalty: 8.

The penalties of non-overlapping patterns are simply added. We are using a second improvement in *Rolling Stone* to speed up the pattern matching which we call *lazy maximization*. The search passes a parameter to the matching algorithm that indicates the minimum (or target) penalty needed to cause a cutoff. When the matching algorithm has produced at least the target penalty, it can prematurely return, thereby saving further matching effort.

Figure 5.7 shows maze #30 with a stone configuration that arises during the search. Two penalty patterns are successfully matched, resulting in a lower-bound of 38 (14+24).

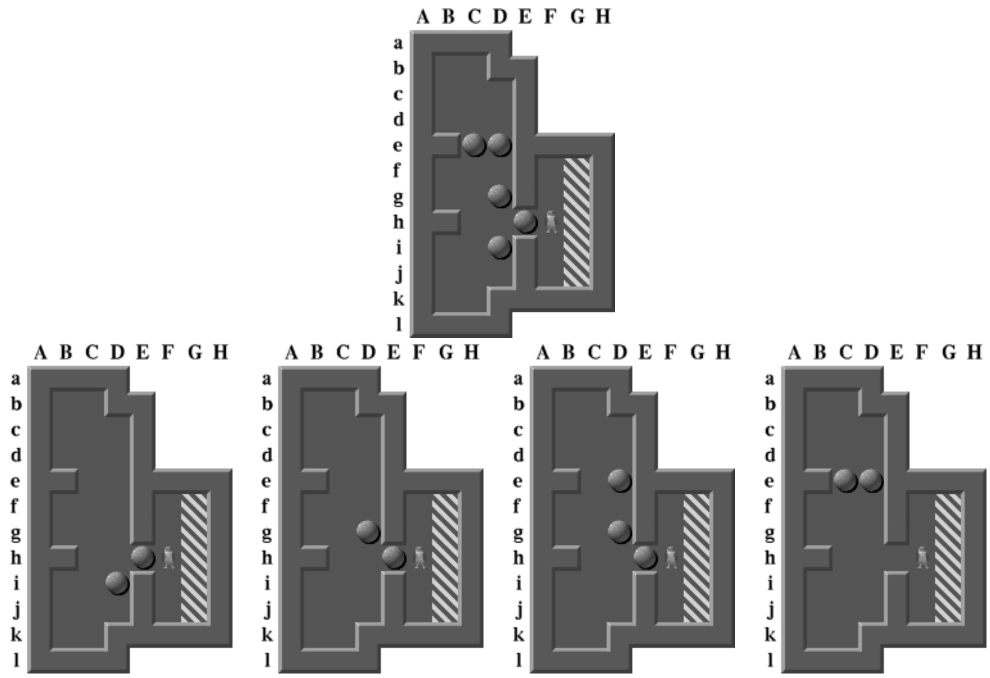


Figure 5.6: Maximizing the Total Penalty (a to d)

5.10 Cutoffs and Back-Jumping

Matching a deadlock pattern always causes a cutoff. Matching a penalty pattern may allow an increase in the lower bound. A cutoff happens only when the matched penalty patterns increase the lower bound sufficiently (above the current threshold).

However, these direct cutoffs are only part of the benefits of the patterns. The pattern searches might uncover a pattern that wasn't created by the last move. In that case, when the last move is unmade, the pattern is still present. In fact, the lower bound of a state can change during the search of a subtree. Therefore, when the search returns from a recursive call, the search has to check if the lower bound is now sufficient to cause a cutoff. In that respect, such a pattern leads to a kind of dependency-directed backtracking, as known from constraint-satisfaction problems [SS77, Gas79]. As long as the pattern exists in the maze, the search continues to backup. When the move that created that pattern is unmade, the associated penalty disappears and the search proceeds normally.

5.11 Scan Search

The accuracy of the evaluation at the root node determines how many iterations are needed to find a solution. The larger the gap between the lower bound of the root node and the correct solution length, the more iterations have to be searched.

Rolling Stone runs one penalty search for each of the stones in the initial position to find preexisting penalty patterns – the *scan search*. Finding such preexisting patterns

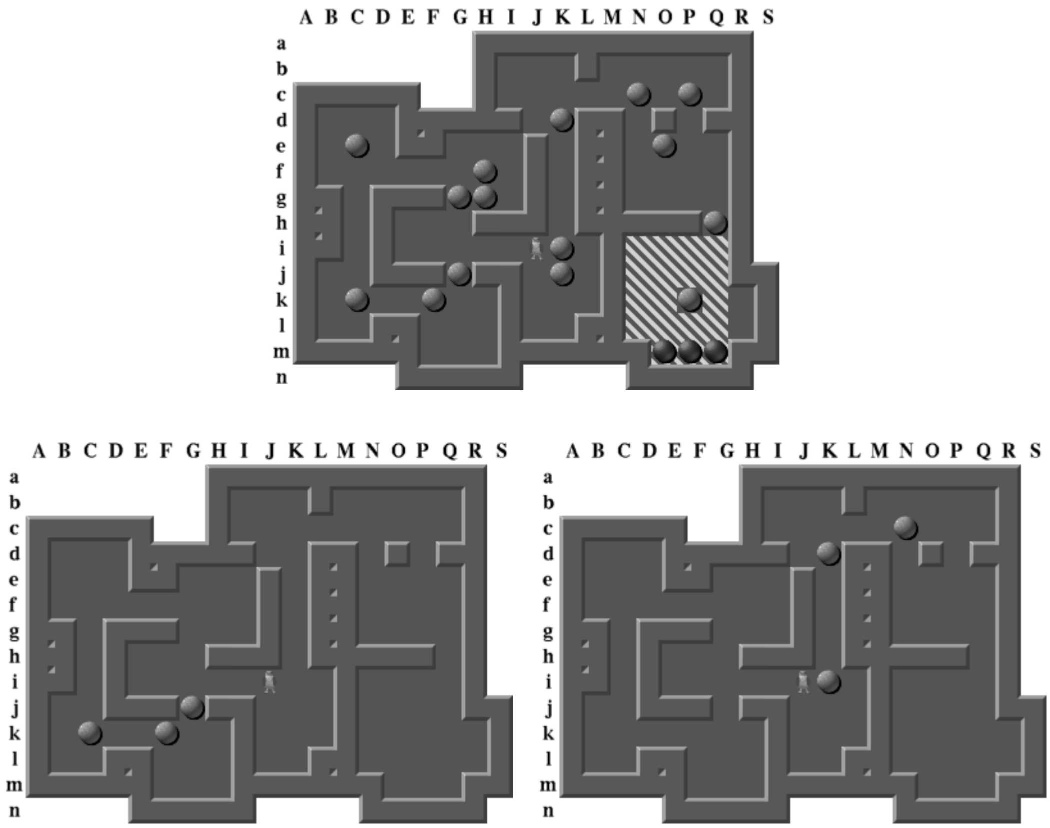


Figure 5.7: Maze #30 With a Penalty of 38 (24+14)

increases the lower bound of the root node and reduces the total number of iterations of the IDA* search.

However, increasing the lower bound at the root node will only help to save the early, small iterations, not the latter, large iterations. Since these early iterations help explore and find patterns, cutting early iterations might be detrimental to the overall performance of the program. Furthermore, executing scan searches comes with a significant overhead, usually over 20,000 nodes and as much as 58,000 nodes. Especially for searches that are small, this overhead can be significant.

5.12 Utility Considerations

Controlling the number of pattern searches and their individual costs is only part of the cost of the pattern knowledge. Whenever a lower bound is calculated, all the patterns in the database have to be tested to find which ones match. This can quickly lead to Minton's utility problem [Min88]: The costs of matching patterns slows the program down to the point where the benefits in node savings are offset by the additional cost of pattern matching. To reduce this cost, a limit can be imposed on the total number of patterns. But, which of the patterns should be kept and which should be deleted?

We chose to limit the number of patterns to 800[5]. When this pattern limit is reached, we remove the worst pattern before inserting a new one. "Worst" is defined as *least recently used*. To avoid deleting patterns before they have had time to show their worth, patterns are given a grace period of 50,000[6] nodes from the time they are created during which they are not removed. Also, once a pattern was used more than 800[7] times, it will never be removed. Thus, the pattern limit of 800 is a soft limit, it is possible that more patterns are stored.

With this limit in place, on average about half of the patterns are eliminated. The removal heuristic seems to work well, because patterns that are removed would rarely be matched. One problem is an exception: #19. Without the pattern limit, *Rolling Stone* can solve problem #19 with 17 million nodes. With the pattern limit in place, the number of nodes needed increases beyond 20 million. Because of the "softness" of the pattern limit, further decreasing the pattern limit results only in small further decrease in the number of patterns stored. Note that in many problems the pattern limit is never reached. In other problems, this limit is exceeded excessively and massive run time savings are possible when large amounts of patterns are deleted. Problem #22 is such an example. Without a pattern limit, 13,458 patterns are collected. With the soft pattern limit of 800, only 1,742 patterns are stored, significantly reducing the cost of pattern matching.

[5]Yet Another Magic Number (YAMN).
[6]YAMN.
[7]YAMN.

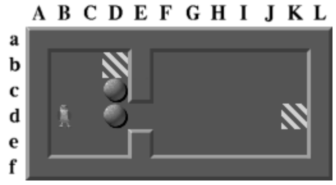


Figure 5.8: No Penalty

5.13 Related Work

The idea of storing minimal patterns is similar to Ginsberg's Partition Search [Gin96], where the entries of a hash table are generalized to hold information about sets of problem states. In *Rolling Stone* a pattern contains the information about the lower-bound increase of the set of problem states in which this pattern is present.

The notion of bit (stone) patterns can be compared to the Method of Analogies [AVAD75]. Pattern searches are a conflict-driven top-down proof of correctness, while the Method of Analogies is a bottom-up heuristic approximation.

5.14 Limitations and Open Problems

There are complications when reasoning about penalties as we have seen in Section 4.3.6. Pattern searches assume that a stone will go to its closest goal. If the optimal path to that goal cannot be used because it is obstructed, a different, potentially longer path has to be taken. A penalty is the result.

But what if the Minmatching lower bound has already targeted the stone towards a goal further away? Consider Figure 5.8. Even though we have a stone configuration that might look like a linear conflict, it is not. One of the stones has to be pushed to the goal further away. This knowledge is implicit in the lower bound. But because the pattern search assumes that each stone will go towards the closest goal, it will find a penalty of 2 in this position. Even though we have so far treated the penalties resulting from pattern searches as admissible, there are rare cases in which they are not.

This problem arises from the discrepancy between the pattern search's assumption and the reality of where the Minmatching is targeting the stones to. Unfortunately, there is no general way of solving this problem, without conditioning the penalties. These conditions would have to account for the assumptions of the pattern search and each pattern matching would have to verify that the current Minmatching is not violating these assumptions (*e.g.* which goals a stone can/cannot move to.) It is an open problem how to encode these conditions efficiently. In the version of *Rolling Stone* described here, this problem is completely ignored, resulting in the occasional wrong penalty (and possibly non-optimal solution).

An observation that we have not been able to exploit is the *hidden pattern*. Assume that at a node all successors are searched without finding a solution. That means

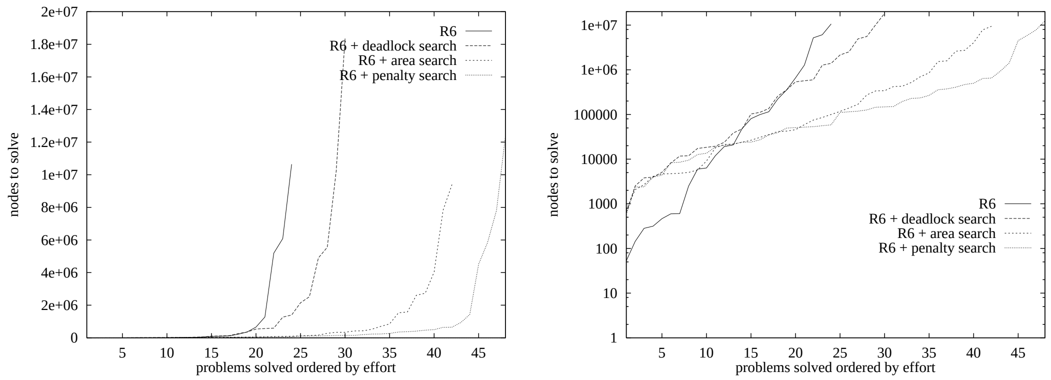


Figure 5.9: Enabling One Pattern Search (Linear and Log Scale)

the search has just proven that there is no solution for the current threshold in this subtree. However, the lower bound did not cause a cutoff when we started searching this subtree. At this point, we know that our lower bound is off. A penalty pattern remains undetected in the current position. The search has no knowledge about why it failed. Back-jumping is impossible. Just executing a pattern search to find the hidden pattern has two draw-backs:

- Presumably we did a pattern search when we created this node, starting with the stone last pushed. With which stone should we start now?
- If we find a solution before we revisit this node, then this speculative search effort would be wasted.

It seems obvious that the knowledge about the existence of a hidden pattern should be used, but we don't know how to do so efficiently.

A fundamental limitation of the current implementation of the penalty patterns and penalty searches is that stones on goals cannot be part of a pattern. The number and kind of penalty anomalies increases dramatically when stones on goals are allowed in patterns. This limits the patterns in the kinds of penalties that they can express. We experimented with stones that where fixed on goals, but found that the dynamic distances capture most of the benefits already.

5.15 Experimental Results

Rolling Stone can solve 24 Sokoban problems without pattern searches. Table 5.1 and Figure 5.9 show the effect of adding each of the pattern searches alone: area search (AR), deadlock search (DL) and penalty search (PN). Penalty searches outperform the rest of the searches clearly, solving 48 problems, that is an increase of 24! Area searches solve 6 problems less, a total of 42. Deadlock searches, the initial idea, can solve only 6 more problems than a version without any pattern searches: 30. Note the entries for problem #54. While the program enhanced with the area search can

#	-AR -DL -PN	+AR -DL -PN		-AR +DL -PN		-AR -DL +PN	
	IDA*	IDA*	IDA*+PIDA*	IDA*	IDA*+PIDA*	IDA*	IDA*+PIDA*
1	53	50	728	53	633	53	573
2	316	85	4,640	82	5,077	82	4,347
3	2,493	166	4,711	119	11,872	107	13,530
4	597	187	45,652	187	47,480	187	49,562
5	1,275,146	57,723	429,175	2,484	135,636	488	50,711
6	283	160	4,110	85	3,954	160	3,982
7	48,209	3,998	21,752	3,504	102,281	1,376	21,645
8	> 20,000,000	23,729	273,954	> 106,657	> 20,000,000	426	408,708
9	659,972	8,460	117,093	3,098	355,472	841	126,353
10	> 20,000,000	> 4,589,251	> 20,000,000	> 63,766	> 20,000,000	2,419	1,429,198
11	> 20,000,000	> 2,186,237	> 20,000,000	> 310,567	> 20,000,000	44,357	7,818,164
12	> 20,000,000	3,613,901	9,394,754	> 565,536	> 20,000,000	300,828	5,852,854
17	11,910	7,470	35,424	3,830	17,332	7,838	27,063
19	> 20,000,000	> 2,308,996	> 20,000,000	> 133,139	> 20,000,000	61,500	12,365,851
21	10,643,971	15,306	168,209	202,317	10,206,666	1,906	145,409
25	> 20,000,000	> 2,282,812	> 20,000,000	> 130,791	> 20,000,000	1,396	373,552
33	> 20,000,000	> 10,349,835	> 20,000,000	> 195,637	> 20,000,000	5,520	639,635
34	> 20,000,000	73,999	697,988	150,281	18,350,039	511	267,401
38	19,083	10,166	41,411	11,971	111,629	9,031	53,341
43	6,084,369	45,373	421,089	> 385,422	> 20,000,000	17,825	935,196
45	> 20,000,000	> 5,363,550	> 20,000,000	> 116,217	> 20,000,000	1,439	467,809
49	5,189,494	228,985	851,493	600,506	5,550,628	195,260	357,651
51	80,504	145	5,720	2,194	38,390	137	8,531
53	> 20,000,000	159	21,334	9,921	597,100	159	24,004
54	> 20,000,000	114,481	336,415	> 2,509,932	> 20,000,000	> 3,896,911	> 20,000,000
55	144	104	2,072	136	3,803	97	2,393
56	> 20,000,000	15,233	99,878	123,173	2,147,733	376	51,996
57	> 20,000,000	75,612	339,663	84,591	4,897,724	265	114,407
58	> 20,000,000	3,386	85,637	> 154,522	> 20,000,000	723	195,767
59	> 20,000,000	1,106,457	2,730,849	> 244,323	> 20,000,000	1,223	499,466
60	> 20,000,000	1,111,060	1,584,426	216,622	1,395,471	205	20,372
61	> 20,000,000	> 10,696,415	> 20,000,000	> 330,504	> 20,000,000	325	110,862
62	6,337	1,996	42,355	18,268	2,519,713	167	56,024
63	> 20,000,000	8,836	139,172	> 156,097	> 20,000,000	437	150,211
64	> 20,000,000	193,037	2,610,202	> 81,434	> 20,000,000	379	234,400
65	604	221	17,899	228	18,343	196	18,747
67	> 20,000,000	773,199	7,828,791	> 197,121	> 20,000,000	54,963	654,594
68	> 20,000,000	26,908	521,170	> 392,557	> 20,000,000	1,119	229,055
70	> 20,000,000	217,772	1,530,250	> 235,488	> 20,000,000	415	118,595
72	> 20,000,000	348	26,524	898	257,112	134	39,038
73	> 20,000,000	424	23,896	6,389	567,742	205	58,459
76	> 20,000,000	1,098,753	4,037,793	> 95,223	> 20,000,000	191,703	4,521,473
78	465	64	2,680	75	2,513	64	2,387
79	5,964	149	9,032	143	11,715	131	12,660
80	114,930	830	30,684	123	19,419	155	24,063
81	221,690	25,943	74,914	9,095	536,209	21,505	147,737
82	99,236	2,126	59,223	15,362	1,266,520	86	34,698
83	20,847	262	5,055	164	8,155	166	9,451
84	354,295	142	4,816	227	23,768	95	8,325
	>524,840,912	> 47,644,501	>174,682,633	> 7,871,059	>429,210,129	> 4,825,891	> 58,760,250

Table 5.1: Enabling One Pattern Search

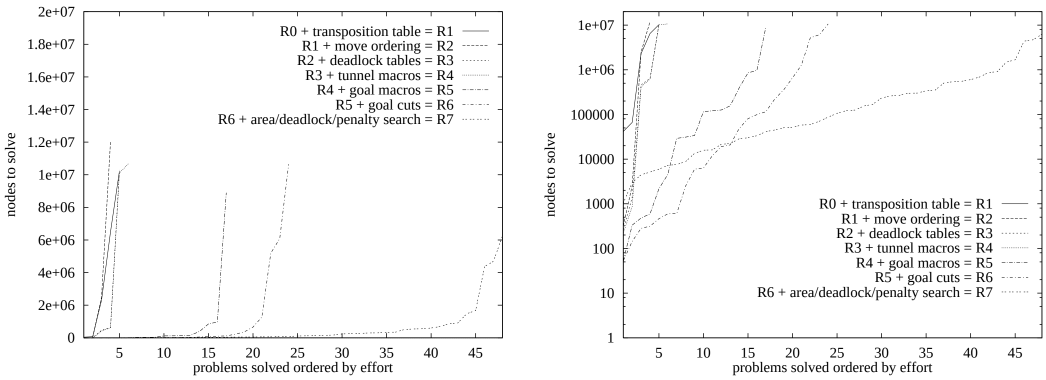


Figure 5.10: Effort Graph Including Pattern Search (Linear and Log Scale)

solve problem #54, the program with the penalty searches cannot. Area searches and penalty searches are finding different kinds of penalties.

Figure 5.10 shows the effort graph, now including the version of *Rolling Stone* using all pattern searches. Turning all the pattern searches on, we can solve 48 problems, 24 more than the previous best version! The last column of Table 5.2 shows the exact numbers.

Since penalty searches alone can solve 48 problems, why is it beneficial to include deadlock and area searches? First, small reductions in search effort are achieved. More importantly however, by allowing different kinds of pattern searches to be executed, we have some insurance against missing some types of patterns that could prevent us from finding solutions to new, unseen problems. As can be seen in Table 5.2, different combinations of pattern searches solve different problems.

Except for the small searches (<20,000 nodes), the cost of performing the additional PIDA* searches is offset by the reduction in the IDA* search nodes. Problem #53 is an example. The savings for the IDA* tree are dramatic. Previously, with 20,000,000 nodes the search was unable to solve this problem. Now the search succeeds with only 159 IDA* nodes and a total of 22,310 nodes (21,081 of those are scan-search nodes). Clearly, the pattern searches dominate the search cost, but the knowledge uncovered allows us to solve the problem where we failed previously. In this example, *Rolling Stone* searches fewer IDA* nodes than the length of the solution! The search backtracks a mere 13 times for a solution of 186 pushes.

Table 5.2 and Figure 5.11 show the version of *Rolling Stone* that uses all pattern searches and what happens when one of the pattern searches is disabled at a time. The smallest loss comes from disabling area search; 48 problems are still solved. Disabling penalty searches loses a total of 11 problems. Turning off the deadlock search loses one problem, but gains one, problem #19!

Problem #19 is an interesting case. Adding penalty searches alone allows *Rolling Stone* to solve the problem with over 12 million nodes. Further enabling area searches increases the number of nodes needed to close to 16 million. When all the pattern searches are enabled, the problem cannot be solved anymore; the overhead becomes too high.

#	-AR +DL +PN		+AR -DL +PN		+AR +DL -PN		+AR +DL +PN	
	IDA*	IDA*+PIDA*	IDA*	IDA*+PIDA*	IDA*	IDA*+PIDA*	IDA*	IDA*+PIDA*
1	53	826	50	864	50	934	50	1,042
2	82	6,122	82	5,790	82	6,468	82	7,532
3	94	13,846	107	13,472	110	10,347	94	13,445
4	187	49,324	187	49,386	187	48,527	187	50,369
5	436	60,141	478	50,071	2,031	138,172	436	59,249
6	85	4,691	160	5,120	85	4,603	85	5,119
7	1,704	26,633	1,376	23,612	2,814	67,350	1,704	28,561
8	408	550,814	328	279,027	10,890	2,141,491	317	339,255
9	810	184,307	745	125,123	1,658	210,090	704	168,412
10	2,127	2,002,162	1,926	999,098	> 80,071	> 20,000,000	1,909	1,480,115
11	14,704	4,429,873	21,985	4,778,984	> 366,200	> 20,000,000	14,048	4,691,929
12	162,263	4,233,053	300,669	6,136,043	> 720,970	> 20,000,000	162,129	4,373,802
17	3,077	25,702	6,767	44,135	3,045	29,532	2,473	30,111
19	> 63,223	> 20,000,000	75,007	15,793,144	> 96,292	> 20,000,000	> 59,433	> 20,000,000
21	1,889	190,935	1,904	125,511	32,571	2,655,175	1,853	154,593
25	1,351	568,490	1,346	417,736	> 126,416	> 20,000,000	1,239	553,900
33	5,009	838,878	5,319	649,862	> 298,642	> 20,000,000	5,035	866,085
34	582	401,802	591	299,695	52,733	5,556,437	542	298,674
38	7,576	72,264	9,031	75,401	3,363	38,608	2,539	51,276
43	16,566	1,417,432	6,758	558,133	17,389	1,205,589	5,308	690,426
45	1,086	439,895	1,799	492,574	> 123,327	> 20,000,000	1,685	508,124
49	371,153	1,246,597	> 7,229,739	> 20,000,000	403,401	2,459,295	375,293	1,670,236
51	137	9,618	137	7,760	145	10,839	137	8,825
53	159	24,008	159	22,306	159	22,310	159	22,310
54	106,663	788,320	> 2,459,627	> 20,000,000	111,832	981,285	106,663	910,532
55	97	2,651	97	2,735	104	3,074	97	2,993
56	452	62,281	381	49,590	8,495	209,164	353	57,785
57	256	122,994	265	112,900	51,777	1,860,321	256	121,384
58	716	315,546	433	170,709	2,382	383,630	426	268,713
59	1,198	668,701	812	240,794	> 391,840	> 20,000,000	795	348,214
60	205	28,124	223	29,016	2,547	127,104	223	41,310
61	290	93,241	325	111,873	> 562,852	> 20,000,000	314	106,206
62	167	60,329	211	64,446	1,865	230,463	211	70,478
63	437	198,790	567	192,649	123,280	12,431,967	567	259,537
64	370	302,697	387	238,103	> 92,640	> 20,000,000	378	300,684
65	196	19,885	196	20,433	221	20,486	196	21,442
67	52,987	905,298	18,571	620,139	> 205,793	> 20,000,000	18,107	601,178
68	1,721	336,291	2,297	359,463	21,054	2,682,015	2,278	541,080
70	413	148,995	412	104,721	556	96,670	412	125,454
72	134	46,411	134	38,519	348	50,085	134	44,908
73	201	87,068	205	58,308	363	84,811	201	87,019
76	192,230	6,726,931	334,655	5,046,214	> 300,026	> 20,000,000	185,633	6,236,656
78	64	3,219	64	3,646	64	3,702	64	4,451
79	125	14,464	131	14,065	141	13,567	125	15,833
80	100	19,640	155	22,986	102	13,849	100	16,114
81	21,501	221,154	21,505	161,099	25,269	467,708	21,501	234,235
82	86	38,450	86	33,506	1,980	183,123	86	33,445
83	91	7,867	97	7,879	91	5,759	91	7,294
84	94	5,578	95	5,944	137	21,304	94	5,960
	> 1,035,555	> 48,022,338	> 10,508,581	> 78,662,584	> 4,248,390	>274,475,854	> 976,746	> 46,536,295

Table 5.2: Disabling One Pattern Search

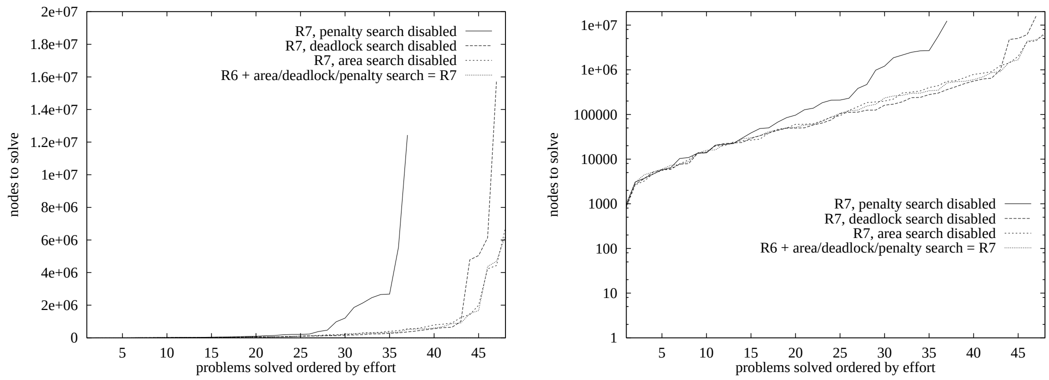


Figure 5.11: Disabling One Pattern Search (Linear and Log Scale)

Analysis of the data shows that the average growth rate of the search tree from iteration to iteration in an IDA* search decreased by a factor of over 600. Although this represents a significant reduction in search effort, it demonstrates how resistant the problem is to search. Decreasing the growth rate of the search-tree size generally increases the number of iterations that the main IDA* search can perform in the same time.

Pattern searches are a gamble: we invest search effort (PIDA* nodes) expecting to find useful knowledge. Problem #78 is one example of where the gamble does not pay off. Even though the tree size (IDA*) is reduced about 50 fold, including the PIDA* nodes triples the total number of nodes searched.

Node numbers and success rates vary for the different pattern searches. An unproductive pattern search costs between roughly 50 and 600 nodes. A productive pattern search typically costs between 600 and 3,200 nodes. While penalty searches are expensive, they are successful about 10% of the time. On the other hand, area searches are cheap, but their success rate is only about 1%. Although this sounds low, the results show the value of the discovered knowledge.

The results reported here are not the best numbers that can be achieved. In Table 5.2, the PIDA* nodes dominate the cost of the search for some problems. Some additional heuristics for deciding when to execute pattern searches could result in further improvements in the search efficiency. There are numerous parameters in the search, each of which can be tuned for maximal performance. For example:

- the effort limit in number of nodes,
- the pattern-size limit,
- the effort limit after finding a pattern,
- the control function, and
- which of the multiple conflicting stones to include next.

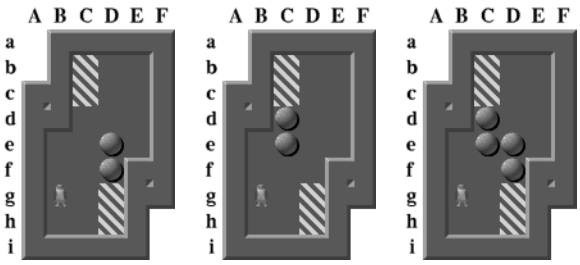


Figure 5.12: Example of C of Infinity

Building the pattern searches was easy. All the effort was spent in tuning the parameters for best performance.

5.16 Theoretical Considerations

The question arises as to whether or not pattern searches can be used in domains other than Sokoban. What fundamental properties of the domain and its heuristics are needed for pattern searches to be applicable and to produce admissible lower bounds?

5.16.1 State Description Properties

First, we will examine the domain properties. Let us assume that a state in a domain can be described by a set of descriptors $S = \{c_1, ..., c_n\}$. These c_i could relate to objects and their properties, such as location or value. For the domain of Sokoban one could imagine a c_i to describe the location of a stone. A subset $S_k \subseteq S$ is a state with fewer or the same number of such descriptors than S, for the Sokoban example, stones. A state description is *reducible*, if the solution for any state S_k is at most as long as the solution for any S:

$$|sol(S_k)| \quad \leq \quad |sol(S)|. \tag{5.1}$$

The term $|sol(S)|$ stands for the length of an optimal solution for S. It is non-negative ($|sol(\emptyset)| = 0$).

A state description is called *splitable*, if for any two disjoint subsets S_1 and S_2 of S ($S_1, S_2 \subseteq S$ and $S_1 \cap S_2 = \emptyset$) the following holds:

$$|sol(S)| \quad = \quad |sol(S_1)| + |sol(S_2)| + |sol(S - (S_1 \cup S_2))| + C \tag{5.2}$$

This means that the solution of S is at least as long as both subsolutions added. The third term accounts for additional steps that might be needed for conditions c_i that are neither in S_1 nor S_2. The term C stands for subsolution interactions. These subsolution interactions can only increase the solution length of S ($C \geq 0$).

The example Sokoban-state description is reducible, because whenever a stone (or even a wall) is removed, the solution is not getting more complicated, but potentially simpler. This state description is also splitable. In Sokoban, the term C can become as large as infinity. Consider Figure 5.12 as an example. The two linear conflicts, shown in the left (S_1) and middle maze (S_2), combine to form a deadlock when added in the right maze (S). The third term in Equation 5.2 ($|sol(S - (S_1 \cup S_2))|$) is 0, because $S - (S_1 \cup S_2) = \emptyset$. Adding the solution to the subproblems S_1 and S_2 leads to an infinitely smaller sum than the actual solution length of the right maze.

5.16.2 Heuristic Properties

Now, let us consider the properties of the admissible heuristic h used for the application domain. A heuristic is *reducible*, if the following holds:

$$h(S_k) \ \leq \ h(S), \tag{5.3}$$

given that $S_k \subseteq S$. A heuristic is *splitable*, if the following holds:

$$h(S) \ = \ h(S_1) + h(S_2) + h(S - (S_1 \cup S_2)), \tag{5.4}$$

given that $S_1, S_2 \subseteq S$ and $S_1 \cap S_2 = \emptyset$.

The Minmatching heuristic in Sokoban is reducible but not splitable, because Minmatching does take stone-goal interactions into account. This is one reason why the pattern searches use the simpler heuristic *Closest*. The lower bound is the sum of the distances of each stone to their respective closest goals. This heuristic is reducible and splitable.

5.16.3 Penalties

The pattern search can start with solving any $S_k \subseteq S$ and adding new conditions c_i will only monotonically increase the solution lengths, as defined in Equation 5.1. A penalty pattern S_k is discovered, when there is a difference between $|sol(S_k)|$ and $h(S_k)$. Since h is admissible, the following must be true:

$$|sol(S_k)| - h(S_k) \ \geq \ 0. \tag{5.5}$$

We will define the *penalty* of S_k as

$$pen(S_k) \ = \ |sol(S_k)| - h(S_k). \tag{5.6}$$

The sum $h(S_k) + pen(S_k)$ is therefore by definition admissible.

What happens with multiple penalty patterns that match in one state? When these patterns overlap, only one can be used, as previous considerations in Section 4.3.6 with the multiple linear conflicts have shown. What about non-overlapping patterns? Would the sum of the lower-bound function and all their penalties still be admissible?

Let S_1 and S_2 be two non-overlapping subproblems of S, with the usual conditions $S_1, S_2 \subseteq S$ and $S_1 \cap S_2 = \emptyset$. For $h(S) + pen(S_1) + pen(S_2)$ to be admissible ($|sol(S)| \geq h(S) + pen(S_1) + pen(S_2)$), the following must hold:

$$pen(S) \ \geq \ pen(S_1) + pen(S_2). \tag{5.7}$$

Using Equations 5.1 to 5.6 this is easy to show.

$$\begin{aligned}
pen(S) &= |sol(S)| - h(S) \\
&= |sol(S_1)| + |sol(S_2)| + |sol(S - (S_1 \cup S_2))| + C \\
&\quad -(h(S_1) + h(S_2) + h(S - (S_1 \cup S_2))) \\
&= |sol(S_1)| - h(S_1) + |sol(S_2)| - h(S_2) \\
&\quad +|sol(S - (S_1 \cup S_2))| - h(S - (S_1 \cup S_2)) \\
&\quad +C \\
&= pen(S_1) + pen(S_2) + \underbrace{pen(S - (S_1 \cup S_2)) + C}_{\geq 0} \\
&\geq pen(S_1) + pen(S_2) \qquad (5.8)
\end{aligned}$$

Thus, given the properties outlined above for the domain and the heuristic, the penalties of non-overlapping patterns can be added and the resulting heuristic remains admissible.

5.16.4 Conclusions

We were able to show the sufficient properties of the state description and the lower bound that ensure the theoretical applicability of pattern searches. If the state descriptions and the heuristic lower bound for a domain have both the properties of reducibility and splitability, pattern searches are possible. Starting with small problems (patterns), pattern searches can iteratively increase the pattern size until a penalty pattern is detected. For Sokoban, we used the conflict heuristic to determine the next c_i (stone) to include, but any other heuristic could be used. It was also shown that the penalties of non-overlapping patterns can be added to the lower bound of a position without losing admissibility. However, in practice it might not be wise to use pattern searches. Their use comes with a considerable overhead and the cost-benefit ratio will determine if pattern searches are beneficial.

We previously discussed a few inconsistencies between the pattern knowledge and the admissibility of the resulting evaluation in Sokoban. Now, we have the theoretical tools to see where these issues arise from. The pattern searches have to use a different heuristic than the main IDA* search. Therefore, the admissibility of a pattern does not necessarily carry over from the pattern search into the main IDA* search. Sokoban proves to be difficult, again.

What about other domains? How common are the properties of reducibility and splitability? The sliding-tile puzzles and Rubik's Cube have these properties. The Manhattan distance used as a lower-bound function for the sliding-tile puzzles is reducible and splitable as well. We will see in the next section, how the ideas developed for Sokoban can easily be transferred into the different domain of the 15-puzzle.

5.17 Pattern Searches in the 15-Puzzle

The 15-puzzle is reducible. Removing tiles introduces more blanks and they allow the problem to be solved faster. It is also splitable. Traditionally, the Manhattan distance is used as a lower bound. The Manhattan distance has both properties: reducibility and splitability. Therefore, the sliding-tile puzzles are perfect candidates for pattern searches from a theoretical point of view.

Practically, however, there are a number of drawbacks to this domain when trying to improve run time with pattern searches.

- Pattern searches excel at finding local conflicts by ignoring irrelevant parts of the problem. Because of the limited physical dimensions of the 15-puzzle, almost everything is local. Thus, one of the main advantages of the pattern search is diminished considerably.

- The sliding-tile puzzle programs have very little overhead per node. Move generation and lower-bound functions reduce to table lookups of small constant time. On today's fast PCs (Pentium III 450MHz) they easily search up to 8 million nodes per second. Adding any kind of overhead will slow down the program considerably, and that slowdown is hard to offset with node savings. Pattern searches will create considerable overhead, because they have to be executed and the patterns have to be matched.

- There are enhancements to the lower bound, such as the linear conflicts [HMY92], that can efficiently improve the Manhattan distance.

- While in Sokoban a move could increase the distance to the goal by an arbitrary amount, in the sliding-tile puzzles, each move can increase the distance to the goal by at most 2, because every move is reversible. Therefore, the penalties that can be found will be smaller for the 15-puzzle and thus the benefits (the likelihood of cutoffs) will be less.

Despite these obstacles, a significant reduction in node count could show the feasibility of pattern searches beyond the domain of Sokoban.

5.17.1 Implementation

We started with Korf's original implementation of a 15-puzzle solver [Kor85a].[8] It contains nothing but the Manhattan distance as lower bound, IDA* as the search algorithm and an enhancement to prevent cycles of length 2. There are no transposition tables, linear conflicts, macro moves and pattern databases.

Pattern Searches

Our implementation of the pattern search starts with a designated tile that is assumed to be part of a penalty pattern. The pattern search tries to solve the problem with

[8]We used Korf's original source code to implement our ideas.

the single tile; everything else is assumed to be blanks. The search returns a solution and a conflict set consisting of all the squares the tile moved over. The next tile included is the one in the conflict set which is closest to the first tile, and so forth. *Closest* in our implementation is a modified Manhattan distance. Every tile in the same row or column has the normal Manhattan distance. All other tiles are assigned the Manhattan distance plus 2. This ensures that all tiles in the same columns or rows are included first, in order to facilitate the detection of linear conflicts.

Patterns are restricted to 4 tiles, and each pattern search is given a limit of 50 nodes. If a pattern is found, the search continues, but the limit is increased to 250. Pattern searches are only executed to a search-tree depth of about half the IDA* threshold (i.e. are restricted to the top of the tree).

At the beginning of an IDA* search, the equivalent of a Scan Search is performed. For each tile a pattern search is called. During the IDA* search, a pattern search is executed if the tile that was moved is not part of a pattern that was included in the penalty for the current position. This reduces the number of unproductive pattern searches, even at the risk of missing a small percentage of the patterns.

Pattern Storage and Matching

To speed things up, each pattern is stored in a number of dynamic arrays. There are 16×16 such arrays, one for each tile-square combination. Each of these arrays contains all the patterns that have a specific tile on a specific square in the puzzle grid. Thus, a linear conflict with two tiles would be stored twice, once for each tile-square combination in the pattern.

To reduce the run-time overhead, we use a greedy approach when trying to determine the penalty of a position. For each tile in the puzzle we try to find the maximal penalty available for this tile using a minimal number of tiles. We commit to using this penalty. The tiles used by committed penalties are marked and excluded from further matches to ensure only non-overlapping patterns are used. This routine is not guaranteed to find the maximal penalty, but was a compromise to obtain most of the benefits of the patterns with the least amount of overhead.

5.17.2 Experimental Results

The parameters given above were tuned only a little, but improvements are certainly possible. We use the 100 problems from Korf's original test suite [Kor85a].

Figure 5.13 shows node numbers corresponding to Korf's original code (upper line), the corresponding node numbers for the pattern-search version (dots), and the same numbers sorted by increasing size (shown as the lower line). Savings of about 66% are possible with our current implementation. The nodes searched split about equally among the top-level nodes and pattern-search nodes.

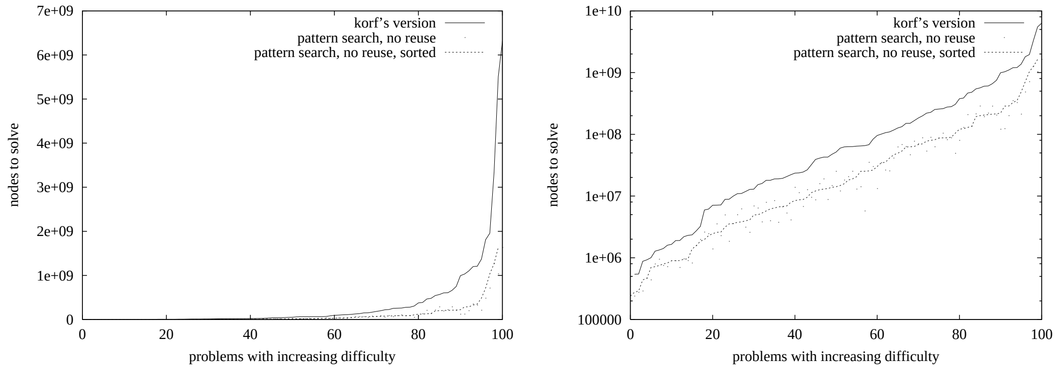


Figure 5.13: Pattern Searches in the 15-Puzzle (Linear and Log Scale)

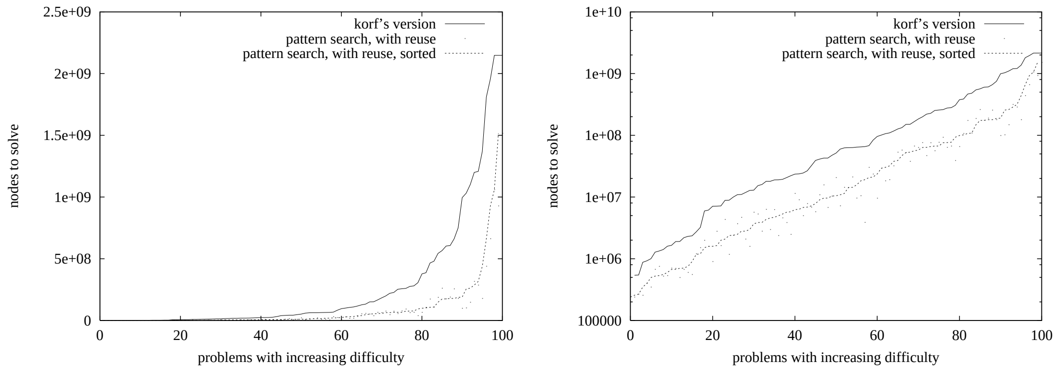


Figure 5.14: Pattern Searches in the 15-Puzzle Reusing Patterns (Linear and Log Scale)

Reusing Patterns

In Sokoban, the patterns found in one problem are not generally usable in another problem, because the layout of the maze and the location of the goals change. Since the 15-puzzle does not change its layout or the goal state, patterns found once are reusable for future problems. A test was run that retains the patterns from problem to problem. An additional 10% savings are possible, reducing the number of nodes required to roughly 24%. Figure 5.14 shows the results. Tables 5.3 and 5.4 contains the numbers for both experiments.

Run Time

Even though we could show wins with respect to node numbers, the overall run time increases. The overhead of matching the patterns is not offset by the node savings. The savings are lower because the penalties that can be found are smaller than in Sokoban, because all moves are reversible. Thus, the likelihood of them being able to cause a cutoff is smaller as well. The matching increases the time spent per node about 35-fold. That is not surprising, since the original code has an extremely low overhead.

5.17.3 Conclusions

The sliding-tile puzzles are quite different when compared to the Sokoban domain. However, they have similar properties that allow pattern searches to work. Even though the pattern searches result in significant node savings, the matching overhead is larger, because the 15-puzzle is a low-overhead domain, with an efficient and effective lower-bound function, reversible moves and high locality. The cost-benefit ratio is not in favor of pattern searches for the 15-puzzle.

The lower-bound improvement of linear conflicts can reduce the node numbers much more, but that knowledge is static and is hand coded. Pattern searches can detect much more general conflicts of tiles and are not restricted by our understanding of the domain. The main objective of this section, to show that pattern searches have potential beyond the domain of Sokoban, was realized.

5.18 Conclusions

The property of deadlocks in a search space adds considerable complexity to the search. Deadlock tables are beneficial for local deadlock detection, but inadequate to handle non-trivial situations. Pattern searches can detect global deadlock scenarios and are able to improve the lower bound considerably, resulting in a substantial improvement in search efficiency.

Patterns give the search knowledge about how the stones and the man interact. This additional knowledge allows the search to avoid parts of the search space that have no solutions and/or only solutions that are longer than the current threshold.

#	Plain IDA*	Plus Pattern Searches			Plus Pattern Searches with Reuse of Patterns		
		IDA*+PIDA*	IDA*	PIDA*	IDA*+PIDA*	IDA*	PIDA*
1	540,859	239,547	142,508	97,039	239,547	142,508	97,039
2	546,343	276,468	198,862	77,606	267,188	193,074	74,114
3	877,822	289,709	201,934	87,775	256,627	182,675	73,952
4	927,211	463,991	283,028	180,963	399,477	253,956	145,521
5	1,002,926	437,631	274,859	162,772	347,658	229,040	118,618
6	1,280,494	794,419	430,652	363,767	674,873	372,343	302,530
7	1,337,339	949,669	543,679	405,990	753,357	437,515	315,842
8	1,411,293	750,560	480,549	270,011	524,327	347,662	176,665
9	1,599,908	718,377	430,931	287,446	534,140	340,121	194,019
10	1,650,695	898,062	500,979	397,083	705,983	407,199	298,784
11	1,897,727	894,044	578,177	315,867	700,093	471,535	228,558
12	1,905,022	696,823	412,605	284,218	496,921	296,423	200,498
13	2,196,592	959,296	654,233	305,063	671,943	489,194	182,749
14	2,304,425	900,628	562,093	338,535	599,977	400,609	199,368
15	2,351,810	823,599	512,078	311,521	556,369	379,907	176,462
16	2,725,455	1,566,132	855,726	710,406	1,200,524	664,060	536,464
17	3,222,275	1,976,483	1,188,101	788,382	1,511,135	935,646	575,489
18	5,934,441	2,611,918	1,734,902	877,016	1,993,995	1,362,206	631,789
19	6,158,732	2,454,218	1,666,046	788,172	1,593,941	1,142,821	451,120
20	7,096,849	1,387,873	843,966	543,907	900,469	579,749	320,720
21	7,115,966	3,535,755	1,963,898	1,571,857	2,798,760	1,557,218	1,241,542
22	7,171,136	2,282,133	1,485,085	797,048	1,633,121	1,088,971	544,150
23	8,841,526	4,945,501	2,670,870	2,274,631	4,338,745	2,376,398	1,962,347
24	8,885,971	1,855,272	1,093,178	762,094	1,179,823	733,615	446,208
25	9,982,568	3,576,460	2,262,624	1,313,836	2,406,177	1,565,450	840,727
26	10,907,149	4,965,290	3,073,845	1,891,445	3,689,342	2,341,576	1,347,766
27	11,020,324	6,196,510	3,496,018	2,700,492	4,692,929	2,702,278	1,990,651
28	11,861,704	3,136,569	1,915,277	1,221,292	2,082,440	1,287,875	794,565
29	12,808,563	2,587,918	1,532,363	1,055,555	1,587,803	958,001	629,802
30	12,955,403	6,900,357	3,904,078	2,996,279	5,689,404	3,244,098	2,445,306
31	15,300,441	6,393,789	3,547,776	2,846,013	5,302,386	2,984,188	2,318,198
32	15,971,318	3,811,782	2,242,701	1,569,081	2,805,897	1,621,794	1,184,103
33	17,954,869	7,914,991	4,198,511	3,716,480	6,326,621	3,375,560	2,951,061
34	17,984,050	3,973,317	2,133,762	1,839,555	2,978,373	1,606,509	1,371,864
35	18,918,268	8,437,534	4,807,670	3,629,864	6,255,292	3,679,995	2,575,297
36	18,997,680	3,756,295	2,417,514	1,338,781	2,380,043	1,602,348	777,695
37	19,355,805	6,671,833	3,570,891	3,100,942	4,590,206	2,501,976	2,088,230
38	20,671,551	5,313,551	2,954,753	2,358,798	3,859,176	2,248,633	1,610,543
39	22,119,319	4,100,671	2,554,452	1,546,219	2,486,509	1,640,228	846,281
40	23,540,412	13,843,415	7,789,833	6,053,582	11,430,164	6,424,676	5,005,488
41	23,711,066	11,348,938	6,333,195	5,015,743	9,004,498	5,067,043	3,937,455
42	24,492,851	6,775,014	4,358,482	2,416,532	4,987,035	3,223,251	1,763,784
43	26,622,862	12,666,910	6,667,429	5,999,481	7,827,761	4,443,516	3,384,245
44	32,201,659	9,579,153	5,226,829	4,352,324	6,865,400	3,866,877	2,998,523
45	39,118,936	8,677,734	5,182,695	3,495,039	5,759,803	3,499,410	2,260,393
46	41,124,766	15,859,652	9,256,714	6,602,938	10,856,777	6,696,506	4,160,271
47	42,693,208	18,962,035	10,458,390	8,503,645	15,720,595	8,760,179	6,960,416
48	42,772,588	8,786,352	4,544,081	4,242,271	6,769,114	3,553,502	3,215,612
49	47,506,055	14,785,232	8,744,593	6,040,639	10,364,124	6,382,613	3,981,511
50	51,501,543	25,153,931	13,156,713	11,997,218	20,651,664	10,892,728	9,758,936

Table 5.3: Experimental Results for the 15 Puzzle (I)

#	Plain IDA*	Plus Pattern Searches			Plus Pattern Searches with Reuse of Patterns		
		IDA*+PIDA*	IDA*	PIDA*	IDA*+PIDA*	IDA*	PIDA*
51	59,802,601	12,074,808	7,129,909	4,944,899	7,198,419	4,642,938	2,555,481
52	62,643,178	18,193,320	10,275,974	7,917,346	14,344,342	8,275,975	6,068,367
53	63,036,421	20,635,688	12,069,795	8,565,893	14,248,474	8,790,770	5,457,704
54	63,276,187	25,257,298	13,569,322	11,687,976	21,132,597	11,433,740	9,698,857
55	64,367,798	12,986,228	7,853,068	5,133,160	9,633,860	6,022,843	3,611,017
56	64,926,493	14,115,956	7,803,041	6,312,915	10,480,885	5,924,554	4,556,331
57	65,533,431	5,740,186	3,704,815	2,035,371	3,885,367	2,573,896	1,311,471
58	67,880,055	35,223,825	17,184,643	18,039,182	30,288,769	14,792,560	15,496,209
59	83,477,693	30,115,905	18,253,002	11,862,903	23,599,128	14,608,924	8,990,204
60	95,733,124	13,243,348	6,998,780	6,244,568	9,588,006	5,208,168	4,379,838
61	100,734,843	34,617,503	17,763,516	16,853,987	29,281,463	14,983,111	14,298,352
62	106,074,302	26,373,781	15,994,142	10,379,639	18,261,302	11,288,695	6,972,607
63	109,562,358	25,511,033	14,047,076	11,463,957	19,080,166	10,744,264	8,335,902
64	117,076,110	41,217,374	20,879,485	20,337,889	31,983,878	16,701,461	15,282,417
65	126,638,416	62,396,258	31,005,988	31,390,270	53,339,225	26,754,479	26,584,746
66	132,945,855	68,225,185	34,463,403	33,761,782	57,658,610	29,456,218	28,202,392
67	150,346,071	63,446,995	34,816,085	28,630,910	52,569,497	29,576,740	22,992,757
68	151,042,570	46,491,721	24,539,442	21,952,279	37,919,076	20,271,548	17,647,528
69	166,571,020	77,540,544	40,375,099	37,165,445	66,931,919	34,669,581	32,262,338
70	183,526,882	69,997,313	34,168,994	35,828,319	63,421,285	31,048,124	32,373,161
71	198,758,702	87,766,281	42,008,873	45,757,408	76,176,289	36,961,403	39,214,886
72	220,374,384	53,558,763	27,602,309	25,956,454	46,973,422	24,213,232	22,760,190
73	226,668,644	89,537,891	45,389,557	44,148,334	75,627,138	39,075,879	36,551,259
74	252,783,877	62,831,732	33,492,626	29,339,106	56,096,209	30,155,218	25,940,991
75	257,064,809	86,979,437	45,004,537	41,974,900	77,236,295	40,110,310	37,125,985
76	260,054,151	104,589,901	53,261,229	51,328,672	93,056,576	47,351,110	45,705,466
77	276,361,932	81,643,020	42,051,951	39,591,069	63,776,719	32,769,750	31,006,969
78	280,078,790	87,685,598	43,861,039	43,824,559	66,929,943	34,529,085	32,400,858
79	306,123,420	49,255,148	27,457,411	21,797,737	39,012,336	22,198,724	16,813,612
80	377,141,880	79,935,949	38,767,275	41,168,674	65,789,788	32,312,648	33,477,140
81	387,138,093	129,453,081	66,898,520	62,554,561	107,124,954	54,052,442	53,072,512
82	465,225,697	208,661,165	101,646,975	107,014,190	174,073,215	85,885,772	88,187,443
83	480,637,866	134,867,828	73,184,295	61,683,533	108,870,176	59,592,195	49,277,981
84	543,598,066	214,335,023	111,301,616	103,033,407	187,555,926	97,691,048	89,864,878
85	565,994,202	287,160,880	137,981,294	149,179,586	261,971,051	125,619,342	136,351,709
86	602,886,857	192,918,480	94,363,487	98,554,993	174,974,578	85,013,556	89,961,022
87	607,399,559	221,127,539	133,786,893	87,340,646	189,874,898	115,809,263	74,065,635
88	661,041,935	286,235,800	139,080,450	147,155,350	255,709,707	125,105,040	130,604,667
89	750,745,754	204,800,942	105,361,957	99,438,985	178,697,814	92,261,405	86,436,409
90	995,472,711	121,031,679	60,948,602	60,083,077	99,164,337	51,023,344	48,140,993
91	1,031,641,139	123,558,336	67,093,299	56,465,037	101,684,685	55,108,784	46,575,901
92	1,101,072,540	200,479,934	101,685,292	98,794,642	147,986,726	76,928,139	71,058,587
93	1,199,487,995	351,686,002	175,182,845	176,503,157	316,488,886	159,031,174	157,457,712
94	1,207,520,463	327,199,700	170,091,890	157,107,810	286,948,657	149,565,191	137,383,466
95	1,369,596,777	212,473,752	120,284,300	92,189,452	178,676,839	103,461,995	75,214,844
96	1,809,933,697	484,738,238	253,644,380	231,093,858	439,595,890	231,180,827	208,415,063
97	1,957,191,377	712,647,322	349,499,366	363,147,956	661,802,397	325,793,021	336,009,376
98	3,337,690,330	1,269,110,078	586,394,155	682,715,923	1,059,849,206	491,356,136	568,493,070
99	5,506,801,122	1,034,169,009	480,236,703	553,932,306	928,186,443	434,107,129	494,079,314
100	6,320,047,979	1,633,097,516	774,048,263	859,049,253	1,516,183,963	721,815,799	794,368,164
	36,302,807,931	10,093,823,634	5,020,547,096	5,073,276,538	8,803,189,857	4,399,402,805	4,403,787,052

Table 5.4: Experimental Results for the 15 Puzzle (II)

This knowledge comes at a price: executing the speculative pattern searches. However, the overhead of pattern searches is well worth the effort in the domain of Sokoban. The knowledge gained allows dramatic improvements in efficiency and leads to twice the number of problems solved. Given 20 million nodes of search effort, our program can now solve 48 problems of the 90 problem test suite.

Pattern searches can be used in other domains, if reducable and splitable state descriptions and heuristics can be found. The 15-puzzle is such a domain. However, to be of practical benefits, the savings of the pattern searches must outweight their considerable overhead. While this is true for Sokoban, the 15-puzzle did not benefit from pattern searches in our implementation.

Chapter 6

Relevance Cuts

6.1 Introduction and Motivation

It is commonly acknowledged that a human's ability to successfully navigate through large search spaces is due to their meta-level reasoning [Gin93b]. The relevance of different actions when composing a plan is an important notion in that process. Each next action is viewed as one logically following in a series of steps to accomplish a (sub-)goal. An action judged as irrelevant is not considered.

When searching small search spaces, the computer's speed at base-level reasoning can effectively overcome the lack of meta-level reasoning by simply enumerating large portions of the search space. However, it is easy to identify a problem that is simple for a human to solve (using reasoning) but is exponentially large for a computer to solve using standard search algorithms. The pigeonhole problem is an example: fit $N+1$ stones into N pigeonholes. We need to enhance search algorithms to be able to reason at the meta-level if they are to successfully tackle these larger search tasks. In the world of computer games (two-player search), a number of meta-level reasoning algorithmic enhancements are well known, such as null-move searches [GC90] and futility cutoffs [Sch86]. For single-agent search, macro moves [Kor85b] are an example.

In this chapter, we introduce *relevance cuts*, a meta-level reasoning enhancement for single-agent search. The search is restricted in the way it chooses its next action. Only actions that are related to previous actions can be performed, with a limited number of exceptions being allowed. The exact definition of relevance is application dependent.

Consider an artist drawing a picture of a wildlife scene. One way of drawing the picture is to draw the bear, then the lake, then the mountains, and finally the vegetation. An alternate way is to draw a small part of the bear, then draw a part of the mountains, draw a single plant, work on the bear again, another plant, maybe a bit of lake, *etc.* The former corresponds to how a human would draw the picture: concentrate on an identifiable component and work on it until a desired level of completeness has been achieved. The latter corresponds to a typical computer method: the order in which the lines are drawn does not matter, as long as the final result is achieved.

Unfortunately, most search algorithms do not follow the human example. At each node in the search, the algorithm will consider all legal moves regardless of their relevance to the preceding play. For example, in chess, consider a passed "a" pawn and a passed "h" pawn. The human will analyze the sequence of moves to, say, push the "a" pawn forward to queen. The computer will consider dubious (but legal) lines such as push the "a" pawn one square, push the "h" pawn one square, push the "a" pawn one square, *etc.* Clearly, considering alternatives like this is not cost-effective.

What is missing in the above examples is a notion of *relevance*. In the chess example, having pushed the "a" pawn and then decided to push the "h" pawn, it seems silly to now return to considering the "a" pawn. If it really was necessary to push the "a" pawn a second time, why weren't both "a" pawn moves considered *before* switching to the "h" pawn? Usually this switching back and forth (or "ping-ponging") does not make sense but, of course, exceptions can be constructed.

In other well-studied single-agent search domains, such as the N-puzzle and Rubik's Cube, the notion of relevance is not important. In both of these problems, the geographic space of moves is limited, *i.e.* all legal moves in one position are "close" (or local) to each other. For two-player games, the effect of a move may be global in scope and therefore moves almost always influence each other (this is most prominent in Othello, and less so in chess). In contrast, a move in the game of Go is almost always local. In non-trivial, real-world problems, the geographic space might be large, allowing for moves with local and non-local implications.

This chapter introduces relevance cuts and demonstrates their effectiveness in Sokoban. For Sokoban, we use a new influence metric that reflects the structure of the maze. A move is considered relevant only if the previous m moves influence it. The search is only allowed to make relevant moves with respect to previous moves and only a limited number of exceptions are permitted. With these restrictions in place, the search is forced to spend its effort locally, since random jumps within the search space are discouraged. In the meta-reasoning sense, forcing the program to consider local moves is making it adopt a pseudo-plan; an exception corresponds to a decision to change plans.

The search-tree size, and thus the search effort expended in solving a problem, depends on the depth of the search tree and the effective branching factor. Relevance cuts aim at reducing the effective branching factor. For *Rolling Stone*, relevance cuts result in a large reduction of the search space. On the standard set of 90 test problems, relevance cuts allow *Rolling Stone* to increase the number of problems it can solve from 48 to 50. Given that the problems increase exponentially in difficulty, this relatively small increase in the number of problems solved represents a significant increase in search efficiency.

6.2 Relevance Cuts

Analyzing the trees built by an IDA* search quickly reveals that the search algorithm considers move sequences that no human would ever consider. Even completely unrelated moves are tested in every legal combination—all in an effort to prove that

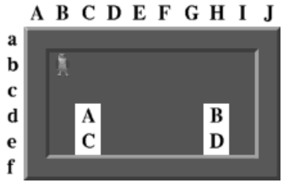


Figure 6.1: The Number of Alternatives Changes the Influence

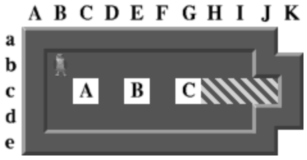


Figure 6.2: The Location of the Goals Matters

there is no solution for the current threshold. How can a program mimic an "understanding" of relevance? We suggest that a reasonable approximation of relevance is *influence*. If two moves do not influence each other, then it is unlikely that they are relevant to each other. If a program had a good "sense" of influence, it could assume that in a given position all previous moves belong to a (unknown) plan of which a continuation can only be a move that is relevant—in our approximation, is influencing whatever was played previously.

6.2.1 Influence

An influence metric can be achieved in different, domain-specific ways. The following shows one implementation for Sokoban. Even though the specifics aren't necessarily applicable to other domains, the basic philosophy of the approach is.

We approximate the influence of two moves on each other by the influence between the move's *from* squares. The influence between two squares is determined using the notion of a "most influential path" between the squares. This can be thought of as a least-cost path, except that influence is used as the cost metric.

When judging how two squares in a Sokoban maze influence each other, using the Euclidean distance is not adequate. Taking the structure of the maze into account would lead to a simple geographic distance which is not proportional to influence either. For example, consider two squares connected by a tunnel; the squares are equally influencing each other, no matter how long the tunnel is. Elongating the tunnel without changing the general topology of the problem would change the geographic distance, but not the influence.

The following is a list of properties we would like the influence measure to reflect:

Alternatives: The more alternatives exist on a path between two squares, the less the squares influence each other. That is, squares in the middle of a room where stones can go in all 4 directions should decrease influence more than squares

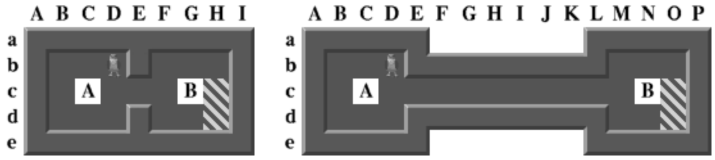


Figure 6.3: Tunnels and Influence

in a tunnel, where no alternatives exist. See Figure 6.1 for an example. The squares A and B influence one another less than the squares C and D. There are more possible ways to get from A to B than from C to D. Squares C and D are more restricted because they are situated on a wall.

Goal-Skew: For a given square sq, any squares on the optimal path from sq to a goal should have stronger influence than squares off the optimal path. For example, square B in Figure 6.2 is influenced by C more than it is by A. The location of the goals is important.

Connection: Two neighboring squares connected such that a stone can move between them should influence each other more than two squares connected such that only the man can move between them. In Figure 6.1, square A influences C less than C influences A, because stones can only move towards C, and not towards A.

Tunnel: In a tunnel, influence remains the same: It does not matter how long the tunnel is (one could, for example, collapse a tunnel into one square). Figure 6.3 shows such an example: two problem mazes that are identical, except for the length of the tunnel. Influence values should not change because of the length of the tunnel.

Our implementation of relevance cuts uses small off-line searches to statically precompute a $(20 \times 20) \times (20 \times 20)$ table ($InfluenceTable$) containing the influence values for each square of the maze to every other square in the maze. Between every pair of squares, a breadth-first search is used to find the path(s) with the largest influence. The algorithm is similar to a shortest-path finding algorithm, except that we are using influence here and not geographic distance. The smaller the influence number, the more two squares influence each other. See Appendix C.4 for details.

Note that influence is not necessarily symmetric.

$$InfluenceTable[a, b] \neq InfluenceTable[b, a]$$

A square close to a goal influences squares further away more than it is influenced by them. Furthermore, $InfluenceTable[a, a]$ is not necessarily 0. A square in the middle of a room will be less influenced by each of its many neighbors than a square in a tunnel. To reflect that, squares in the middle of a room receive a larger bias than more restricted squares.

Our approach is quite simple and can undoubtedly be improved. For example, influence is statically computed. A dynamic measure, one that takes the current positions of the stones into account, would certainly be more effective.

6.2.2 Relevance Cut Rules

Given the above influence measure, we can now proceed to explain how to use that information to cut down on the number of moves considered in each position. To do this, we need to define *distant moves*. Given two moves, $m1$ and $m2$, move $m2$ is said to be distant with respect to move $m1$ if the from squares of the moves ($m1.from$ and $m2.from$) do not influence each other. More precisely, two moves influence each other if

$$\mathit{InfluenceTable}[m1.from, m2.from] \ <= \mathit{infthreshold}$$

where *infthreshold* is a tunable threshold.

Relevance cuts eliminate some moves that are distant from the previous moves played (*i.e.* do not influence), and therefore are considered not relevant to the search. There are two ways that a move can be cut off:

1. If within the last m moves more than k distant moves were made. This cut will discourage arbitrary switches between non-related areas of the maze.

2. A move that is distant with respect to the previous move, but not distant to a move in the past m moves. This will not allow switches back into an area previously worked on and abandoned just briefly.

In our experiments, we set k to 1. This way, the first cut criterion will entail the second.

To reflect differences in mazes, the parameters $infthreshold$ and m are set at the beginning of the search, taking the average values in the $InfluenceTable$ into account. By varying $infthreshold$ and m in the definition of relevance, the cutting in the search tree can be made more or less aggressive. The desired aggressiveness is application dependent, and should be chosen relative to the quality of the relevance metric used.

6.2.3 Example

Figure 6.4 shows an example where humans immediately identify that solving this problem involves considering two separate subproblems. The solution to the left and right sides of the problem are completely independent of each other. An optimal solution needs 82 pushes; *Rolling Stone*'s lower bound estimator returns a value of 70. Standard IDA* will need 7 iterations to find a solution (our lower-bound estimator preserves the odd/even parity of the solution length, meaning it iterates by 2 at a time). IDA* will try every possible (legal) move combination, intermixing moves from both sides of the problem. This way, IDA* proves for each of the first 6 iterations

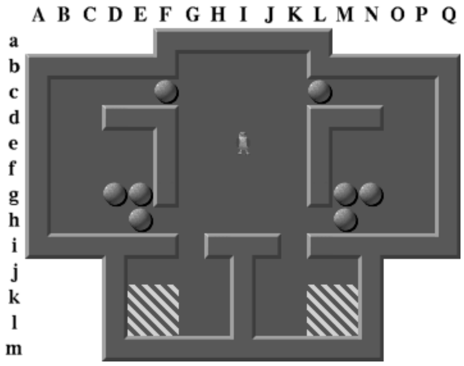


Figure 6.4: Example Maze With Locality

($i = 0..5$) that the problem cannot be solved with $70 + 2 * i$ moves, regardless of the order of the considered moves. Clearly, this is unnecessary and inefficient. Solving one of the subproblems requires only 4 iterations, since the lower bound is off by only 6. Considering this position as two separate problems will result in an enormous reduction in the search complexity.

Our implementation considers all moves on the left side as distant from those on the right, and *vice versa*. This way only a limited number of switches is considered during the search. Our parameter settings allow for only one non-local move per 9-move sequence. For this contrived problem, relevance cuts decrease the number of nodes searched from 32,803 nodes to 24,748 nodes while still returning an optimal solution (the pattern searches were turned off for simplicity). The savings (25%) appear relatively small because the transposition table catches repeated positions (many of which may be the result of irrelevant moves) and eliminates them from the search. Although the relevance cuts provide a welcome reduction in the search effort required, it is only a small step towards achieving all the possible savings. For example, each of the subproblems can be solved by itself in only 329 nodes! The difference between 329×2 and 32,803 illustrates why IDA* in its current form is inadequate for solving large, non-trivial real-world problems. Clearly, more sophisticated methods are needed.

6.2.4 Discussion

Further refinement of the parameters used are certainly possible and necessary if the full potential of relevance cuts is to be achieved. Some ideas with regards to this issue will be discussed in Section 6.5.

The overhead of the relevance cuts is negligible, at least for our current implementation. The influence of two moves can be established by a simple table lookup. This is in stark contrast to the pattern searches, where the overhead dominates the cost of the search for most problems.

6.3 A Closer Look at Relevance Cuts

The goal of using relevance cuts is to reduce the search-tree size. This is achieved by eliminating legal moves from the search, thereby reducing the effective branching factor of the tree. As with many other (unsafe) forward pruning techniques, this could potentially remove solutions or postpone their discovery. Therefore, aggressive pruning can increase the search effort by requiring additional search to find a non-pruned solution. A solution could be found in the same IDA* iteration, or could result in an additional iteration being started. A good heuristic for relevance is the key to finding the right balance between tree reduction and the risk of eliminating solutions.

6.3.1 Relevance Cuts in Theory

To better understand the implications of relevance cuts, we will now try to apply Korf's theoretical model [Kor97] to our algorithm.[1] Section 6.4.2 discusses how well the model predicts the practical performance of our algorithm.

The number of nodes considered in a standard IDA* search is given by the following formula, which is a generalization of Korf's model.

$$n \approx \underbrace{\sum_{i=h(root)}^{d-1} b^{i-e}}_{complete\ iterations} + \underbrace{\frac{b^{d-e}}{1+s_d}}_{last\ (partial)\ iteration} \tag{6.1}$$

where

n is the total number of nodes;

d is the length of optimal solutions;

$h(root)$ is the heuristic value of the root node ($<= d$);

b is the effective branching factor;

e is the average heuristic value of the interior nodes in the tree; and

s_d is the number of solutions with (optimal) length d.

In this formula, the variable-depth search tree is approximated as a fixed-depth tree. With no lower-bound information ($h(position) = 0$), the search tree would be of size $O(b^d)$. An average lower bound of e reduces this exponent to $d - e$.

The first part of the formula represents the sum of the sizes of all the iterations that have no solution in them. The second part is the size of the last iteration. It

[1]Korf and Reid refine this model in [KR98]. The irregularity of the search space (irreversible moves and search enhancements such as transposition tables) and heuristic function (caused by the numerous lower-bound enhancements) for Sokoban and *Rolling Stone*render this model less suitable than the one in [Kor97].

assumes that the solutions are uniformly distributed throughout the leaf nodes. Thus, if there is only one unique solution path, that solution will be found, on average, half way through the search of the last (d) iteration.

Relevance cuts modify the equation in two ways. First, the iterations without solutions are reduced in size. This is achieved by eliminating moves from consideration, in effect reducing the branching factor. Second, there is the possibility that additional search will be needed if the first solution happens to be eliminated by a relevance cut. Thus, on iterations $>= d$ the savings from the reduced branching factor can be (partially) offset by having to do extra work. If all solutions at depth d happen to be cut off, then at least one more iteration is required (and possibly more). Equation 6.1 is modified to reflect both ways that relevance cuts affect the search:

$$
\begin{aligned}
n &\approx \underbrace{\sum_{i=h(root)}^{d-1} (b - r(x))^{i-e}}_{complete\ iterations} + \underbrace{\sum_{i=d}^{d+a(x)-1} (b - r(x))^{i-e}}_{additional\ full\ iterations} + \underbrace{\frac{(b - r(x))^{d+a(x)-e}}{1 + (1 - p(x)) * s_{d+a(x)}}}_{last\ (partial)\ iteration} && (6.2)\\
&\approx \underbrace{\sum_{i=h(root)}^{d+a(x)-1} (b - r(x))^{i-e}}_{complete\ iterations} + \underbrace{\frac{(b - r(x))^{d+a(x)-e}}{1 + (1 - p(x)) * s_{d+a(x)}}}_{last\ (partial)\ iteration} && (6.3)
\end{aligned}
$$

where

x is the aggressiveness of the cuts (in our relevance metric, this corresponds to changing m or *infthreshold*);

$r(x)$ is the average branching-factor reduction as a function of the aggressiveness;

$p(x)$ is the probability that a solution is cut from the search tree, assuming these probabilities are independent. This probability also depends on the aggressiveness x of the relevance cuts;

$a(x)$ is the expected number of additional iterations. This number depends on the aggressiveness x of the cuts, and the probability that these cuts will eliminate *all* solutions in an iteration; and

$s_{d+a(x)}$ is the number of solutions at level $d + a(x)$.

The effectiveness of relevance cuts in reducing the search-tree size depends solely on the aggressiveness of the cuts, which controls the branching-factor reduction and the penalty incurred for missing a solution. Increasing the aggressiveness of the cuts will decrease the number of nodes searched in the complete iterations (iterations $< d$), but will increase the risk of solutions being cut off. When solutions are cut off, not only can the last iteration potentially grow, but we might actually introduce new iterations when all the solutions contained in an iteration are pruned. Hence, relevance cuts can introduce non-optimal solutions, or postpone the discovery of solutions beyond the resource limits.

The performance tuning effort must therefore be directed towards finding the right balance between savings (reduced search-tree size) and cost (the overhead of having to search further than should be needed).

6.3.2 Randomizing Relevance Cuts

In a deterministic environment, where relevance cuts follow the exact same rules for the same situation, the search will always cut off solutions that depend on a maneuver mistakenly considered "irrelevant". Given that relevance cuts will make mistakes (albeit, hopefully, at a very low rate), some mechanism must be introduced to avoid worst-case scenarios, such as eliminating *all* solutions.

A solution is to introduce randomness into the relevance cut decision. If a branch is to be pruned by a relevance cut, a random number can be generated to decide whether or not to go ahead with the cut. The randomness reflects our confidence in the relevance cuts. For example, the random decision can be used to approve 100% of all possible relevance cuts (corresponding to the scheme outlined thus far, confident that not all solutions will be eliminated), down to 0% (which implies no confidence—relevance cuts will never be used). Somewhere between these two extremes is a percentage of cuts that balances the reductions in the search-tree size with the overhead of postponing when a solution is found.

6.4 Experimental Results

Our previous best version of *Rolling Stone* (R6) was capable of solving 48 of the test problems within the tree-size limit of 20 million nodes. With the addition of relevance cuts (no random cutting), the number of problems solved has increased to 50. Table 6.1 shows a comparison of *Rolling Stone* with and without relevance cuts for each of the 50 solved problems.[2]

The tree size for each program version given in Table 6.1 is again broken into two numbers: IDA* nodes and total nodes, including pattern-search nodes. The third column gives the number of IDA* iterations that the program took to solve the problem. Note that problems #9, #11, #12, #21, #25, #34 and #38 are now solved non-optimally, taking at least one iteration longer than the program without relevance cuts. This confirms the unsafe nature of the cuts. However, since none of the problems solved before is lost and 2 more are solved within the 20,000,000 node limit, the gamble paid off. The size of the search space dictates radical pruning measures if we want to have any chance of solving some of the tougher problems.

Table 6.1 shows that relevance cuts improve search efficiency by at least a factor of 2 in IDA* nodes. The savings in terms of total nodes are less with about 25%. Clearly, the numbers are dominated by a few problems, such as #19 and #40.

[2]The numbers reported in [JS98a, JS99b] differ slightly from the ones presented here. Since these publications, *Rolling Stone* was significantly improved, specifically the pattern searches, allowing for a much more efficient search. The resulting smaller searches allowed less room for improvement.

#	without relevance cuts			with relevance cuts		
	IDA* nodes	total nodes	iterations	IDA* nodes	total nodes	iterations
1	50	1,042	2	50	1,042	2
2	82	7,532	1	80	7,530	1
3	94	13,445	1	87	12,902	1
4	187	50,369	1	187	50,369	1
5	436	59,249	2	202	43,298	2
6	85	5,119	1	84	5,118	1
7	1,704	28,561	2	1,392	28,460	2
8	317	339,255	3	291	311,609	3
9	704	168,412	2	1,884	435,388	5
10	1,909	1,480,115	1	1,810	1,713,429	1
11	14,048	4,691,929	10	5,679	2,994,297	11
12	162,129	4,373,802	3	4,912	559,184	8
17	2,473	30,111	7	2,038	29,116	7
19	59,433	$>$ 20,000,000	9	16,606	7,269,595	9
21	1,853	154,593	6	1,177	179,734	7
25	1,239	553,900	6	21,536	5,784,086	7
33	5,035	866,085	3	2,765	586,684	3
34	542	298,674	2	11,431	1,981,993	3
38	2,539	51,276	5	7,011	154,969	6
40	41,131	$>$ 20,000,000	6	23,274	17,004,253	7
43	5,308	690,426	7	1,729	421,483	7
45	1,685	508,124	2	339	181,566	2
49	375,293	1,670,236	9	53,113	327,643	9
51	137	8,825	1	256	21,491	1
53	159	22,310	1	157	22,308	1
54	106,663	910,532	2	163,757	2,031,577	2
55	97	2,993	1	97	2,993	1
56	353	57,785	3	377	61,189	3
57	256	121,384	2	234	114,416	2
58	426	268,713	2	211	130,474	2
59	795	348,214	4	1,420	775,753	4
60	223	41,310	1	160	27,386	1
61	314	106,206	5	309	105,411	5
62	211	70,478	3	195	101,934	3
63	567	259,537	1	703	312,546	1
64	378	300,684	4	405	332,402	4
65	196	21,442	2	196	21,442	2
67	18,107	601,178	6	12,669	512,488	6
68	2,278	541,080	6	1,953	538,509	6
70	412	125,454	3	431	140,765	3
72	134	44,908	2	134	44,908	2
73	201	87,019	1	214	94,568	1
76	185,633	6,236,656	4	74,315	3,775,394	4
78	64	4,451	1	64	4,913	1
79	125	15,833	2	122	15,527	2
80	100	16,114	1	165	26,943	1
81	21,501	234,235	1	2,662	42,445	1
82	86	33,445	2	86	33,445	2
83	91	7,294	1	80	5,631	1
84	94	5,960	1	106	7,938	1
	1,017,877	$>$ 66,536,295		419,155	49,388,544	

Table 6.1: Experimental Data

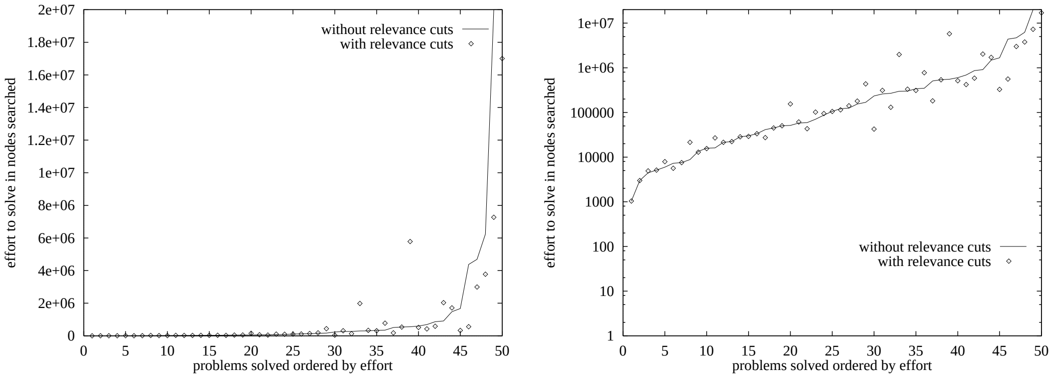


Figure 6.5: The Effect of Relevance Cuts

Comparing node numbers of individual searches is difficult because of many volatile factors in the search. For example, a relevance cut might eliminate a branch from the search justifiably. However, by doing so a pattern search might now not be done that could have uncovered valuable information that might have been useful for reducing the search in other parts of the tree. Problem #80 is one such example: despite the relevance cuts the node count goes up from 100 to 165; an important discovery was not made and the rest of the search increases. However, the overall trend is in favor of the relevance cuts. An excellent example is problem #49: the total nodes are cut by roughly a factor of 5.

In Figure 6.5, the amount of effort to solve a problem, with and without relevance cuts, is plotted. The numbers from Table 6.1 are used, sorted by the number of nodes searched by the version without relevance cuts. The figure shows that the exponential growth in difficulty with each additional problem solved is being dampened by relevance cuts, allowing for more problems being solved with the same search constraints. For the 25 to 30 "easiest" problems, there is very little difference in effort required; the relevance cuts do not save significant portions of small search trees. As the searches become larger, the success of relevance cuts gets more pronounced. However, there are two problems where relevance cuts result in a large increase in node numbers: #25 and #34. Their numbers increase roughly 10 and 6 fold, respectively.

Figure 6.6 shows the effort graph, now including the relevance cuts. Only the last problems show that relevance cuts are beneficial.

6.4.1 Randomizing Relevance Cuts

The numbers presented so far deal with a version of *Rolling Stone* that executes 100% of the relevance cuts. A version of *Rolling Stone* was instrumented to simulate the effects of different degrees of randomization, varying from 0% (all relevance cuts are ignored) to 100% (all relevance cuts are used). Thus, the level of, for example, 80% corresponds to randomly accepting 80% of the cuts, while rejecting 20% of them.

Figure 6.7 illustrates the relevance cuts' potential for savings in the search tree. The graph presents for various degrees of randomness (from 0% to 100% in 10%

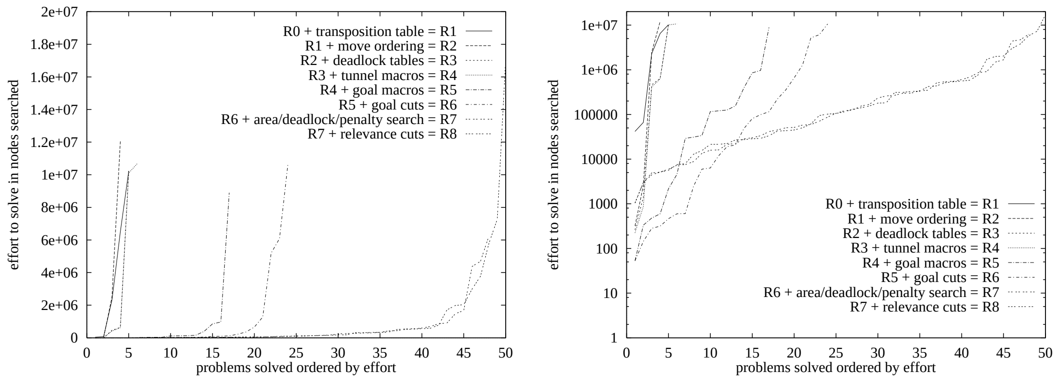


Figure 6.6: Effort Graph Including Relevance Cuts (Linear and Log Scale)

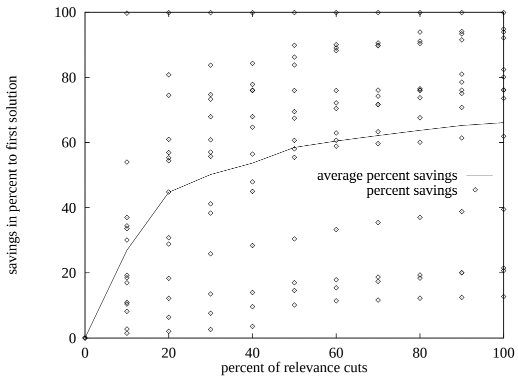


Figure 6.7: Relevance Cuts Savings

increments) the percent of the search tree that can be saved by the relevance cuts. For each search, the relative savings are plotted. Only searches where the 0% version required at least 500 nodes and the 100% version found a solution were included. The small search trees (< 500 nodes) were excluded from this and subsequent graphs, since these trees tend to have very few opportunities for savings. For example, problem #1 is already a paltry 50 nodes; there is neither need nor room for further improvement. Each of the data points in a column corresponds to one of the 14 problems that passed our filter. The line represents the average of the savings.

The figure shows that roughly 65% of the search tree can be eliminated by relevance cuts. Further, in our implementation one need only perform 50% of the cuts to reduce the search by 60%. Thus, even a small amount of cutting can translate into large savings.

To put this into perspective, one might suggest that the relevance cuts are just a fancy way of randomly cutting branches in the search tree. An additional experiment was performed with random cutting, in line with the frequency of relevance cuts. The result was some savings for a small amount of cutting, but as the frequency of cutting increased, so did the search-tree sizes! By cutting randomly, more solution paths were being eliminated from the search, increasing the likelihood of having to search more

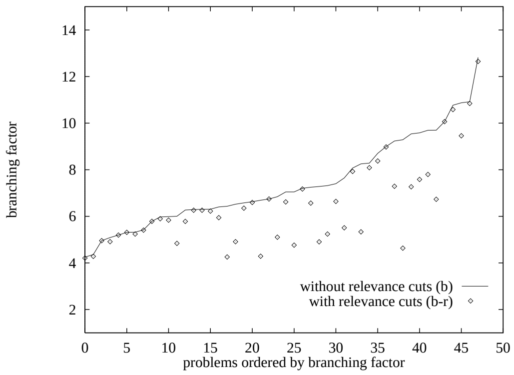


Figure 6.8: Measuring b and r

iterations.

Equation 6.3 essentially broke the relevance-cut search nodes into two components. The first was the search effort required to reach the first solution. Clearly, relevance cuts provably reduce this portion of the search since some branches are not explored. In fact, Figure 6.7 is portraying exactly these savings. However, these savings can be offset by the the second component, the additional effort needed to find a non-cutoff solution.

Of the 50 problems solved, 7 have non-optimal solutions (14%). As stated earlier, solution quality is not a concern since, given the difficulty of the problem domain, *any* solution is welcome. Furthermore, for solutions of lengths typical in Sokoban (several hundred) adding two or four pushes is a small increase. The significance of these non-optimal solutions is discussed in the next subsection.

6.4.2 Relevance Cuts in Theory Revisited

Let's revisit Equation 6.3. These generic formulas contain several assumptions, some of which are explicitly stated in [Kor97], while others are implicit. In theory, we should be able to use our experimental data to confirm these equations. Of interest in Equation 6.3 is that the term

$$(b \ - \ r)^{d-e} \tag{6.4}$$

dominates the calculation. We know d (the optimal solution length), and we can measure b, r and e. *Rolling Stone* has been instrumented to measure these quantities.

Figure 6.8 shows the average b and r for the 48 problems that were solved by both versions, sorted in order of increasing b. These statistics were gathered at nodes in the search that were visited by both programs (one with relevance cuts; the other without). In other words, nodes which were visited only by the non-relevance cuts program were not averaged in. As can be seen, the reduction in branching factor varies dramatically, depending on the problem.

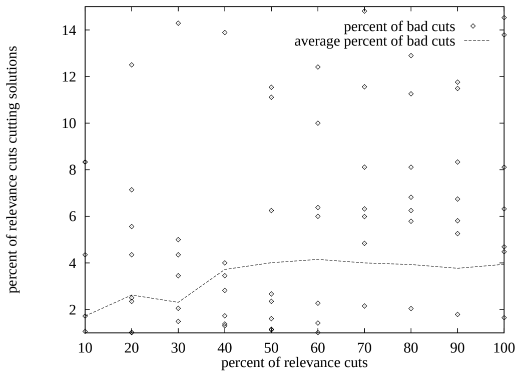


Figure 6.9: Percent of Relevance Cuts Eliminating Solutions

Measuring e, the average heuristic value of the interior nodes in the tree, showed little difference with/without relevance cuts.

Plugging d, e, b and r into Equation 6.4 produced a large discrepancy between the predicted tree size and the observed tree size. Since d is constant in both versions of the program, and e is effectively a constant, the improvements of relevance cuts rests solely on r, the reduction in the branching factor. However, in most cases the observed savings are *larger* than the predicted savings.

Equation 6.3 has the implicit assumption that the branching factor is relatively uniform throughout the tree. Certainly this is true for the sliding-tile puzzle. But Sokoban has different properties. In particular, the branching factor can swing wildly from move to move. Also, our data shows that the branching factor tends to be smaller near the root of the tree (too many obstacles in the puzzle) and, as the problem simplifies (jams get cleared, stones get pushed to their goal squares), the branching factor increases until near the end of the game when there are few stones left to move and the branching factor decreases again. In addition, the data shows that the relevance cuts tend to occur early in the search, rather than later. Hence, the majority of the savings from relevance cuts come from the smaller branching factor b near the root of the tree combined with a larger branch reduction. Korf's formula only considers averages over the entire tree, whereas any bias towards the root of the tree can produce larger observed reductions.

The other component of Equation 6.3 is the additional search effort required when relevance cuts miss the first solution. Earlier, it was suggested that the probability of searching an extra iteration was quite high (14%). This suggests that the relevance cuts are being too extreme in their cutting. *Rolling Stone* was instrumented to keep searching subtrees that would have been eliminated by a relevance cut to determine if a solution path lay in that subtree. Figure 6.9 shows that only about 2-4% of the cuts eliminate a solution. Note that some problems have a relatively high error rate; these results come from the problems that have small searches, where the total number of cuts is small and a single error can skew the percentages.

A relevance cut error rate of 4% might seem high. However, consider that these cuts are done throughout the tree, including near the root. Given that a cut near the

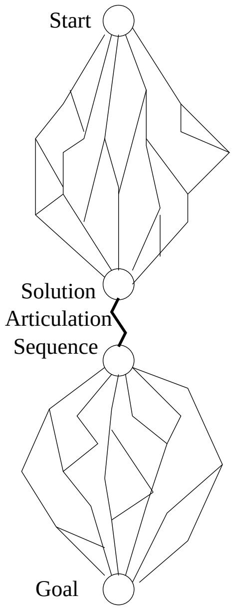


Figure 6.10: Solution Articulation Sequence

root of the search will eliminate huge portions of the search space, and few of these cuts eliminate any optimal solution, the cuts must be doing a good job of identifying irrelevant portions of the search.

Infrequently eliminating solutions may seem important if there are few solutions. In fact, our experience with Sokoban shows that there are many optimal solutions for every problem. The number of solution paths grows exponentially with any additional search beyond the optimal solution length. For example, consider a d-ply optimal solution. If we now look at solutions of length $d+2$,[3] then we can randomly insert irrelevant moves into the solution path, giving $O(d*b)$ more solution paths.

Equation 6.3 assumes that the probability of a solution being cut off is independent of any other solution being cut off. Unfortunately, this is a simplifying assumption that does not hold for Sokoban. Since Sokoban problems have been composed to be challenging to humans (and, inadvertently, computers as well), most problems in our test suite contain specific maneuvers that are *mandatory* for all solutions. In other words, every solution to some problems requires a specific sequence of moves to be made. We call these maneuvers *solution articulation sequences.*

A solution articulation sequence is illustrated in Figure 6.10. It shows the set of move sequences that are solutions to the problem of getting from the start state to the goal state. First, there are many possible sequences of moves (possibly even move transpositions) until a specific maneuver is required. Then a fixed sequence of moves is required (the solution articulation sequence). Having completed the sequence, then many different permutations of moves can be used to reach the goal(s). Note that a problem may have multiple solution articulation sequences. As well, there may be

[3]In general, this would be $d+1$. However, since Sokoban solutions preserve odd/even parity, solutions increase by two pushes at a time.

classes of solutions, with each class having a different set of articulation sequences.

Relevance cuts use a sequence of moves (the past m moves) to decide whether to curtail the search or not. If the moves forming the solution articulation sequence happen to meet the criterion for a relevance cut, then it will be falsely considered "irrelevant". Consequently, many solution paths will be eliminated from the search. One can construct a scenario by which *all* solutions could be removed from the search.

Solution articulation sequences illustrate that the assumed solution independence property is, in fact, incorrect. Coming up with a realistic model is difficult. The solutions tend to be distributed in clusters. Many clusters of solutions are, essentially, the same solution with minor differences (such as move transpositions or, for non-optimal solutions, irrelevant moves added).

Although the number of optimal solutions appears high from our experiments, relevance cuts are vulnerable to solution articulation sequences. Hence, a single cut has the potential for eliminating *all* solutions. Randomization seems to be an effective way of handling this problem.

6.4.3 Summary

Relevance cuts have been shown experimentally to result in large reductions in the effort required to solve Sokoban problems. Given the exponentially increasing nature of the search trees, solving an extra 2 problems represents a substantial improvement.

Although it would be nice to have a clean analytic model for Sokoban searches that could be used to predict search effort, this is proving elusive. Although a model for single-agent search exists [Kor97], it is inadequate to handle the non-uniformity of Sokoban. In the past, numerous analytic models for tree-searching algorithms have appeared in the literature. They are all based on simplifying assumptions that make the analysis tractable, but result in a model that mimics an artificial reality. Historically, these models correlate poorly with empirical data from real-world problems. An interesting recent example from two-player search can be found in [PSPdB96].

6.5 Conclusions

Relevance cuts provide a crude approximation of human-like problem-solving methods by forcing the search to favor local moves over global moves. This simple idea provides large reductions in the search-tree size at the expense of possibly returning a longer solution. Given the breadth and depth of Sokoban search trees, finding optimal solutions is a secondary consideration; finding *any* solution is challenging enough.

We have numerous ideas on how to improve the effectiveness of relevance cuts. Some of them include:

- Use different distances depending on *crowding*. If many stones are crowding an area, it is likely that the relevant area is larger than it would be with fewer stones blocking each other. Dynamic influence measures should be better than static approaches.

- There are several parameters used in the relevance cuts. The settings of those are already dependent on properties of the maze. These parameters are critical for the performance of the cuts and are also largely responsible for increased solution lengths. More research on these details is needed to fully exploit the possibilities relevance cuts are offering.

- Using the analogy from Section 6.1, one could characterize *Rolling Stone* as "painting" locally but not yet painting in an "object oriented" way. If a flower and the bear are close, painting both at the same time is very likely. Better methods are needed to further understand subgoals, rather than localizing by area.

Although relevance cuts introduce non-optimality, this is not an issue. Once humans solve a Sokoban problem, they have two choices: move on to another problem (they are satisfied with the result), or try and re-solve the same problem to get a better solution. *Rolling Stone* could try something similar. Having solved the problem once, if we want a better solution, we can reduce the probability of introducing non-optimality in the search by decreasing the aggressiveness of the relevance cuts. This will make the searches larger but, on the other hand, the last iteration does not have to be searched, since a solution for that threshold was already found.

Relevance cuts are yet another way to significantly prune Sokoban search trees. We have no shortage of promising ideas, each of which potentially offers another order of magnitude reduction in the search-tree size. Although this sounds impressive, our experience suggests that each factor of 10 improvement seems to yield no more than 2 or 3 additional problems being solved.

Chapter 7

Overestimation

7.1 Introduction and Motivation

To ensure optimality of solutions produced by A*-based algorithms, such as IDA*, the heuristic has to be admissible. The admissibility constraint limits the choice of knowledge. Even if some knowledge correlates well with the distance to the goal, but there is the slightest chance that it overestimates, it cannot be used. Solution optimality would not be guaranteed.

This shows that optimality has its price. Instead of fitting the function h as closely as possible to h^*, we are restricted to creating a lower bound. The error of such a lower-bound function is often larger than a function that is allowed to occasionally overestimate. The larger the error of the lower-bound function, the less efficient the search.

We have seen in previous chapters that an aggressive treatment of the search space is needed to make significant progress. The examples of the goal macros and relevance cuts have shown the benefits that are achievable when the small risk of losing optimality and completeness is taken. Therefore, it seems logical to question the admissibility constraint for the heuristic function. The hope is to achieve a closer fit of the heuristic function h to the correct distance h^*, albeit at the cost of non-optimal solutions.

7.2 WIDA*

To achieve a better approximation of h^*, one can scale the admissible heuristic by a constant factor. Statistical tests measuring the difference between h and h^* can produce a constant w that can be used. *Weighted IDA** (WIDA*) uses the cost function $f(s) = g(s) + w * h(s)$, with $w > 1$ [Kor93].

This scaling has the effect of a *depth bonus*. The further the search penetrates into the tree, the more it is encouraged. Nodes close to the root will have larger h values and the scaling will inflate these values more in absolute terms than those nodes closer to the leaves with smaller h values. Each move that decreases the h value will implicitly receive a small bonus, because the cost of the move is not balancing the decrease of h; f drops as the search approaches leaf nodes. This small decrease in the f value with each move deeper into the tree will eventually allow a non-optimal move to be considered. This can lead to radical shifts of where the search effort is spent, further towards nodes deeper in the tree. In Sokoban, trees are highly irregular and these shifts can lead to large changes in the number of nodes searched per IDA* iteration.

WIDA* increases h uniformly. The only knowledge implicitly entailed in this scheme is that nodes deeper in the tree are preferred, because they tend to be closer to the goal nodes. Because of deadlocks and arbitrary penalties that might remain undetected in Sokoban, nodes deeper in the tree are not necessarily closer to goals. The search might end up expanding more effort in parts of the tree that contain no solution.

7.3 Pattern Overestimation

The lack of domain knowledge used in WIDA* leads to poor performance when traversing Sokoban search trees. What knowledge could be used to improve the overestimation? The obvious choice is the dynamic pattern knowledge. How can this be used effectively?

Since the pattern searches are limited in certain ways to keep them tractable, the correct size of the penalties and shape of the patterns might not be known. Therefore, the patterns represent incomplete knowledge. Furthermore, when patterns are matched, only some of the penalties can be used to preserve admissibility (see Section 5.9 for details). However, each of the patterns that is matched in a position suggests that there are complications in the current position. Not using the penalty of such a pattern is equivalent to ignoring available knowledge.

7.3.1 Maximum Partial Penalties

The following is the best of our attempts to use the knowledge contained in all the patterns that match in a position. We call this method *maximum partial penalties.*

Instead of maximizing and adding the penalties of patterns, the penalties are attributed to the stones in the maze. The penalty of a pattern that is matched is split equally among all the stones contained in the pattern. For each stone the maximum of these partial penalties is stored. The total penalty of a position is the sum of all the maximum partial penalties for each stone. Thus, every stone involved in a penalty pattern contributes to the total penalty assigned to a stone configuration.

This total penalty is at least as large as the admissible penalty achieved by the methods described in Section 5.9. The following explains why:

- Non-overlapping patterns are contributing in the same way as before.
- For the admissible penalty, some patterns cannot be used because they overlap with others. That means that some stones do not contribute to the penalty, even though they are part of a penalty pattern that was matched. When using

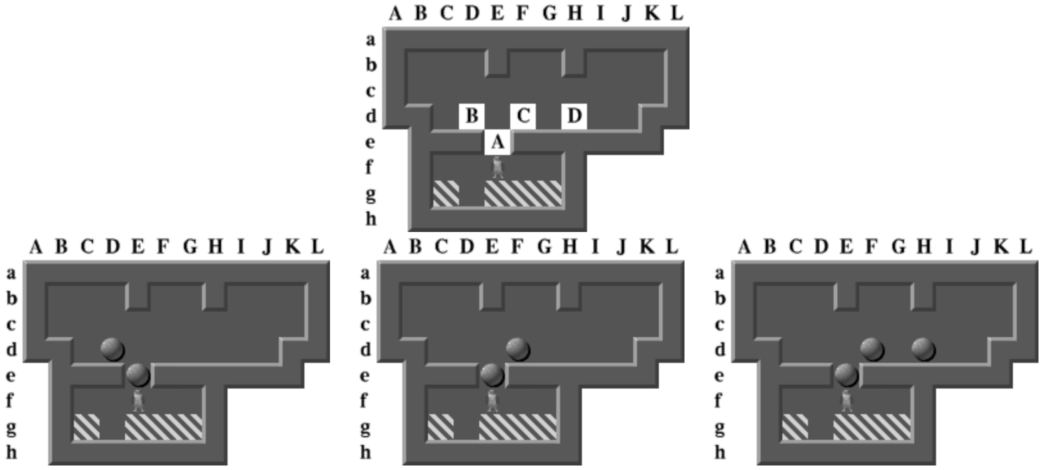


Figure 7.1: Maximum Partial Penalty Example

pattern	penalty	partial penalties			
		A	B	C	D
left	2	1.0	1.0	0	0
center	2	1.0	0	1.0	0
right	8	2.66	0	2.66	2.66
maximum partial penalty		2.66	1.0	2.66	2.66
sum of maximum partial penalties		9.0			
scaled by 1.8		16.2			
rounded for parity		16			

Table 7.1: Calculation of Maximum Partial Penalties

maximum partial penalties, each stone of a matching pattern contributes to the total penalty.

- The contribution of each stone to the total penalty is at least as large in the maximum partial penalty method as it is for the admissible penalty, because the maximum of the partial penalties is used.

To tune the overestimation further, the penalty is scaled by a factor s. A final rounding step assures that the total penalty is an even number to preserve the parity property of the heuristic.

7.3.2 Example

The upper maze in Figure 7.1 shows a position with four stones A,B,C, and D. The lower three mazes show three penalty patterns presumably found by the search. The penalties are 2, 2 and 8 for the patterns from left to right. Table 7.1 shows the maximum partial penalty calculation. For each pattern (1,2, and 3) the stones in that pattern share the penalty evenly. Summing the maximum partial penalties gives 9.0. When scaling it by s =1.8, a value of 16.2 results[1]. Rounding it to the next factor of two sets the final penalty to 16, twice the original penalty of 8.

The position in Figure 7.1 is a deadlock – any increase is justified. Other positions, such as multiple linear conflicts as seen in Section 4.3.6, will be incorrectly overestimated. The scaling factor s has to be carefully tuned to optimize the benefits of the maximum partial penalties, balancing the advantages and dangers of overestimation.

7.3.3 Pruning *versus* Postponing

Adding a limited penalty to the heuristic estimation of the distance to the goal will only delay the examination of a node. If no solution can be found, the threshold will increase until the position's estimated f-value does not cause a cutoff anymore. The exploration of the node is only *postponed*. This is in stark contrast to forward pruning with fixed rules, such as deterministic relevance cuts, that will prune the same node in every iteration.

Because new patterns are added and useless patterns are dropped, the decisions to postpone a node change dynamically over the course of a search as new knowledge is found or other knowledge is discarded.

7.4 Experimental Results

7.4.1 WIDA*

We experimented with different values for w, ranging from 1.025 to 1.25, but the results suggest an unpredictable behavior. On the one hand, the search can benefit greatly, saving orders of magnitude by extending lines that lead to solutions in early iterations. On the other hand, large irrelevant parts of the search tree might be explored that have no solution for the current threshold. The blind scaling of h is not effective in Sokoban.

Figure 7.2 shows that changing w effects the search-tree sizes almost randomly. The line indicates the search-tree size of the problems solved by a version of *Rolling Stone* that does not use overestimation (R8). The problems are ordered according to increasing search-tree size. The dots in each column represent the corresponding tree sizes for *Rolling Stone* using WIDA* with different settings of w. Table 7.2 shows the exact numbers of total nodes for these versions of *Rolling Stone*. Even though one more problem can be solved when using $w = 1.15$, the erratic behavior of the search makes it difficult to justify the use of WIDA*.

7.4.2 Pattern Overestimation

Several different values for the scaling factor s of the total penalty were tested. Figure 7.3 shows the results for a selected number of these tests. The results for this

[1] See the results section about the origin of the magic number 1.8 for s.

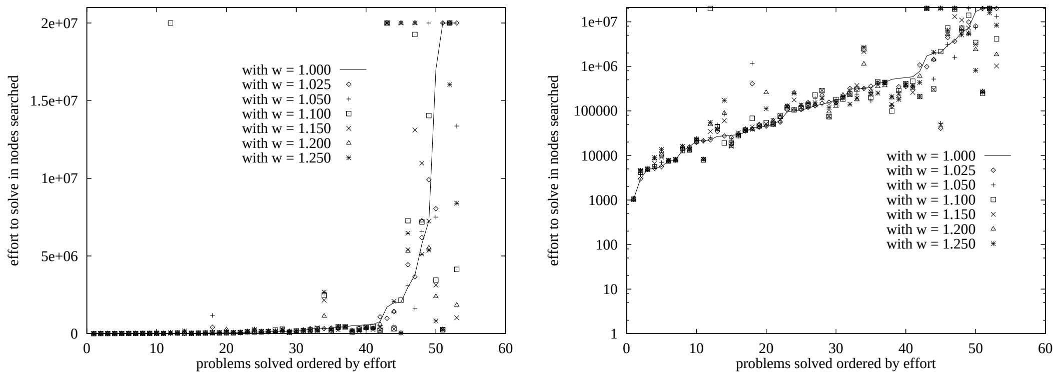


Figure 7.2: WIDA*, Varying w (Linear and Log Scale)

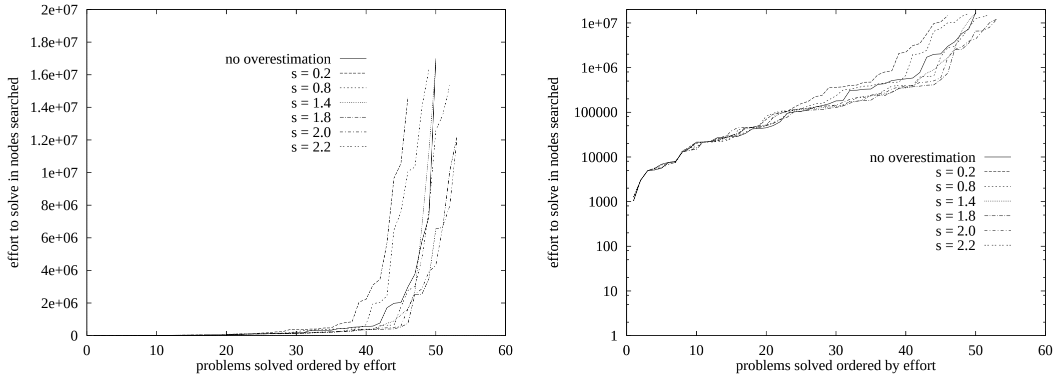


Figure 7.3: Pattern Overestimation, Varying s (Linear and Log Scale)

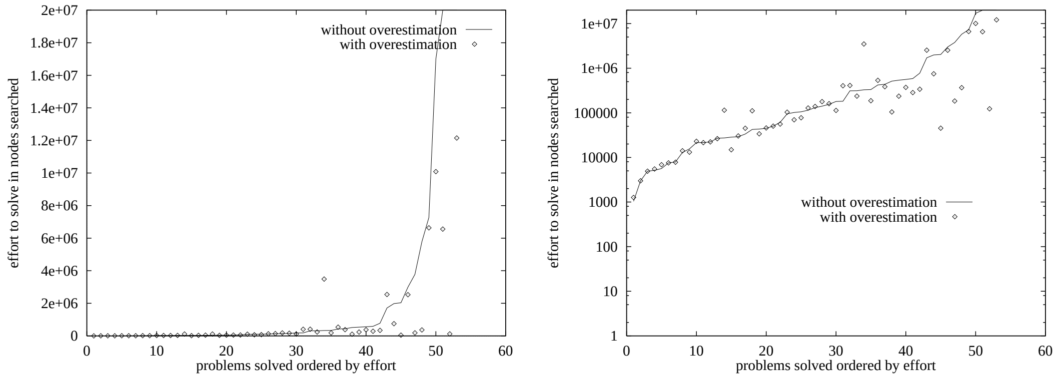


Figure 7.4: Scatter Plot for Overestimation With $s = 1.8$ (Linear and Log Scale)

Table 7.2: WIDA*, Varying w

#	$w = 1.025$			$w = 1.050$			$w = 1.100$			$w = 1.150$			$w = 1.200$			$w = 1.250$		
	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns
1	50	1,042	2	50	1,042	2	50	1,042	2	50	1,042	2	52	1,044	2	50	1,042	2
2	83	7,543	1	83	7,543	1	83	7,543	1	83	7,750	1	83	7,543	1	83	7,543	1
3	95	14,080	1	88	12,887	1	88	12,887	1	95	14,080	1	95	15,869	1	95	15,869	1
4	187	50,369	1	187	50,369	1	187	50,369	1	198	60,972	1	198	60,972	1	187	50,369	1
5	205	43,325	2	209	43,742	2	224	45,909	2	225	45,904	2	247	49,631	2	225	45,939	2
6	84	5,118	1	106	5,679	1	120	5,647	1	151	6,676	1	150	8,448	2	174	8,901	1
7	839	25,578	2	1,297	23,527	2	325	18,962	1	134	16,203	1	140	16,263	1	352	19,323	1
8	268	319,432	3	282	274,534	3	223	237,563	2	236	230,065	2	216	230,593	2	158	141,539	1
9	1,884	435,449	5	1,897	440,025	5	1,884	435,498	5	1,897	440,025	5	1,369	368,445	4	1,834	429,997	5
10	1,087	987,188	1	> 18,792	> 20,000,000	1	> 21,066	> 20,000,000	1	> 17,876	> 20,000,000	1	> 21,880	> 20,000,000	1	> 21,050	> 20,000,000	1
11	8,704	4,436,816	11	5,848	3,104,222	11	16,448	7,271,274	12	10,754	5,406,302	12	10,057	5,315,444	12	12,435	6,463,687	12
12	936	354,151	8	1,377	351,783	8	1,068	412,189	9	1,008	378,364	9	1,008	379,934	8	1,083	411,965	8
17	2,204	31,052	7	1,808	27,202	7	1,845	27,906	7	2,280	32,214	7	1,822	28,577	7	1,794	28,906	7
19	21,796	9,906,276	9	> 132,815	> 20,000,000	9	119,484	14,041,498	9	14,349	7,241,004	9	20,575	5,522,483	8	15,204	5,365,996	8
21	784	153,574	6	1,186	180,938	7	1,163	180,532	7	727	149,827	6	572	126,955	6	681	158,387	6
25	19,511	6,185,785	6	23,143	6,570,080	7	25,678	7,193,315	7	37,364	10,968,201	7	22,729	7,264,953	7	12,445	5,108,715	5
26	>2,182,655	> 20,000,000	8	> 2,561,324	> 20,000,000	7	3,731	248,845	6	1,260	269,558	6	905	273,594	6	1,008	264,000	6
33	1,156	353,432	3	1,077	350,680	3	1,360	464,887	3	739	262,206	3	1,340	321,785	3	1,168	366,238	3
34	10,929	1,426,154	3	891	519,188	2	2,163	313,461	1	948	302,144	2	23,822	1,410,840	1	36,484	2,077,968	1
38	7,449	156,259	6	3,619	72,532	5	3,296	74,171	5	3,900	75,044	6	5,132	95,851	6	6,718	119,464	6
40	11,133	8,046,261	6	10,045	7,503,696	5	4,565	3,442,797	4	4,302	3,121,816	4	2,460	2,387,446	3	1,066	819,836	2
43	1,742	422,391	7	1,742	422,391	7	1,957	452,816	7	1,848	433,746	7	1,539	354,230	7	1,031	250,927	6
45	380	202,542	2	349	183,349	2	362	185,029	2	381	199,133	2	395	225,558	2	384	210,170	2
49	51,286	318,704	9	59,827	317,331	8	413,718	2,471,208	9	369,687	2,150,887	8	148,422	1,130,394	8	351,409	2,668,480	8
51	230	21,400	1	452	20,966	1	146	7,947	1	98	8,119	1	98	8,119	1	98	8,119	1
53	157	22,308	1	283	25,206	1	> 552,138	> 20,000,000	1	6,558	34,593	1	3,900	49,145	1	8,600	55,589	1
54	215	40,737	2	854	52,793	2	203,348	2,155,609	1	190	48,644	1	> 1,661,855	> 20,000,000	1	> 1,809,413	> 20,000,000	1
55	97	2,993	1	105	3,712	1	115	4,239	1	121	4,389	1	123	4,435	1	125	4,639	1
56	232	56,070	1	220	55,976	1	301	76,642	1	290	64,237	1	288	70,827	1	347	80,166	1
57	249	120,179	2	248	117,928	2	222	137,332	1	234	126,904	1	238	152,757	1	233	148,843	1
58	211	130,474	2	276	190,884	2	1,268	229,296	1	213	136,742	1	218	136,761	1	214	153,687	1
59	1,903	1,077,731	4	425	206,863	2	399	211,000	2	399	211,000	2	1,271	602,470	2	981	435,208	2
60	158	27,384	1	393	84,848	1	147	19,012	1	241	60,438	1	351	88,837	1	751	172,177	1
61	304	107,227	5	311	112,473	5	330	115,131	5	337	120,666	5	353	132,410	5	375	135,636	5
62	225	105,460	3	232	106,869	3	227	106,147	2	208	178,071	2	283	256,023	2	268	248,424	1
63	548	313,547	1	524	238,083	1	565	306,892	1	671	372,544	1	415	178,149	1	424	193,946	1
64	319	358,836	4	176	168,047	1	239	200,448	1	314	270,426	1	323	278,368	1	211	231,923	1
65	136	19,706	1	137	19,830	1	145	21,259	1	156	22,130	1	156	23,204	1	156	23,204	1
67	473	129,400	5	477	134,723	5	285	99,545	1	323	139,625	1	367	196,735	1	392	209,076	1
68	1,471	349,388	5	1,162	268,415	5	994	286,392	4	357	247,631	1	303	207,747	1	287	181,097	1
70	420	149,731	3	456	179,115	3	611	285,603	4	604	288,028	3	546	229,156	3	509	193,981	3
72	133	45,695	1	140	46,207	1	128	54,712	1	134	51,121	1	759	259,080	1	197	112,411	1
73	218	106,558	1	226	116,000	1	248	118,004	1	317	128,516	1	317	128,516	1	317	128,516	1
75	> 36,609	> 20,000,000	8	> 32,282	> 20,000,000	8	> 272,263	> 20,000,000	8	> 69,564	> 20,000,000	7	> 614,488	> 20,000,000	9	441,895	16,042,868	7
76	43,071	3,660,025	4	20,666	1,600,408	3	760,595	19,264,490	3	702,830	13,111,226	3	> 505,884	> 20,000,000	3	> 343,016	> 20,000,000	1
77	> 801,181	> 20,000,000	1	924,824	13,368,970	1	362,265	4,141,556	1	57,384	1,021,452	1	204,884	1,835,139	1	1,238,968	8,400,844	1
78	64	4,916	1	64	4,913	1	64	4,916	1	64	4,916	1	64	4,916	1	64	4,916	1
79	123	15,518	2	123	15,518	2	105	13,331	1	105	13,331	1	105	13,331	1	107	13,623	1
80	315	34,638	1	280	48,605	1	288	44,568	1	193	38,801	1	193	38,801	1	193	38,801	1
81	36,676	409,001	1	143,579	1,166,420	1	2,177	68,565	2	337	43,753	1	192	38,448	1	207	38,460	1
82	91	35,536	2	94	34,979	2	92	36,825	2	92	36,825	2	115	38,890	2	115	38,178	2
83	80	5,631	1	108	6,856	1	131	10,387	1	137	9,210	1	127	10,408	1	127	13,598	1
84	108	7,818	1	110	7,901	1	115	8,018	1	115	8,018	1	115	8,018	1	115	8,018	1
	>3,251,464	>101,240,428		> 3,957,269	>118,866,248		> 2,781,039	>125,623,214		> 1,313,078	> 88,614,519		> 3,263,539	>110,587,542		> 4,315,823	>112,351,209	

#	no overestimation			$s = 0.2$			$s = 0.8$			$s = 1.4$			$s = 1.8$			$s = 2.0$			$s = 2.2$		
	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns
1	50	1,042	2	50	1,067	2	50	1,042	2	50	1,042	2	55	1,267	3	55	1,267	3	55	1,267	3
2	80	7,530	1	82	7,532	2	80	7,530	1	80	7,530	1	80	7,530	1	80	7,530	1	80	7,530	1
3	87	12,902	1	92	13,168	3	87	12,902	1	94	14,095	1	94	14,095	1	95	15,929	1	95	15,929	1
4	187	50,369	1	187	50,369	1	187	50,369	1	187	50,369	1	187	50,369	1	187	50,369	1	187	50,369	1
5	202	43,298	2	215	44,715	3	202	42,224	2	153	33,755	1	153	33,755	1	151	38,041	1	211	40,344	2
6	84	5,118	1	91	5,659	3	84	5,118	1	84	5,503	1	84	5,503	1	84	5,503	1	84	5,503	1
7	1,392	28,460	2	1,351	29,306	4	967	22,624	2	748	25,931	2	338	14,832	2	132	14,194	3	111	17,040	3
8	291	311,609	3	300	362,122	5	279	313,066	4	285	351,139	4	315	409,714	5	321	465,538	5	249	252,727	4
9	1,884	435,388	5	2,032	470,297	8	1,872	429,765	5	1,472	349,224	4	1,591	385,084	4	1,626	395,046	4	1,644	402,507	4
10	1,810	1,713,429	1	2,685	3,110,294	3	16,970	13,999,701	2	3,919	2,702,015	2	2,920	2,539,524	3	5,535	4,324,956	3	> 37,634	> 20,000,000	4
11	5,679	2,994,297	11	6,464	3,453,690	19	4,483	2,438,271	12	1,919	1,272,688	10	4,058	2,527,286	12	4,744	2,872,254	13	3,071	1,766,229	14
12	4,912	559,184	8	6,330	847,715	9	5,967	670,189	8	26,773	1,637,292	8	951	372,264	8	954	378,407	8	986	396,851	8
17	2,038	29,116	7	2,115	31,024	7	2,035	29,258	7	2,125	29,600	8	2,158	30,242	9	2,063	26,631	9	2,010	25,531	9
19	16,606	7,269,595	9	22,146	9,650,935	10	18,147	7,590,151	8	> 143,629	> 20,000,000	8	14,178	6,631,475	10	16,767	7,952,770	11	16,196	7,675,969	12
21	1,177	179,734	7	1,254	223,583	11	1,219	191,023	7	606	116,004	5	573	113,042	3	752	132,953	3	556	120,465	3
23	> 59,498	> 20,000,000	7	> 52,530	> 20,000,000	9	> 65,636	> 20,000,000	8	40,379	11,282,331	11	23,337	6,555,398	12	11,944	2,550,595	11	12,122	2,771,673	13
25	21,536	5,784,086	7	28,330	10,556,135	11	20,643	6,479,152	7	435	193,212	5	683	366,035	7	426	195,203	6	1,267	393,192	6
26	>2,125,116	> 20,000,000	9	>2,010,809	> 20,000,000	13	>2,137,211	> 20,000,000	10	>2,727,425	> 20,000,000	5	380	122,997	7	354	120,553	7	359	130,072	6
33	2,765	586,684	3	3,658	798,615	11	1,595	452,670	3	576	274,321	1	604	283,926	1	1,083	365,889	1	490	227,156	1
34	11,431	1,981,993	3	12,148	2,221,831	9	11,401	1,962,631	4	14,771	895,284	4	9,746	749,787	2	7,259	504,465	2	29,854	3,014,988	3
36	> 23,467	> 20,000,000	5	> 19,856	> 20,000,000	10	> 21,205	> 20,000,000	6	24,934	16,799,961	6	18,338	12,150,606	7	7,173	3,911,640	6	23,032	13,542,898	7
38	7,011	154,969	6	7,067	156,235	8	7,001	154,959	5	6,556	133,843	2	10,473	160,176	1	11,115	165,784	1	8,865	137,754	1
40	23,274	17,004,253	7	> 24,473	> 20,000,000	8	22,342	16,318,067	7	11,139	6,692,116	8	16,725	10,086,547	9	20,835	11,944,211	10	26,555	15,352,541	11
43	1,729	421,483	7	1,871	474,358	8	1,939	468,265	7	2,571	564,007	7	2,225	535,148	8	2,647	592,528	8	3,252	639,213	9
45	339	181,566	2	884	406,750	8	830	389,749	3	719	372,366	2	602	404,217	2	450	334,328	1	278	174,861	1
49	53,113	327,643	9	53,575	371,122	12	53,124	328,829	10	17,267	137,288	7	441,638	3,486,905	9	865,286	6,700,434	9	1,677,679	12,600,342	9
51	256	21,491	1	230	21,400	1	230	21,400	1	256	21,491	1	256	21,491	1	256	21,491	1	256	21,491	1
53	157	22,308	1	157	22,308	1	157	22,308	1	157	22,308	1	157	22,308	1	157	22,308	1	157	22,308	1
54	163,757	2,031,577	2	165,732	2,065,955	5	163,886	2,033,377	2	>1,041,418	> 20,000,000	3	269	45,332	3	282	46,343	3	>1,944,336	> 20,000,000	4
55	97	2,993	1	97	2,993	1	97	2,993	1	97	2,993	1	97	2,993	1	97	2,993	1	97	2,993	1
56	377	61,189	3	709	69,328	7	445	59,326	1	534	42,300	6	911	55,865	7	4,253	136,754	8	8,093	145,689	9
57	234	114,416	2	1,206	129,778	5	1,067	119,393	2	223	120,485	1	209	128,282	1	211	145,836	1	226	116,911	1
58	211	130,474	2	443	239,008	2	443	234,188	2	211	130,474	2	231	138,838	3	231	138,838	3	231	138,838	3
59	1,420	775,753	4	80,206	14,628,874	6	82,403	10,340,916	5	3,638	256,708	5	602	337,905	4	36,157	1,645,577	4	1,462	386,523	4
60	160	27,386	1	162	24,479	3	150	24,450	1	21,087	122,505	1	18,100	114,642	1	18,100	114,642	1	18,100	114,642	1
61	309	105,411	5	8	84,242	4	>1,524,270	> 20,000,000	4	387	126,272	8	299	77,555	8	335	122,881	9	308	82,726	9
62	195	101,934	3	206	105,058	5	172	66,839	3	184	75,002	4	180	69,728	4	183	77,825	5	208	94,658	5
63	703	312,546	1	765	361,208	3	1,022	354,324	1	520	242,420	1	473	237,196	1	476	263,201	1	942	643,671	1
64	405	332,402	4	> 860,545	> 20,000,000	11	573	384,994	3	221	173,204	4	193	186,508	1	284	213,755	1	219	311,995	1
65	196	21,442	2	204	21,535	6	168	20,222	1	145	21,190	1	165	23,004	1	192	24,459	1	156	22,130	1
67	12,669	512,488	6	12,922	691,462	12	1,727	136,904	5	379	94,003	1	298	104,356	1	298	105,639	1	298	105,639	1
68	1,953	538,509	6	2,327	399,233	10	1,732	473,317	5	549	239,606	5	324	236,157	1	371	240,815	1	317	232,112	1
70	431	140,765	3	463	175,073	3	453	155,760	3	428	173,712	4	446	178,657	5	498	211,520	6	498	211,520	6
72	134	44,908	2	142	45,791	6	136	45,928	2	141	46,012	3	123	45,735	1	123	45,951	1	123	45,951	1
73	214	94,568	1	219	94,763	2	214	94,568	1	225	103,494	1	225	103,494	1	225	103,494	1	234	107,556	1
76	74,315	3,775,394	4	361,704	5,681,518	6	644,192	10,062,414	3	23,749	751,299	2	251	183,656	2	1,587	476,069	2	17,332	613,786	2
77	>1,019,702	> 20,000,000	1	> 973,871	> 20,000,000	4	>1,043,492	> 20,000,000	1	>1,055,068	> 20,000,000	1	>1,108,195	> 20,000,000	1	>1,055,933	> 20,000,000	1	257,001	4,729,912	1
78	64	4,913	1	64	4,913	1	64	4,913	1	64	4,913	1	64	4,913	1	64	4,913	1	64	4,913	1
79	122	15,527	2	132	16,868	5	125	15,530	3	127	16,086	2	127	13,114	2	124	12,869	2	124	12,869	2
80	165	26,943	1	173	30,199	3	167	26,945	2	176	26,309	1	176	26,309	1	516	48,571	1	104	21,495	1
81	2,662	42,445	1	5,020	90,840	3	1,183	113,961	1	269	109,570	1	875	111,033	1	19,152	270,782	1	4,625	140,321	1
82	86	33,445	2	6	33,423	3	95	35,987	3	114	52,160	3	117	45,014	3	120	45,896	3	117	44,995	3
83	80	5,631	1	84	5,635	3	80	5,631	1	108	6,856	1	108	6,856	1	108	6,856	1	108	6,856	1
84	106	7,938	1	177	7,534	3	161	6,882	1	107	6,612	1	108	7,818	1	110	7,929	1	110	7,929	1
	>3,646,938	>129,388,544		>4,728,569	>178,339,940		>5,864,460	>177,221,025		>5,179,478	>126,928,900		>1,686,065	> 70,566,483		>2,102,561	> 68,625,225		>4,102,768	>108,153,380	

Table 7.3: Pattern Overestimation, Varying s

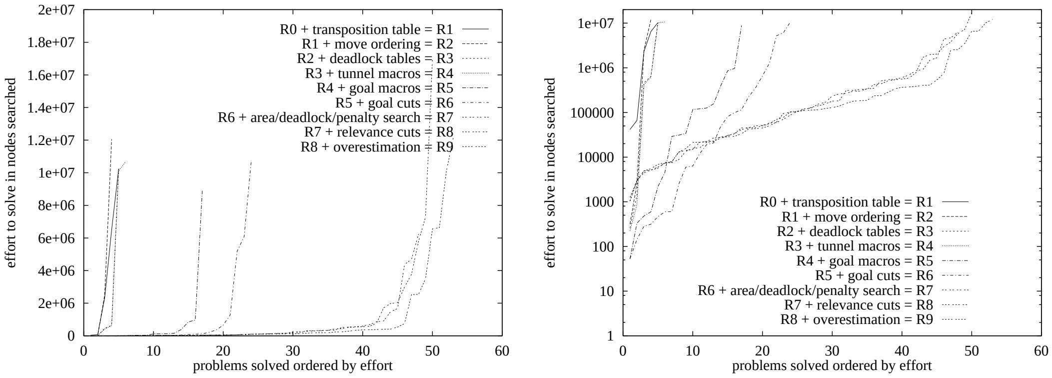


Figure 7.5: Adding Overestimation to *Rolling Stone* (Linear and Log Scale)

experiment are more conclusive, good values for s can be selected. It appears that the value of 1.8 is a good setting for s, allowing three more problems to be solved. Even though setting s to 2.0 can also solve 53 problems, it is inferior because the average number of top-level nodes is almost double that for $s = 1.8$. See Table 7.3 for the node numbers corresponding to Figures 7.3 and 7.4.

7.4.3 Summary

Figure 7.5 shows the effort diagram, now including the version of *Rolling Stone* with overestimation using maximum partial penalties and a scaling factor of $s = 1.8$. The improvement appears significant with about one order of magnitude savings in search-tree size.

There are a couple of interesting points about the data in Table 7.3. With relevance cuts, almost all problems, except #49, have smaller or insignificantly larger number of nodes. Problem #26, for example, drops from over 20 million nodes to just under 123,000. Other problems, like #23, #25, #36, #40, #54 and #76, also drop in node numbers significantly. While most searches with overestimation use more iterations to find a goal, the search for problem #26 uses less. The initial position is overestimated enough to allow the search to find a solution in fewer iterations. On average, the top-level and total nodes are reduced by roughly half, from 3.6 to 1.7 million and 129 to 71 million, respectively.

7.5 Conclusions and Open Problems

With respect to WIDA*, Sokoban is again proving to be a difficult domain. While in other domains scaling h allows at least the opportunity to trade off solution quality for search effort, it seems to only randomly shift the search effort in Sokoban. The quality of the lower-bound function is not good enough to indicate reliably when progress is made. Therefore, using depth as an indicator for progress has its pitfalls. Parts of the search tree that do not contain solutions are explored with more effort without the expected success.

Using knowledge that is readily available (the patterns matching in each position) to identify situations that are likely difficult was proven to be of greater value. *Rolling Stone* using this dynamic, knowledge-driven overestimation is able to solve three more problems.

When looking through Tables 7.2 and 7.3 one can see that *Rolling Stone* has found solutions to a total of 54 problems. Problem #77 can be solved when s is set to 2.2. In fact, *Rolling Stone* has solved 56 different problems with different combinations of s and w, but never with one version. A control function to set s and w according to features of the maze that will have to be identified could be of benefit. How to identify such features is an open problem. Especially in domains such as Sokoban, where the absence of a good heuristic function causes inefficient searches, discovering reliable, predictable features seems a daunting task.

Chapter 8

Single-Agent Search Enhancements

8.1 Introduction

The AI research community has developed an impressive suite of techniques for solving state-space problems. These techniques range from general-purpose domain-independent methods such as A*, to domain-specific enhancements, as we have seen in this thesis. There is a strong movement towards developing domain-independent methods to solve problems. While these approaches require minimal effort to specify a problem to be solved, the performance of these solvers is often limited, exceeding available resources on even simple problem instances. This requires the development of domain-dependent methods that exploit additional knowledge about the search space. These methods can greatly improve the efficiency of a search-based program, as measured in the size of the search tree needed to solve a problem instance.

Previous chapters reported on our attempts to solve Sokoban problems using an array of different techniques and search enhancements. This allowed 53 problems to be solved.[1] These results show the large gains achieved by dynamically discovering and applying knowledge in our program *Rolling Stone*. With each enhancement, reductions of search-tree sizes by several orders of magnitude are possible.

Analyzing all the additions made to *Rolling Stone* reveals that the most valuable search enhancements are based on search (both on-line and off-line) to improve the lower bound. In this chapter, we classify the search enhancements along several dimensions including their generality, computational model, completeness and admissibility. Not surprisingly, the more specific an enhancement is, the greater its impact on search performance.

When presented in the literature, single-agent search (usually IDA*) consists of a few lines of code. Most textbooks do not discuss search enhancements other than cycle detection. In reality, non-trivial single-agent search problems require more extensive programming (and possibly research) effort. For example, achieving high performance at solving sliding-tile puzzles requires enhancements such as cycle detection,

[1] Due to an oversight, we failed to detect problem # 30 as being solved until it was too late to include the numbers in this thesis. Recent experiments using Rapid Random Restart [GSK98] increased this number even further to 57.

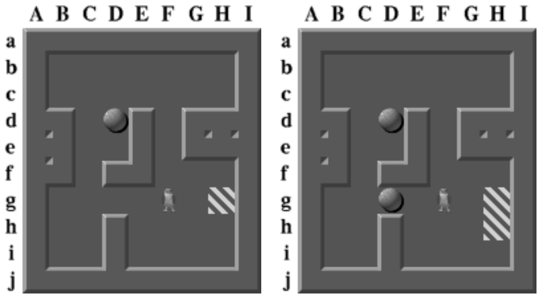


Figure 8.1: Two Simple Sokoban Problems

pattern databases, move ordering and enhanced lower-bound calculations [CS96]. In this chapter, we outline a new framework for high-performance single-agent search programs and propose a taxonomy of single-agent search enhancements.

8.2 Application-Independent Techniques

Ideally, applications should be specified with minimal effort and a "generic" solver would be used to compute the solutions. In small domains this is attainable (e.g., if it is easily enumerable). For more challenging domains, there have recently been a number of interesting attempts at domain-independent solvers (e.g., *blackbox* [KS96]). Before investing a lot of effort in developing a Sokoban-specific program, it is important to understand the capabilities of current AI tools. Hence, we include this information to illustrate the disparity between what application-independent problem solvers can achieve, compared to application-dependent techniques.

The Sokoban problems in Figure 8.1 [McD98] were given to the program *blackbox* to solve. *Blackbox* was one of the best programs at the AIPS'98 fastest planner competition. The first problem was solved within a few seconds and the second problem was solved in over an hour.

Clearly, domain-independent planners, like *blackbox*, have a long way to go if they are to solve the even simplest problem in the test suite. Hence, for this application domain, we have no choice but to pursue an application-dependent implementation.

Note also, that many of the domain-description languages used, such as STRIPS, often do not allow for efficient domain descriptions. While *Rolling Stone* can use simplifications, such as ignoring the exact position of the man, planners reading a STRIPS-like problem description have to deal with a much larger search space, because the man's position is encoded explicitly and cannot be handled efficiently.

8.3 Application-Dependent Techniques

Application-dependent techniques are not *per se* application dependent, in fact they can be applied to a variety of domains. We call them application (or domain) dependent because the knowledge they use applies to a particular domain.

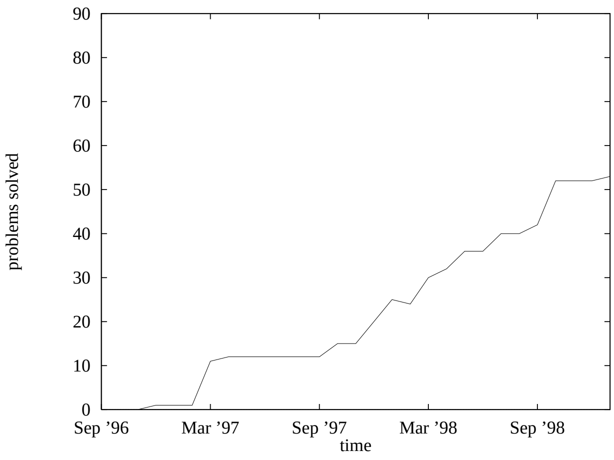


Figure 8.2: Number of Problems Solved Over Time

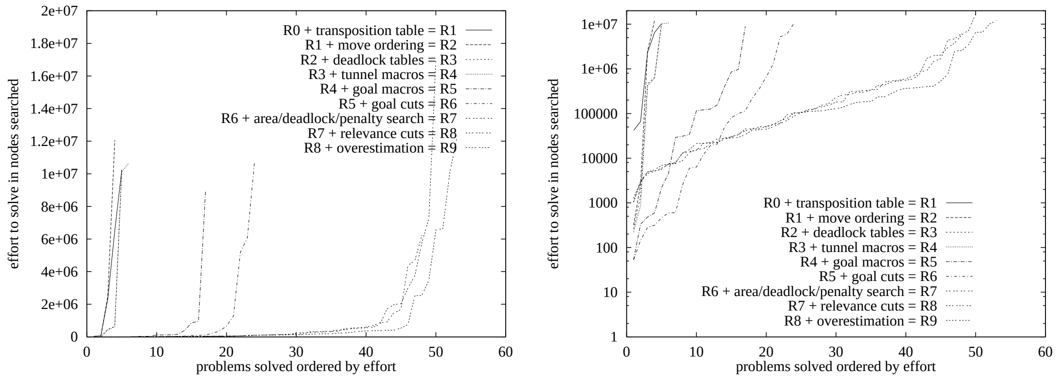


Figure 8.3: Effort Graph, Repeated (Linear and Log Scale)

The preceding chapters of this thesis show the power and limitations of application-dependent search enhancements. Their performance comes at a price: programming and research effort. Figure 8.2 shows how these results were achieved during 2.5 years of development time. The development effort equates to a full-time PhD student, a part-time professor, a full-time summer student (4 months), and feedback from many people. Additionally, a large number of machine cycles were used for tuning and debugging. It is interesting to note the occasional *decrease* in the number of problems solved, the result of (favorable) bugs being fixed. The long, slow, steady increase is indicative of the reality of building a large system. Progress is incremental and often painfully slow.

The large reductions in search-tree sizes that we have seen previously are not achievable with the current state-of-the-art domain-independent techniques. Unfortunately, if solutions to complex problems are required, application-dependent techniques are necessary.

The performance gap between the first and last versions of *Rolling Stone* in Fig-

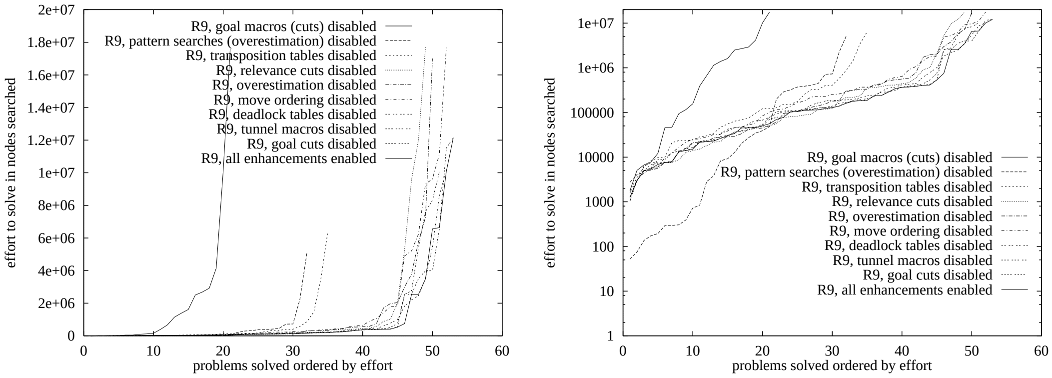


Figure 8.4: Turning One Enhancement Off (Linear and Log Scale)

ure 8.3 is astounding. For example, consider extrapolating the performance of *Rolling Stone* only with transposition tables (R1) so that it can solve the same number of problems (53) as the complete program (R9). 10^{50} (not a typo!) seems to be a reasonable lower bound on the difference in search-tree sizes.

As already discussed in Chapter 4, the results in Figure 8.3 may misrepresent the importance of each feature. Figure 8.4 shows the results of taking the full version of *Rolling Stone* (R9) and disabling one search enhancement at a time. The exact numbers can be found split over Tables 8.1 and 8.2. In the absence of a particular method, other search enhancements might compensate such that most of the solutions can still be found. But if the search-tree reductions of an enhancement are mostly unique, turning it off will reduce the total number of problems solved significantly.

While the lower-bound function alone cannot solve a single problem, neither can the complete system solve a single problem without the lower-bound function. This explains why the lower bound is never disabled in our tests. It is of paramount importance, without it no problem can be solved.

Figure 8.4 shows that turning off goal macros reduces the number of problems solved by 32, more than 50%! When turning off pattern searches, the number of solved problems drops by 21. Turning off transposition tables loses 18 problems. Besides the lower-bound function, these three enhancements are the most important ones for *Rolling Stone*; losing any one of them dramatically reduces the performance. Relevance cuts are responsible for 4 solutions and tunnel macros for 2. Turning off either move ordering or deadlock tables results in the loss of only one problem. Note that even though in Section 4.8 disabling goal cuts lost 7 problems, the full version (R9) still solves all problems, only with slightly larger node counts. Pattern searches, relevance cuts and/or overestimation are able to compensate for the loss of the goal cuts.

#	goal macros (cuts) disabled			pattern searches (overestimation) disabled			transposition tables disabled			relevance cuts disabled			all enabled		
	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns
1	111	1,798	3	52	52	2	55	1,752	3	55	1,267	3	55	1,267	3
2	129	7,775	1	194	194	2	80	9,729	1	82	7,532	1	80	7,530	1
3	9,975	45,553	1	305	305	2	101	23,614	1	106	15,188	1	94	14,095	1
4	21,469	665,602	3	392	392	1	187	63,527	1	187	50,369	1	187	50,369	1
5	811,912	4,145,913	8	20,213	20,213	3	287	121,360	2	218	36,778	1	153	33,755	1
6	9,163	115,558	1	174	174	3	111	9,922	1	85	5,504	1	84	5,503	1
7	207	12,177	1	8,295	8,295	5	336	28,439	1	391	13,984	2	338	14,832	2
8	738,055	2,921,408	4	450,161	450,161	6	540	867,357	5	371	465,263	5	315	409,714	5
9	2,859	390,770	4	34,514	34,514	8	824	327,033	1	172	80,434	1	1,591	385,084	4
10	> 99,661	> 20,000,000	3	> 20,000,000	> 20,000,000	2	> 25,983	> 20,000,000	1	19,119	12,029,738	5	2,920	2,539,524	3
11	> 734,353	> 20,000,000	12	> 20,000,000	> 20,000,000	16	> 32,628	> 20,000,000	8	6,553	2,076,276	12	4,058	2,527,286	12
12	> 307,056	> 20,000,000	8	> 20,000,000	> 20,000,000	10	> 252,493	> 20,000,000	1	> 1,674,420	> 20,000,000	1	951	372,264	8
17	> 2,515,393	> 20,000,000	12	9,182	9,182	7	> 919,122	> 20,000,000	9	2,698	30,287	9	2,158	30,242	9
19	> 910,691	> 20,000,000	10	> 20,000,000	> 20,000,000	8	> 31,982	> 20,000,000	6	> 59,322	> 20,000,000	8	14,178	6,631,475	10
21	> 7,398,297	> 20,000,000	3	286,982	286,982	10	2,762	577,354	10	17,221	884,501	3	573	113,042	3
23	> 584,364	> 20,000,000	12	> 20,000,000	> 20,000,000	3	> 169,507	> 20,000,000	7	31,868	5,264,113	12	23,337	6,555,398	12
25	> 2,440,537	> 20,000,000	7	> 20,000,000	> 20,000,000	5	619	399,815	6	901	409,941	7	683	366,035	7
26	> 2,240,971	> 20,000,000	7	> 20,000,000	> 20,000,000	15	> 3,219,787	> 20,000,000	13	> 4,285,048	> 20,000,000	7	380	122,997	7
33	> 863,437	> 20,000,000	2	> 20,000,000	> 20,000,000	7	> 65,650	> 20,000,000	1	2,155	378,220	1	604	283,926	1
34	32,227	2,500,435	2	2,234,289	2,234,289	9	17,069	6,376,898	3	513	358,480	3	9,746	749,787	2
36	> 194,690	> 20,000,000	6	> 20,000,000	> 20,000,000	6	2,167	1,461,524	6	259,772	9,570,145	6	18,338	12,150,606	7
38	22,278	93,959	1	27,259	27,259	9	9,348	174,488	1	2,012	38,114	1	10,473	160,176	1
40	> 86,192	> 20,000,000	8	> 20,000,000	> 20,000,000	7	> 27,370	> 20,000,000	6	43,639	17,686,156	8	16,725	10,086,547	9
43	341,553	2,681,897	8	385,869	385,869	8	> 39,775	> 20,000,000	8	10,477	1,025,451	8	2,225	535,148	8
45	> 462,451	> 20,000,000	2	> 20,000,000	> 20,000,000	7	5,898	3,479,354	5	398	216,082	1	602	404,217	2
49	> 3,421,140	> 20,000,000	6	320,669	320,669	13	> 2,128,496	> 20,000,000	9	> 4,294,023	> 20,000,000	9	441,638	3,486,905	9
51	99,047	1,388,288	2	2,819	2,819	1	277	34,152	1	137	8,825	1	256	21,491	1
53	> 2,164,029	> 20,000,000	1	> 20,000,000	> 20,000,000	1	157	23,416	1	159	22,310	1	157	22,308	1
54	> 1,382,259	> 20,000,000	4	> 20,000,000	> 20,000,000	3	344	46,619	4	224	40,708	3	269	45,332	3
55	> 3,408,250	> 20,000,000	1	136	136	1	97	3,490	1	97	2,993	1	97	2,993	1
56	65,549	156,228	7	> 20,000,000	> 20,000,000	5	2,893	75,141	5	2,415	87,905	7	911	55,865	7
57	> 3,485,817	> 20,000,000	1	5,070,525	5,070,525	5	252	127,478	2	239	122,033	1	209	128,282	1
58	> 6,848,326	> 20,000,000	3	> 20,000,000	> 20,000,000	1	> 26,375	> 20,000,000	2	783	439,792	3	231	138,838	3
59	> 2,468,760	> 20,000,000	7	> 20,000,000	> 20,000,000	6	> 855,045	> 20,000,000	8	1,105	590,482	4	602	337,905	4
60	259,279	1,138,911	1	12,683	12,683	3	> 3,148,019	> 20,000,000	1	1,375	55,725	1	18,100	114,642	1
61	> 1,377,709	> 20,000,000	8	> 20,000,000	> 20,000,000	6	> 132,563	> 20,000,000	2	302	76,588	8	299	77,555	8
62	> 1,549,298	> 20,000,000	5	3,812	3,812	5	184	95,696	5	195	82,351	4	180	69,728	4
63	> 989,550	> 20,000,000	1	715,579	715,579	3	584	322,947	3	483	217,033	1	473	237,196	1
64	> 158,253	> 20,000,000	1	369,870	369,870	10	348	406,986	1	191	178,401	1	193	186,508	1
65	> 387,680	> 20,000,000	1	293	293	5	144	29,983	1	156	21,739	1	165	23,004	1
67	> 2,539,252	> 20,000,000	1	> 20,000,000	> 20,000,000	10	524	152,897	5	302	115,122	1	298	104,356	1
68	> 6,935,468	> 20,000,000	9	> 20,000,000	> 20,000,000	6	> 1,467,936	> 20,000,000	10	334	235,911	1	324	236,157	1
70	> 476,971	> 20,000,000	5	> 20,000,000	> 20,000,000	1	430	252,302	5	480	118,716	4	446	178,657	5
72	> 785,374	> 20,000,000	1	727,780	727,780	5	148	69,042	5	123	45,735	1	123	45,735	1
73	> 1,524,435	> 20,000,000	1	408,462	408,462	3	232	151,776	1	212	90,906	1	225	103,494	1
76	2,208,039	17,583,377	2	> 20,000,000	> 20,000,000	5	> 36,784	> 20,000,000	4	1,044	409,335	3	251	183,656	2
78	139	5,058	1	75	75	1	64	6,017	1	64	4,451	1	64	4,913	1
79	591,019	1,612,679	2	723	723	5	155	24,317	5	133	12,788	2	127	13,114	2
80	> 3,395,043	> 20,000,000	1	842	842	1	205	55,016	1	118	18,004	1	176	26,309	1
81	> 5,572,202	> 20,000,000	1	48,302	48,302	4	> 2,215,418	> 20,000,000	2	31,962	360,177	1	875	111,033	1
82	287	46,526	3	39,043	39,043	5	140	121,397	5	104	48,197	3	117	45,014	3
83	174	6,694	1	297	297	1	138	21,032	1	96	7,440	1	108	6,856	1
84	2,127,005	10,109,624	1	199,985	199,985	4	106	8,260	1	97	6,127	1	108	7,818	1
	>75,048,385	>685,630,230		>431,379,776	>431,379,776		>14,842,739	>375,950,130		>10,754,736	>134,063,617		577,870	50,566,483	

Table 8.1: Turning One Enhancement Off (I)

#	overestimation disabled			tunnel macros disabled			move ordering disabled			deadlock tables disabled			goal cuts disabled			all enabled		
	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns	IDA* nodes	total nodes	itrns
1	50	1,042	2	65	1,598	3	114	2,731	3	56	1,272	3	55	1,267	3	55	1,267	3
2	80	7,530	1	86	8,831	1	124	5,352	1	80	7,598	1	80	7,530	1	80	7,530	1
3	87	12,902	1	94	14,095	1	149	25,029	1	105	16,153	1	94	14,095	1	94	14,095	1
4	187	50,369	1	243	67,490	1	406	83,627	1	193	46,980	1	187	50,369	1	187	50,369	1
5	202	43,298	2	153	33,755	1	131	21,709	1	165	34,731	1	153	33,755	1	153	33,755	1
6	84	5,118	1	87	5,723	1	128	6,019	1	86	7,746	1	181	7,232	1	84	5,503	1
7	1,392	28,460	2	118	15,996	3	174	9,785	2	109	15,962	3	1,133	20,776	2	338	14,832	2
8	291	311,609	3	315	409,714	5	491	389,334	5	238	201,910	2	325	427,271	5	315	409,714	5
9	1,884	435,388	5	348	146,380	1	4,728	1,070,275	6	2,286	441,909	5	1,591	385,084	4	1,591	385,084	4
10	1,810	1,713,429	1	2,920	2,539,524	3	> 27,300	> 20,000,000	3	1,782	1,815,042	3	2,920	2,539,524	3	2,920	2,539,524	3
11	5,679	2,994,297	11	5,161	2,764,259	16	3,813	1,840,154	12	3,584	2,203,618	13	4,071	2,527,299	12	4,058	2,527,286	12
12	4,912	559,184	8	951	372,264	8	2,646	603,990	1	1,066	369,286	8	951	372,264	8	951	372,264	8
17	2,038	29,116	7	2,368	65,480	9	2,946	32,065	9	2,264	30,524	9	2,158	30,242	9	2,158	30,242	9
19	16,606	7,269,595	9	16,869	8,300,723	9	21,411	9,304,716	11	12,033	5,452,990	10	6,710	3,514,503	10	14,178	6,631,475	10
21	1,177	179,734	7	1,843	313,507	4	599	138,481	4	634	122,882	7	576	115,755	3	573	113,042	3
23	> 59,498	> 20,000,000	7	> 63,547	>20,000,000	12	21,295	5,202,990	11	35,594	8,603,866	9	23,337	6,555,398	12	23,337	6,555,398	12
25	21,536	5,784,086	7	723	385,622	7	965	424,750	5	1,977	1,006,404	11	709	366,069	7	683	366,035	7
26	>2,125,116	> 20,000,000	9	503	119,343	7	126,026	1,787,791	5	657	130,246	7	752	128,806	7	380	122,997	7
33	2,765	586,684	3	604	283,926	1	3,257	227,672	1	603	291,506	1	604	283,926	1	604	283,926	1
34	11,431	1,981,993	3	9,746	749,787	2	351,818	9,615,236	2	11,390	780,578	2	93,838	3,978,637	2	9,746	749,787	2
36	> 23,467	> 20,000,000	5	> 41,993	>20,000,000	8	278,538	17,666,104	6	5,527	2,465,788	6	18,338	12,150,606	7	18,338	12,150,606	7
38	7,011	154,969	6	11,228	186,979	1	2,789	28,559	1	11,678	166,511	1	14,021	165,930	1	10,473	160,176	1
40	23,274	17,004,253	7	17,772	10,048,532	9	20,683	11,135,404	9	20,623	10,220,101	9	19,029	11,508,074	9	16,725	10,086,547	9
43	1,729	421,483	7	2,225	535,148	8	2,357	549,141	8	2,604	612,082	8	2,225	535,148	8	2,225	535,148	8
45	339	181,566	2	697	434,293	2	986	637,176	2	703	426,790	1	602	404,217	2	602	404,217	2
49	53,113	327,643	9	774,682	5,413,505	9	72,419	369,491	9	452,534	3,524,020	9	510,730	3,989,973	9	441,638	3,486,905	9
51	256	21,491	1	264	23,774	1	1,207	51,895	1	260	23,377	1	406	22,269	1	256	21,491	1
53	157	22,308	1	171	22,345	1	309	45,633	1	157	21,993	1	157	22,308	1	157	22,308	1
54	163,757	2,031,577	2	>1,503,864	>20,000,000	1	36,390	901,024	2	>1,655,503	>20,000,000	1	269	45,332	3	269	45,332	3
55	97	2,993	1	97	2,993	1	322	4,084	1	96	2,927	1	97	2,993	1	97	2,993	1
56	377	61,189	3	911	55,865	7	36,677	252,888	8	961	54,227	7	627	84,730	11	911	55,865	7
57	234	114,416	2	217	108,909	1	887	222,032	2	235	111,068	1	209	128,282	1	209	128,282	1
58	211	130,474	2	275	180,500	3	1,584	259,662	3	240	138,725	3	231	138,838	3	231	138,838	3
59	1,420	775,753	4	778	214,008	7	1,465	292,744	7	7,535	416,503	7	602	337,905	4	602	337,905	4
60	160	27,386	1	34,700	287,151	1	280	25,623	1	18,320	119,642	1	32,916	126,160	1	18,100	114,642	1
61	309	105,411	5	493	164,727	7	1,337	179,764	8	334	77,926	8	299	77,555	8	299	77,555	8
62	195	101,934	3	216	78,095	4	253	116,583	5	190	93,424	5	180	69,728	4	180	69,728	4
63	703	312,546	1	473	237,196	1	>1,892,813	> 20,000,000	1	698	349,299	3	473	237,196	1	473	237,196	1
64	405	332,402	4	209	190,724	1	1,859	1,425,833	1	230	193,795	1	193	186,508	1	193	186,508	1
65	196	21,442	2	181	24,500	1	> 687,199	> 20,000,000	1	167	22,363	1	165	23,004	1	165	23,004	1
67	12,669	512,488	6	322	108,248	1	17,495	311,723	1	300	101,536	1	298	104,356	1	298	104,356	1
68	1,953	538,509	6	324	236,157	1	1,140	228,388	4	556	228,978	6	324	236,157	1	324	236,157	1
70	431	140,765	3	474	188,237	5	1,421	164,454	4	437	160,238	5	446	178,657	5	446	178,657	5
72	134	44,908	2	123	45,735	1	243	58,255	1	150	39,160	5	123	45,735	1	123	45,735	1
73	214	94,568	1	225	103,494	1	408	119,498	1	229	105,679	1	225	103,494	1	225	103,494	1
76	74,315	3,775,394	4	257	193,943	2	2,854	369,255	2	255	185,001	2	251	183,656	2	251	183,656	2
78	64	4,913	1	70	4,934	1	90	5,017	1	65	6,341	1	64	4,913	1	64	4,913	1
79	122	15,527	2	132	13,748	2	403	38,396	2	130	13,296	2	219	13,286	2	127	13,114	2
80	165	26,943	1	285	42,338	1	302	32,638	1	176	28,967	1	176	26,309	1	176	26,309	1
81	2,662	42,445	1	333	109,338	1	360	101,241	1	925	114,593	1	2,427	113,291	1	875	111,033	1
82	86	33,445	2	118	45,197	3	184	48,111	3	133	46,823	2	117	45,014	3	117	45,014	3
83	80	5,631	1	159	12,350	1	303	12,381	1	108	7,472	1	108	6,856	1	108	6,856	1
84	106	7,938	1	121	8,816	1	219	16,615	1	109	8,004	1	108	7,818	1	108	7,818	1
	>2,627,236	>109,388,544		>2,501,133	>95,919,796		>3,634,297	>126,461,318		>2,257,070	>61,667,852		747,080	52,686,100		577,870	50,566,483	

Table 8.2: Turning One Enhancement Off (II)

#	IDA*	IDA* + PIDA*	#	IDA*	IDA* + PIDA*
1	14	53	32	4	9
2	786	830	33	12	91
3	9	22	34	37	129
4	14	38	35	25	107
5	23	58	36	15	71
6	10	30	37	82	155
7	6	30	38	43	183
8	19	50	39	28	65
9	6	13	40	20	119
10	7	14	41	39	139
11	15	26	42	13	67
12	10	20	43	34	95
13	5	20	44	15	45
14	7	16	45	33	108
15	16	27	46	49	157
16	55	125	47	22	45
17	16	26	48	76	131
18	15	57	49	18	36
19	22	125	50	68	220
20	20	40	51	256	848
21	22	72	52	79	232
22	30	52	53	11	32
23	11	54	54	771	1,938
24	14	57	55	318	531
25	5	22	56	170	290
26	63	148	57	135,255	342,785
27	4	9	58	> 9,486,886	> 20,000,000
28	16	48	59	27	76
29	41	112	60	2,866	4,460
30	23	49	61	> 11,044,404	> 20,000,000
31	19	41			
				> 20,672,999	> 40,355,448

Table 8.3: The Kids Problems

8.4 Test Sets and Search Effort

Using one test set to tune *and* measure progress with will necessarily lead to overfitting of the program to the test set. We have tested our program *Rolling Stone* on a set of 61 simple problems to verify that it is at least not geared towards large problems. *Rolling Stone* solves 59 of the 61 problems. The two it cannot solve with 20 million nodes of search effort require parking in the goal area, a concept the program does not know about. Appendix B shows the complete test set.

The limit of 20 million nodes in our experiments is arbitrarily chosen. However, Figures 8.3 and 8.4 show that decreasing the search effort even by 2 orders of magnitude would lead to almost the same qualitative results. To see how close we are to solving more problems with our 20 million node effort limit, we conducted an experiment with the best version of *Rolling Stone*, allowing for 1 billion nodes of search effort. One more problem could be solved: #24 uses 591,287,416 nodes. This confirms the exponential nature of the domain.

8.5 Knowledge Taxonomy

Several different ways of classifying the domain-specific knowledge used to solve Sokoban problems can be identified:

Generality: Classify based on how general the knowledge is: *domain* (e.g., Sokoban), *instance* (a particular Sokoban problem), and *subtree* (within a Sokoban search).

Computation: Differentiate how the knowledge was obtained: *static* (such as advice from a human expert) and *dynamic* (gleaned from a search).

Admissibility/Completeness: Knowledge can be: *admissible* (preserve optimality in a solution) or *non-admissible*. Non-admissible knowledge can either preserve *completeness* of the algorithm or render it *incomplete*. Admissible knowledge is necessarily complete.

Figure 8.5 summarizes the search enhancements used in *Rolling Stone*. Other enhancements from the literature could easily be added into spaces that are still blank, e.g. perimeter databases [Man95] (dynamic, admissible, instance). Note that some of the enhancement classifications are fixed by the type of the enhancement. For example, any type of heuristic (unsafe) forward pruning is incomplete by definition, and move ordering always preserves admissibility. For some enhancements, the properties depend on the implementation. For example, overestimation techniques can be static or dynamic; goal macros can be admissible or non-admissible; pattern databases can be domain-based or instance-based.

It is interesting to note that, apart from the lower-bound function itself, the three most important program enhancements in terms of program performance are all dynamic (search-based) and instance/subtree specific. The static enhancements, while of value, turn out to be of less importance. Static knowledge is usually rigid

Classification		Domain	Instance	Subtree
Static	admissible	lower bound	tunnel macros	move ordering
	complete			
	incomplete		relevance cuts	goal cuts
Dynamic	admissible	deadlock tables		pattern searches
				transposition table
	complete			overestimation
	incomplete		goal macros	

Figure 8.5: Taxonomy of Search Enhancements in Sokoban

and does not include the myriad of exceptions that search-based methods can uncover and react to.

8.6 Control Functions

There is another type of application-dependent knowledge that is critical to performance, but receives scant attention in the literature. *Control functions* are intrinsic parts of efficient search programs, controlling when to use or not use a search enhancement. In *Rolling Stone* numerous control functions are used to improve the search efficiency. Some examples include:

Transposition Table: A fixed-size transposition table can only hold so much information. Control knowledge is needed to decide when new information should replace older information in the table. Also, when reading from the table, control information can decide whether or not the benefits of the lookup justify the cost. For example, search applications may not look up table entries close to the leaf nodes.

Goal Macros: If a goal area has too few goal squares, then goal macros are disabled. With a small number of goals or too many entrances, the search will likely not need macro moves, and the potential savings are not worth the risk of eliminating possible solutions.

Pattern Searches: Pattern searches are executed only when a non-trivial heuristic function indicates the likelihood of a penalty being present. Executing a pattern search is expensive, so this overhead should be introduced only when it is likely to be cost effective. Control functions are also used to stop a pattern search when success appears unlikely.

Implementing a search enhancement is often only one part of the programming effort. Implementing and tuning its control function(s) can be significantly more time consuming and more critical to performance. We estimate that whereas the search enhancements take about 90% of the coding effort and the control functions only 10%, the reverse distribution applies to the amount of tuning effort needed and machine cycles consumed.

A clear separation between the search enhancements and their respective control functions (task and control knowledge) can help the tuning effort. For example, while the goal macro creation only considers which order the stones should be placed into the goal area, the control function can determine if goal macros should be created at all. Both tuning efforts have very different objectives: one is search efficiency, the other risk minimization. Separating the two seems natural and convenient.

However, this split is not solving the general problem we are facing when tuning. As shown in the NFL discussion, when specializing an algorithm (by tuning or any other measure, such as search enhancements in general) we are trading off performance of the algorithm for one kind of problem against the performance for other kinds of problems. When tuning parameters using performance on a test suite as a measure of improvement, we are implicitly adapting the algorithm to the properties exemplified in the test suite. For our 90 problems, this is most certainly true. Humans composed the problems, using concepts such as rooms and hallways, structuring the problems in a very specific way. Goal macros are a good example how we exploited one of these properties: goals are often together in lumps in a designated area. Random instances would defy goal macros. Control functions are an attempt to recognize these situations and turn goal macros off.

8.7 Single-Agent Search Framework

Figure 8.6 illustrates the basic IDA* routine, with our enhancements included (in *italics*). This routine is specific to *Rolling Stone*, but could be written in more general terms. It does not include a number of well-known single-agent search enhancements available in the literature. Control functions are indicated by parameters to search enhancement routines. In practice, some of these functions are implemented as simple *if* statements controlling access to the enhancement code.

Examining the code in Figure 8.6, one realizes that there are really only four types of search enhancements:

1. Modifying the lower bound (as indicated by the updates to *lb*). This can take two forms: optimally increasing the bound (e.g. using patterns) which reduces the distance to search, or non-optimally (using overestimation) which redistributes where the search effort is concentrated.

```
IDA*() {

  /** Compute the best possible lower bound **/
  lb = ComputeLowerBound();
  lb += UsePatterns();      /** Match Patterns **/
  lb += UseDeadlockTable();
  lb += UseOverestimate( CntrlOverestimate() );
  IF( cutoff ) RETURN;

  /** Preprocess **/
  lb += ReadTransTable();
  IF( cutoff ) RETURN;
  PatternSearch( CntrlPatternSearch() );
  lb += UsePatterns();
  IF( cutoff ) RETURN;

  /** Generate searchable moves **/
  movelist = GenerateMoves();
  RemoveDeadMoves( movelist );
  IdentifyMacros( movelist );
  OrderMoves( movelist );

  FOREACH( move ) {
    IF( Irrelevant( move, CntrlIrrelevant() )) NEXT;
    solution = IDA*();
    IF( solution ) RETURN;
    IF( GoalCut() ) BREAK;
    UpdateLowerBound();   /** Use New Patterns **/
    IF( cutoff ) RETURN;
  }

  /** Post-process **/
  SaveTransTable( CntrlTransTable() );

  RETURN;
}
```

Figure 8.6: Enhanced IDA*

```
FOREACH( domain ) {

  /** Preprocess **/
  BuildDeadlockTable( CntrlDeadlockTable() );

  FOREACH( instance ) {

    /** Preprocess **/
    FindTunnelMacros();
    FindGoalMacros( CntrlGoalMacros() );

    WHILE( NOT solved ) {
      SetSearchParamaters();
      IDA*();
    }

    /** Postprocess **/
    SavePatterns( CntrlSavingPatterns() );
  }
}
```

Figure 8.7: Preprocessing Hierarchy

2. Removing branches unlikely to add additional information to the search (the *next* and *break* statements in the *for* loop). This forward pruning can result in large reductions in the search tree, at the expense of possibly removing solutions.

3. Collapsing the tree height by replacing a sequence of moves with one move (for example, macros).

4. Move ordering allows for savings in the last iteration by exploring promising lines first.

Some of the search enhancements involve computations outside of the search. Figure 8.7 shows where the pre-search processing occurs at the domain and instance levels. Off-line computation of pattern databases or preprocessing of problem instances are powerful techniques that receive scant attention in the literature (chess endgame databases are a notable exception). Yet these techniques are an important step towards the automation of knowledge discovery and machine learning. Preprocessing is involved in many of the most valuable enhancements that are used in *Rolling Stone*.

Similar issues occur with other search algorithms. For example, although it takes only a few lines to specify the alpha-beta algorithm, the *Deep Blue* chess program's search procedure includes numerous enhancements (many similar in spirit to those used in *Rolling Stone*) that cumulatively reduce the search-tree size by several orders of magnitude. If nothing else, the *Deep Blue* result demonstrated the degree of engineering required to build high-performance search-based systems.

8.8 Conclusions

This chapter summarizes our experiences working with Sokoban. In contrast to the simplicity of the basic IDA* formulation, building a high-performance single-agent searcher can be a complex task that combines both research and engineering. Application-dependent knowledge, specifically that obtained using search, can result in an orders-of-magnitude improvement in search efficiency. This can be achieved through a judicious combination of several search enhancements. Control functions are overlooked in the literature, yet are critical to performance. They represent a significant portion of the program development time and most of the program experimentation resources.

Domain-independent tools offer a quick programming solution when compared to the effort required to develop domain-dependent applications. However, with current AI tools, performance is commensurate with effort. Domain-dependent solutions can be vastly superior in performance. The trade-off between programming effort and performance is the critical design decision that needs to be made.

Chapter 9

Conclusions and Future Work

Research into single-agent search methods has been dominated by relatively simple domains. Domains, such as the 15-puzzle or Rubik's Cube, have relatively small search-space complexities and/or decision complexities. The conclusions from the research in these domains have simplified our view of single-agent search. Often, implicit assumptions are made for certain methods to work. Well-behaved search spaces with reversible moves and relatively small branching factors and search depths are usually assumed. The availability of high-quality, low-cost lower-bound estimators is another one of these assumptions. Naturally, one has to be careful about conclusions drawn from domains having such nice properties.

In this thesis, we have seen an instance of a problem domain that defies the traditional approaches and requires more sophisticated methods. Sokoban has not just a large search space, but also exhibits the challenging search-space property of non-reversible moves which lead to deadlock configurations. Furthermore, an efficient and effective lower-bound function remains elusive. Many or even all the implicitly assumed preconditions of the text-book approaches are violated and the state-of-the-art methods fail.

This thesis shows how to tackle this challenge and makes significant progress in solving non-trivial problem configurations for Sokoban. New search enhancements are introduced. The most successful of them use small specialized searches to discover knowledge that can be used to improve the efficiency of the main search. Static, off-line searches producing goal macros show considerable improvements in search efficiency. However, dynamic, on-line pattern searches gather knowledge that leads to more significant reductions in search-tree sizes. It appears that searches, and dynamic searches in particular, can glean the information that is needed to break the complexity barrier build up by the combinatorial explosion characterizing these challenging domains. Other enhancements suggested here, such as relevance cuts and the pattern-driven overestimation, indicate that further considerable progress is possible.

An interesting observation made in this thesis is that the most powerful of search enhancements are closely linked to specific knowledge about the problem instances, or even specific problem configurations. Examples are:

- Transposition table entries store search results about specific states.

- Penalty patterns containing information about sets of states of one problem instance (with these patterns present) are results of searches.

- Goal macros are found by off-line searches and represent the knowledge of how to solve the subproblem of goal-packing in one instance.

We believe that this is no coincidence. While generalized and broadly applicable heuristics can help to give the search a general direction, it cannot possibly capture the subtleties of complex domains. Exceptions and special cases make these problems difficult and challenging and they have to be found by the search on an instance-by-instance basis. For puzzle games such as Sokoban, this wealth of intricate details is what draws humans and keeps them coming back. For the practice, this wealth is what characterizes many of the challenging real-world problems we are interested in solving. After all, if general guidelines or rules would apply, we would probably not perceive these problems as hard.

Even though Sokoban is primarily used as our research domain here, the methods and enhancements suggested as well as the lessons learned are largely domain independent and carry therefore over to other domains.

Short from excusing ourselves for picking Sokoban as an experimental testbed, we would like to point out that it could be a rich and fertile ground for many subfields of AI. Even though we have made considerable progress in this domain using advanced search methods, search alone is not going to be sufficient to solve the toughest of the Sokoban problems.

A Sokoban solver could benefit from any of the following areas of AI:

- Reasoning in all its different forms (automated, case-based, probabilistic, geometric and spatial,...) could help to decompose Sokoban instances into subproblems and, taking all the interactions of the subproblems into account, reassemble the solution for the complete problem.

- Belief revision must certainly play a role for the dynamic discovery of subsolution interactions. As new, possibly conflicting facts (interactions, partial solutions, constraints), are discovered they have to be integrated into the current knowledge base.

- Case-based reasoning could help to adapt solutions from similar instances solved in the past to new problems currently at hand.

- Knowledge acquisition and representation can help to tackle one of the fundamental problems of AI, of how to represent and store all the knowledge efficiently. As we have seen, this becomes an important problem.

- Planning can help to direct the search by providing it with the global context of local actions to assist in critical decisions like forward pruning and move ordering.

However, these areas can also benefit from Sokoban! Sokoban offers a non-trivial test bed for many techniques from different subfields of AI.

There are many more search related challenges and open questions left to explore in the domain of Sokoban. Search methods can most certainly be improved significantly. Some of the immediate issues that come to mind are: Can relevance cuts benefit from dynamically accumulated knowledge? Can move ordering be improved with additional knowledge? Are there better heuristics to decide which stone to include next in a pattern search? What could that knowledge be and how could it be collected?

However, a much more fruitful question to explore is probably how to use the methods developed for Sokoban in other domains. Different domains can provide different conditions and properties which these methods can be subjected to. The necessary generalizations can yield interesting new insights into why and how certain methods work for different application domains.

Yet another step is to try to use the methods developed here (and of course elsewhere) in a domain-independent way. It is fairly straightforward for some of the simpler search enhancements, like transposition tables, to be instantiated for a new domain. Especially transposition tables could also be turned off, when simple statistical tests about hit rates show that the savings do not justify their use. But, how can other, more complex methods be automatically instantiated like that? How can the knowledge needed for these domain-dependent search enhancements be automatically extracted? More to the point: How can we invoke a method, instead of a scientist? With the example of pattern searches we have shown that we can identify necessary conditions for the use of search enhancements. Can these conditions be tested automatically and, depending on the result of the test, search enhancements be enabled or disabled, or even adjusted? Are domain descriptions the source of most of the necessary information? Or would example searches reveal certain properties of the search space (of course assuming we are dealing with a well behaved, predictable domain)? Or both?

Humans are incredibly apt in adapting their problem solving methods. They do this on many different levels, such as for different domains, as well as for different instances of the same domain, and even for different phases of the solution of one problem instance. Humans are able to recognize when they are not making any progress and they can change their solution strategies. What are the next steps towards creating an artificial entity with such capabilities?

Bibliography

[All88] V. Allis. A knowledge-based approach of connect-four. the game is solved: White wins. Master's thesis, Free University, Amsterdam, The Netherlands, 1988.

[All94] V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.

[AVAD75] G. Adelson-Velskiy, V. Arlazarov, and M. Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6(4):361–371, 1975.

[BG98] B. Bonet and H. Geffner. Hsp: Heuristic search planner. In *AIPS-98 Planning Competition*, June 1998.

[BG99] B. Bonet and H. Geffner. Planning as heuristic search: New results, May 1999. http://www.ldc.usb.ve/˜hector/reports/hspr.ps.

[Bis92] R. Bisiani. Search, beam. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1467–1468. Wiley-Interscience, New York, 1992.

[BPSS99] D. Billings, L. Pena, J. Schaeffer, and D. Szafron. Using probabilistic knowledge and simulation to play poker. In *AAAI*, 1999.

[Bre98] D. Breuker. *Memory versus Search in Games*. PhD thesis, University of Maastricht, Computer Science Department, Maasticht, 1998.

[Bur97] M. Buro. The othello match of the year: Takeshi murakami vs. logistello. *ICCA Journal*, 20(3):189–193, 1997.

[Caz99] T. Cazenave. Generation of patterns with external conditions for the game of Go. In *Advances in Compter Chess 9*, 1999. to appear.

[CKW91] P. Cheeseman, B. Kanefsky, and Taylor W.M. Where the really hard problems are. In *IJCAI*, pages 331–337, 1991.

[CS94] J. Culberson and J. Schaeffer. Efficiently searching the 15-puzzle. Technical Report TR94–08, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1994. ftp.cs.ualberta.ca/pub/TechReports/1994/TR94–08.

[CS96] J. Culberson and J. Schaeffer. Searching with pattern databases. In G. McCalla, editor, *Advances in Artificial Intelligence*, pages 402–416. Springer-Verlag, 1996.

[CS98] J. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[Cul96] J. Culberson. On the futility of blind search. Technical Report TR96–18, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1996. ftp.cs.ualberta.ca/pub/TechReports/1996/TR96–18.

[Cul97] J. Culberson. Sokoban is PSPACE-complete. Technical Report TR97–02, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1997. ftp.cs.ualberta.ca/pub/TechReports/1997/TR97–02.

[dee99] deepgreen. Personal communications, March 1999.

[DZ95] D. Dor and U. Zwick. SOKOBAN and other motion planning problems, 1995. http://www.math.tau.ac.il/˜ddorit.

[Ede98] S. Edelkamp. Personal communications, July 1998.

[EK98] S. Edelkamp and R.E. Korf. The branching factor of regular search spaces. In *AAAI*, pages 299–304, 1998.

[FMS+89] A. Fiat, S. Moses, A. Shamir, I. Shimshoni, and G. Tardos. Planning and learning in permutation groups. In *Proceedings of the 30th A.C.M. Foundations of Computer Science Conference (FOCS)*, pages 274–279, 1989.

[Gas79] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[Gas94] R. Gasser. *Harnessing computational resources for efficient exhaustive search.* PhD thesis, ETH Zürich, Switzerland, 1994.

[GC90] G. Goetsch and M.S. Campbell. Experiments with the null-move heuristic. In T.A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 159–181, New York, 1990. Springer-Verlag.

[Gin93a] M. Ginsberg. Dynamic backtracking. *Jounal of Artificial Intelligence Research*, 1:25–46, 1993.

[Gin93b] M. Ginsberg. *Essentials in Artificial Intelligence.* Morgan Kaufman Publishers, San Francisco, 1993.

[Gin96] M. Ginsberg. Partition search. In *AAAI*, pages 228–233, 1996.

[Gin99] M. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *IJCAI*, Stockholm, 1999. To appear. UPDATE.

[GSK98] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *AAAI*, pages 431–437, 1998.

[Hik99] T. Hikita. Sokoban, January 1999. Personal communications: Translation of Sokoban article [UNH97] into English.

[Hir98] M. Hiramatsu. Personal communications, December 1998.

[HMY92] O. Hannson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissable heuristics. *Information Sciences*, 63(3):207–227, 1992.

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HPRA96] R. Holte, M. Perez, Zimmer R., and MacDonald A. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI*, pages 530–535, 1996.

[Iba89] G.A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–318, 1989.

[Jam93] M.L. James. Optimal tunneling: A heuristic for learning macro operators. Master's thesis, University of Calgary, Alberta, Canada, May 1993.

[JS97] A. Junghanns and J. Schaeffer. Sokoban: A challenging single-agent search problem. In *IJCAI*, pages 27–36, Nagoya, Japan, August 1997. At the Workshop: Using Games as an Experimental Testbed for AI Research.

[JS98a] A. Junghanns and J. Schaeffer. Relevance cuts: Localizing the search. In *The First International Conference on Computers and Games*, pages 1–13, Tsukuba, Japan, 1998. Also in: *Lecture Notes in Computing Science*, Springer Verlag.

[JS98b] A. Junghanns and J. Schaeffer. Single-agent search in the presence of deadlock. In *AAAI*, pages 419–424, Madison/WI, USA, July 1998.

[JS98c] A. Junghanns and J. Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In R. Mercer and E. Neufeld, editors, *Advances in Artificial Intelligence*, pages 1–15. Springer Verlag, 1998.

[JS99a] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *IJCAI*, pages 570–575, 1999.

[JS99b] A. Junghanns and J. Schaeffer. Sokoban: Improving the search with relevance cuts. *Journal of Theoretical Computing Science*, 1999. To appear.

[KK96] G. Kainz and H. Kaindl. Dynamic improvements of heuristic evaluations during search. In *AAAI*, pages 311–317, 1996.

[KK97] H. Kaindl and G. Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[Kle67] M. Klein. A primal method for minimal cost flows. *Management Science*, 14:205–220, 1967.

[Kor85a] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[Kor85b] R.E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[Kor93] R.E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, July 1993.

[Kor96] R.E. Korf. Finding optimal solutions to the twenty-four puzzle. In *AAAI*, pages 1202–1207, 1996.

[Kor97] R.E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI*, pages 700–705, 1997.

[KR98] R.E. Korf and M. Reid. Complexity analysis of admissible heuristic search. In *AAAI*, pages 305–310, 1998.

[KS96] H. Kautz and B. Selman. Pushing the envelope: planning, propositional logic and stochastic search. In *AAAI*, pages 1194–1201, 1996.

[Kuh55] H.W. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, pages 83–98, 1955.

[LW66] E.L. Lawler and D. Woods. Branch-and-bound methods: A survey. *Operations Research*, 14, 1966.

[Man95] G. Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75:347–360, 1995.

[McD98] Drew McDermott. Using regression-match graphs to control search in planning, 1998. Unpublished manuscript.

[Min88] S. Minton. Quantitative results concerning the utility of explanation-based learning. In *AAAI*, pages 564–569, 1988.

[Mye97] A. Myers. Personal communications, March 1997.

[New96] M. Newborn. *Kasparov versus Deep Blue. Computer Chess Comes of Age*. Springer-Verlag, New York, 1996.

[Nil80] N. Nilsson. *Principles in Artificial Intelligence*. Morgan Kaufman Publisher, Inc., Tioga, Palo Alto, CA, 1980.

[Poh71] I. Pohl. *Bi-directional Search*, pages 127–140. Edinburgh University Press, Edinburgh, 1971.

[Pri93] A.E. Prieditis. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–141, 1993.

[PSPdB96] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1–2):255–293, November 1996.

[Rei93] A. Reinefeld. Complete solution of the eight-puzzle and the benefit of node ordering in ida*. In *IJCAI*, pages 248–253, 1993.

[RKK91] V. Rao, V. Kumar, and R.E. Korf. Depth-first vs. best-first search. In *AAAI*, pages 434–440, 1991.

[RM94] A. Reinefeld and T.A. Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, July 1994.

[SA77] D. Slate and L. Atkin. Chess 4.5 — The Northwestern University chess program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118, New York, 1977. Springer-Verlag.

[Sch67] P.D.A. Schofield. *Complete solution of the eight puzzle*, pages 125–133. American Elsevier, New York, 1967.

[Sch86] J. Schaeffer. *Experiments in Search and Knowledge*. PhD thesis, University of Waterloo, Canada, 1986.

[Sch97] J. Schaeffer. *One Jump Ahead*. Springer Verlag, New York, 1997.

[SKC94] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *AAAI*, pages 337–343, 1994.

[SLLB96] J. Schaeffer, R. Lake, P. Lu, and Martin Bryant. Chinook: The man-machine world checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[SS77] R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[Tes95] G. Tesauro. Temporal difference learning and td-gammon. *CACM*, 38(3):58–68, March 1995.

[TK93] L. Taylor and R.E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI*, pages 756–761, 1993.

[UNH97] A. Ueno, K. Nakayama, and T. Hikita. Sokoban. *bit, special issue "Game programming"*, pages 158–172, 1997.

[Wil88] G. Wilfong. Motion planning in the presence of movable obstacles. In *4th ACM Symposium on Computational Geometry*, pages 279–288, 1988.

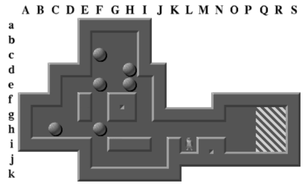
[Win92] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, 1992.

[WM96] D.H. Wolpert and W.G. Macready. No free lunch theorems for search. Technical Report SFI–TR–95–02–010, The Santa Fee Institute, Santa Fe, New Mexico, 1996. ftp.santafe.edu/pub/wgm/.

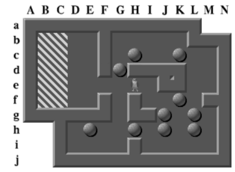
[Zha98] W. Zhang. Complete anytime beam search. In *AAAI*, pages 425–430, Madison/WI, USA, July 1998.

Appendix A

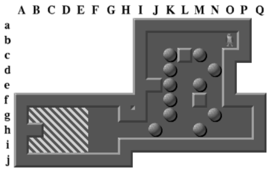
The 90 Problem Test Suite



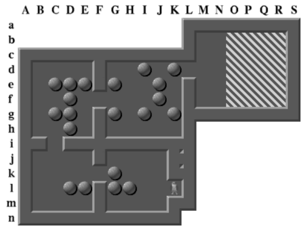
Problem #1



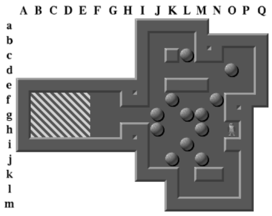
Problem #2



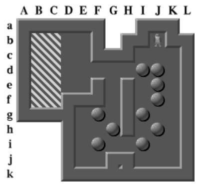
Problem #3



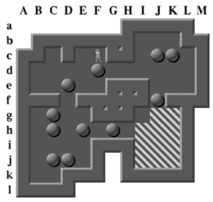
Problem #4



Problem #5

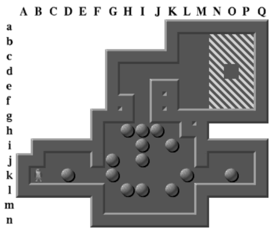


Problem #6

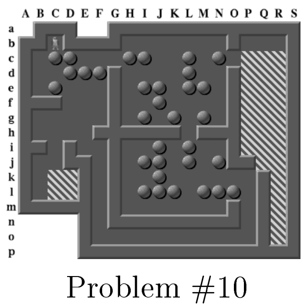


Problem #7

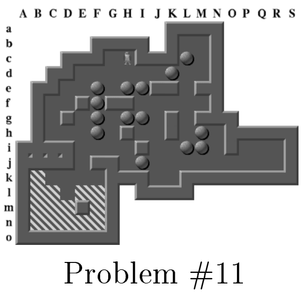
Problem #8



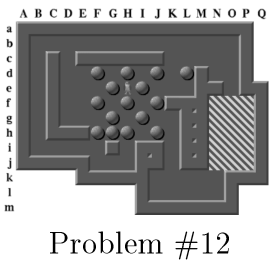
Problem #9



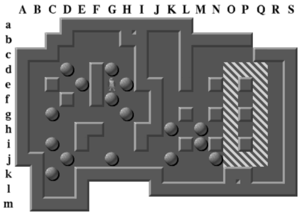
Problem #10



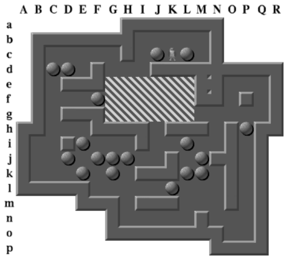
Problem #11



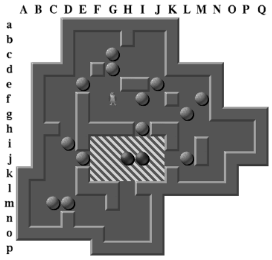
Problem #12



Problem #13

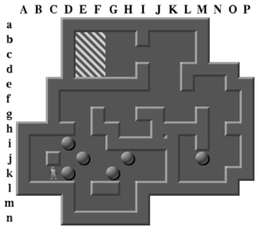


Problem #14

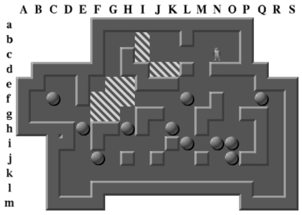


Problem #15

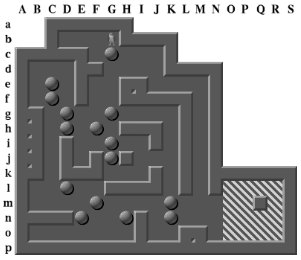
Problem #16



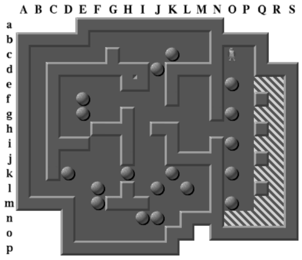
Problem #17



Problem #18

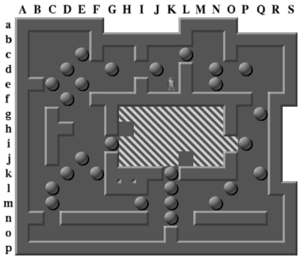


Problem #19

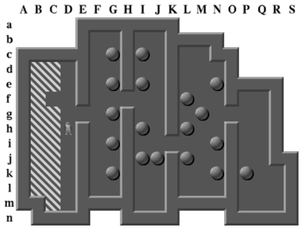


Problem #20

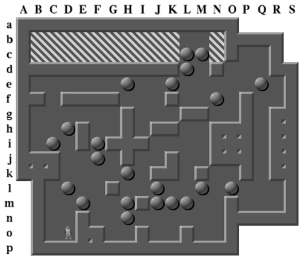
Problem #21



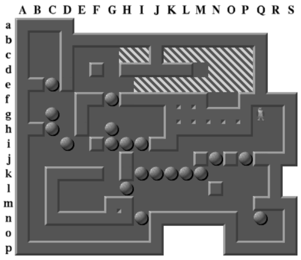
Problem #22



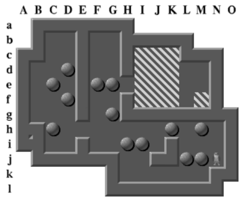
Problem #23



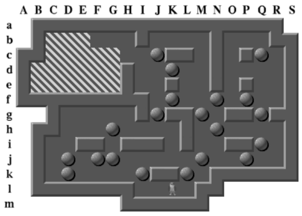
Problem #24



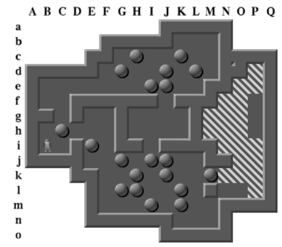
Problem #25



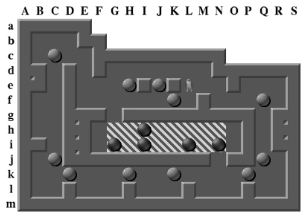
Problem #26



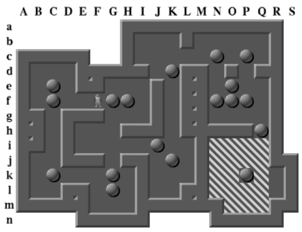
Problem #27



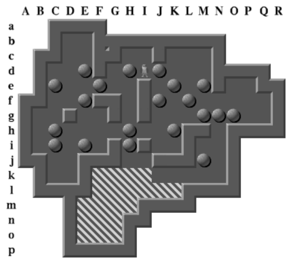
Problem #28



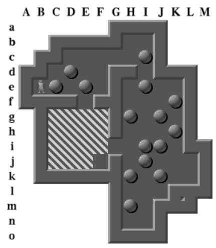
Problem #29



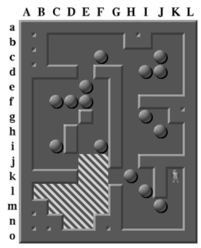
Problem #30



Problem #31

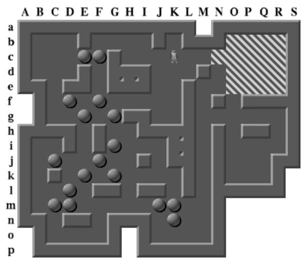


Problem #32

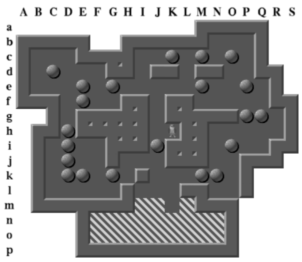


Problem #33

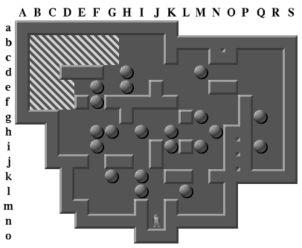
Problem #34



Problem #35

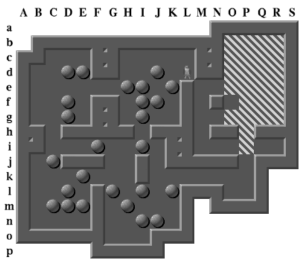


Problem #36

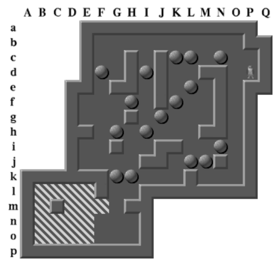


Problem #37

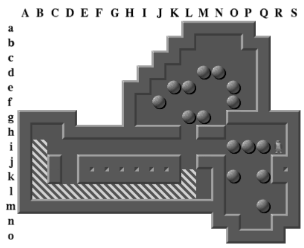
Problem #38



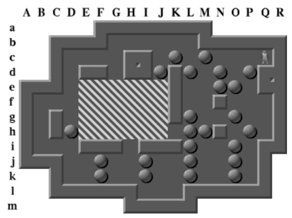
Problem #39



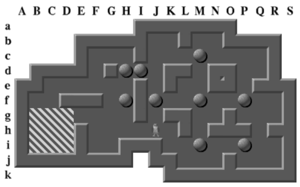
Problem #40



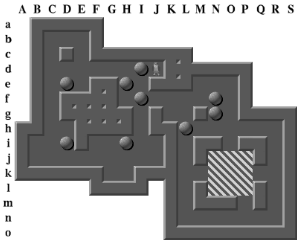
Problem #41



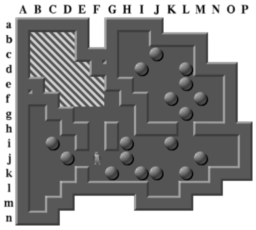
Problem #42



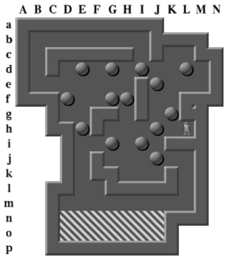
Problem #43



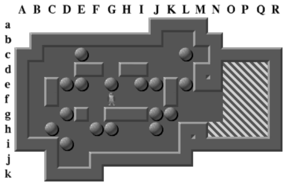
Problem #44



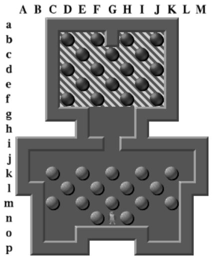
Problem #45



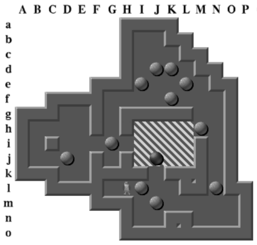
Problem #46



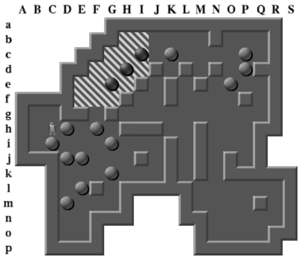
Problem #47



Problem #48



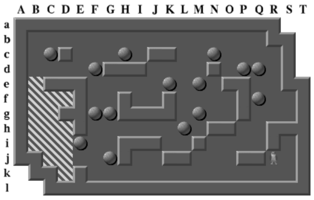
Problem #49



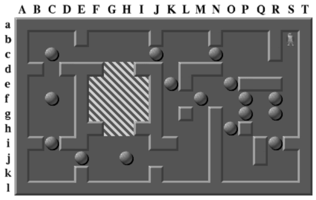
Problem #50

Problem #51

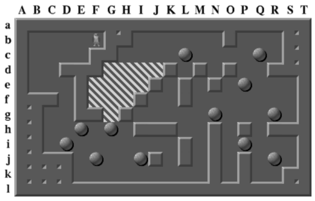
Problem #52



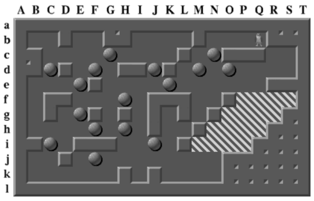
Problem #53



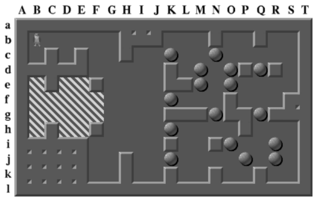
Problem #54



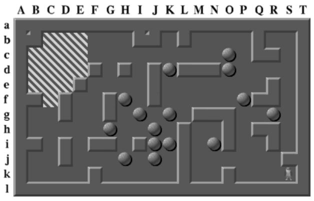
Problem #55



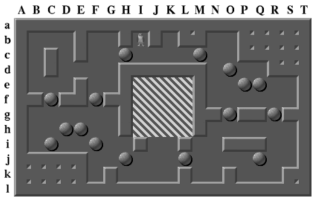
Problem #56



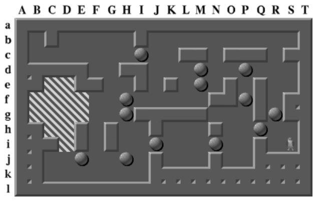
Problem #57



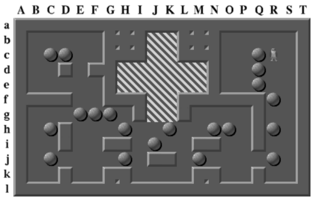
Problem #58



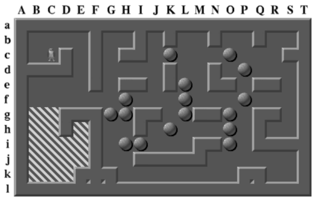
Problem #59



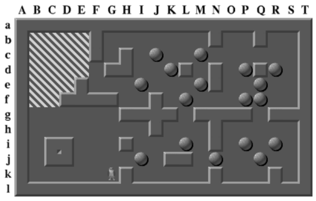
Problem #60



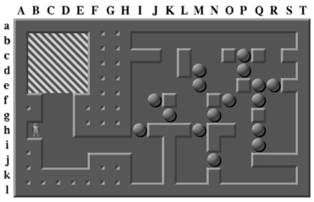
Problem #61



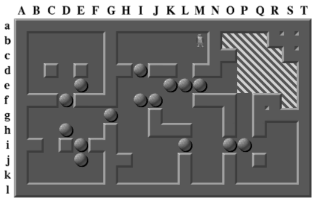
Problem #62



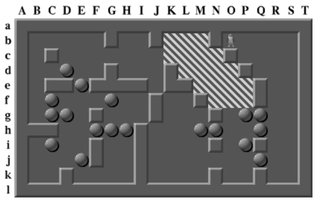
Problem #63



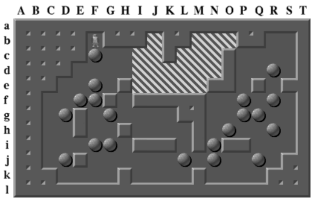
Problem #64



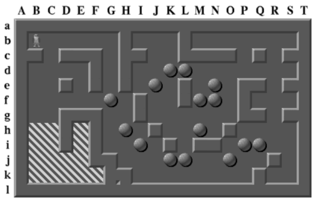
Problem #65



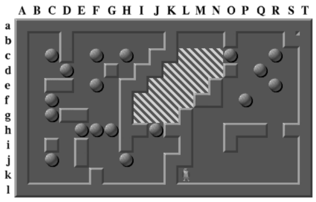
Problem #66



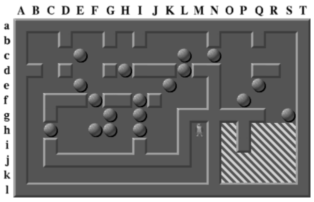
Problem #67



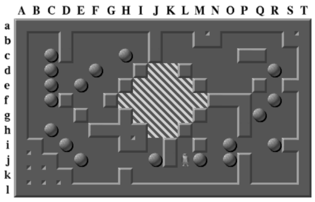
Problem #68



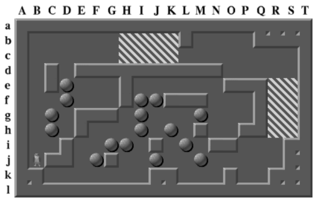
Problem #69



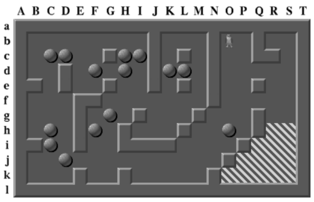
Problem #70



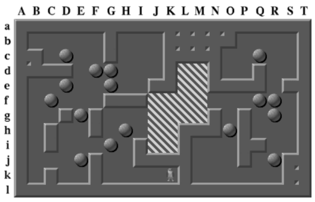
Problem #71



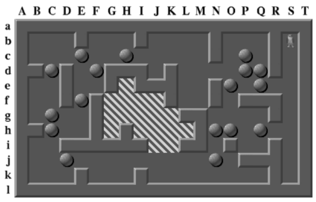
Problem #72



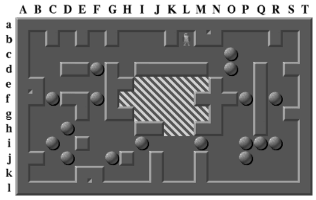
Problem #73

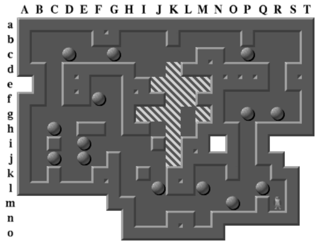


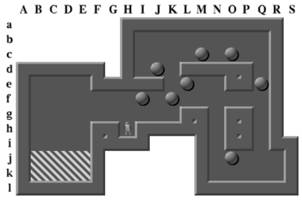
Problem #74

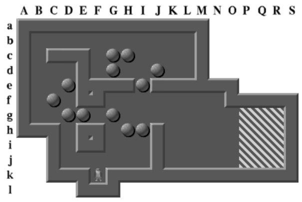


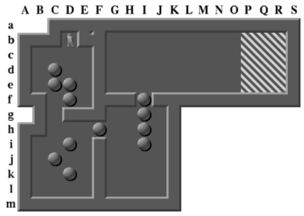
Problem #75

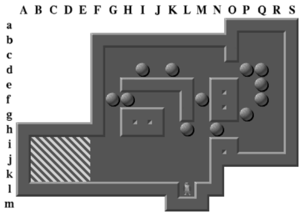

Problem #76

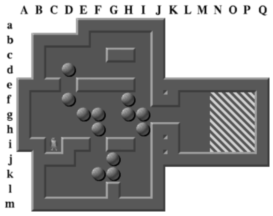

Problem #77

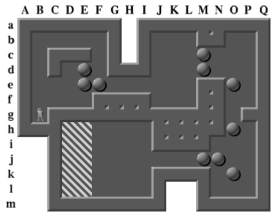

Problem #78

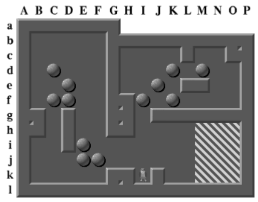

Problem #79

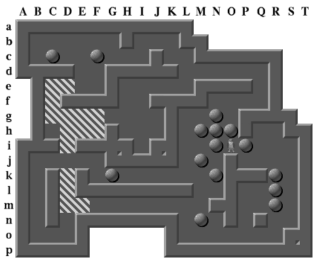

Problem #80

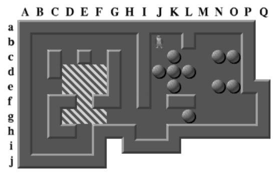

Problem #81

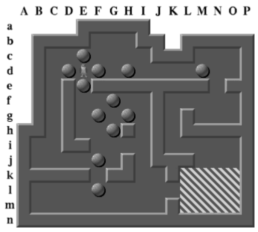

Problem #82

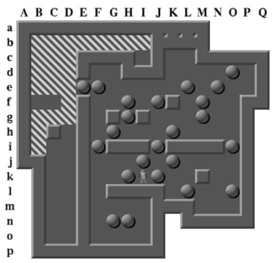

Problem #83

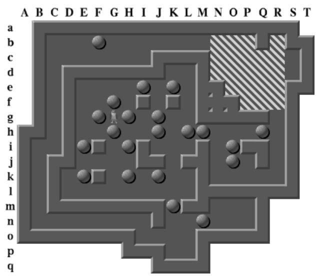

Problem #84

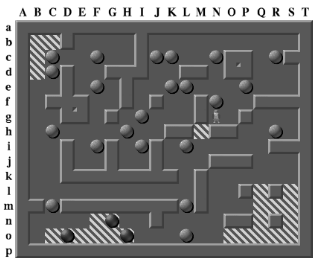

Problem #85


Problem #86


Problem #87


Problem #88


Problem #89

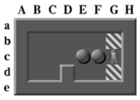

Problem #90

Appendix B

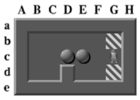
The 61 Kids Problems

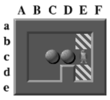

Problem #1

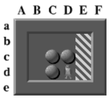

Problem #2

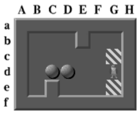

Problem #3


Problem #4


Problem #5


Problem #6


Problem #7

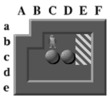

Problem #8


Problem #9


Problem #10

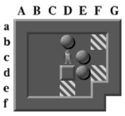

Problem #11


Problem #12

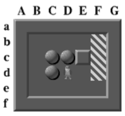

Problem #13


Problem #14

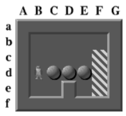

Problem #15


Problem #16


Problem #17

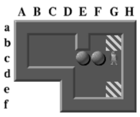

Problem #18

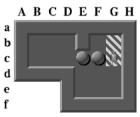

Problem #19

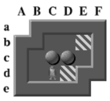

Problem #20

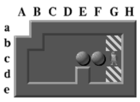

Problem #21

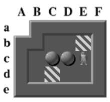

Problem #22


Problem #23


Problem #24


Problem #25


Problem #26

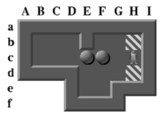
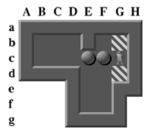

Problem #27


Problem #28


Problem #29

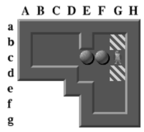
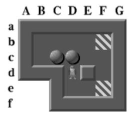
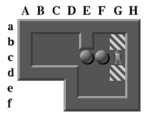
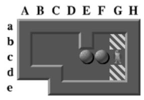

Problem #30

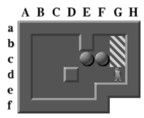
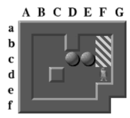

Problem #31

Problem #32

Problem #33

Problem #34

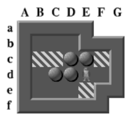
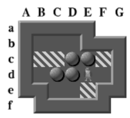
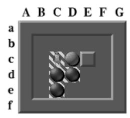
Problem #35

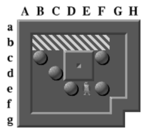
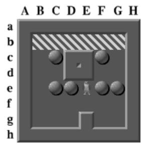
Problem #36

Problem #37

Problem #38

Problem #39

Problem #40

Problem #41

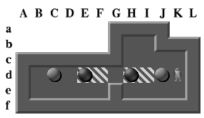
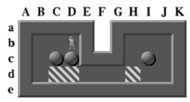
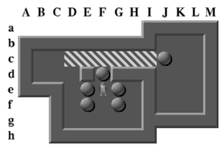
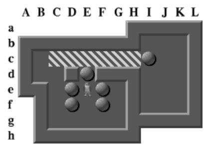
Problem #42

Problem #43

Problem #44

Problem #45

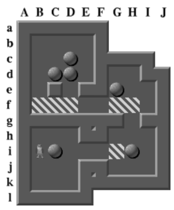
Problem #46

Problem #47

Problem #48

Problem #49

Problem #50

Problem #51

Problem #52

Problem #53

Problem #54

Problem #55

Problem #56

Problem #57

Problem #58

Problem #59

Problem #60

Problem #61

Appendix C

Implementation Details

To improve readability and not to wear the patience of the reader too thin, we decided to move most of the implementation details into this appendix. These details are non-essential to the basic ideas of the algorithms, but are important to understand how the implementation is realized. It is geared towards explaining how something works, without trying to justify anything. There will be more "magic numbers", but we will not point to them specifically anymore.

C.1 Construction of Deadlock Tables

This section contains a detailed account of the implementation used to construct the deadlock tables.

An off-line search was used to enumerate all possible combinations of walls, stones and empty squares for a fixed-size region. For each combination of squares and their contents, a small search was performed to determine whether or not a deadlock was present. This information was stored in a tree data structure.

Each node in the tree of Figure C.1 represents a certain pattern of stones, walls and empty squares. The root of the tree is the empty maze, except for the man and one stone. The three successors of the root represent a pattern with an additional stone, wall or empty square. Each of their successors represent a pattern containing one more stone, wall or empty square, and so on. Figure C.2 shows a possible order of placing/querying the squares in a maze. The pattern with a wall on square 1 and a wall on square 2 represents a deadlock, and the tree terminates at that point. To find out if a certain pattern is a deadlock, a special search is performed which tries to push all stones to goal squares. Every square that is not part of the currently investigated pattern is a goal square. If the search fails to find a solution – pushing all stones to goals, a deadlock pattern was discovered.

There are many optimizations that make the computation of the tree more efficient.

- If a wall is placed, such that a stone becomes immediately deadlocked (the wall creates a dead square on which a stone is positioned), the search can be avoided and deadlock is declared immediately.

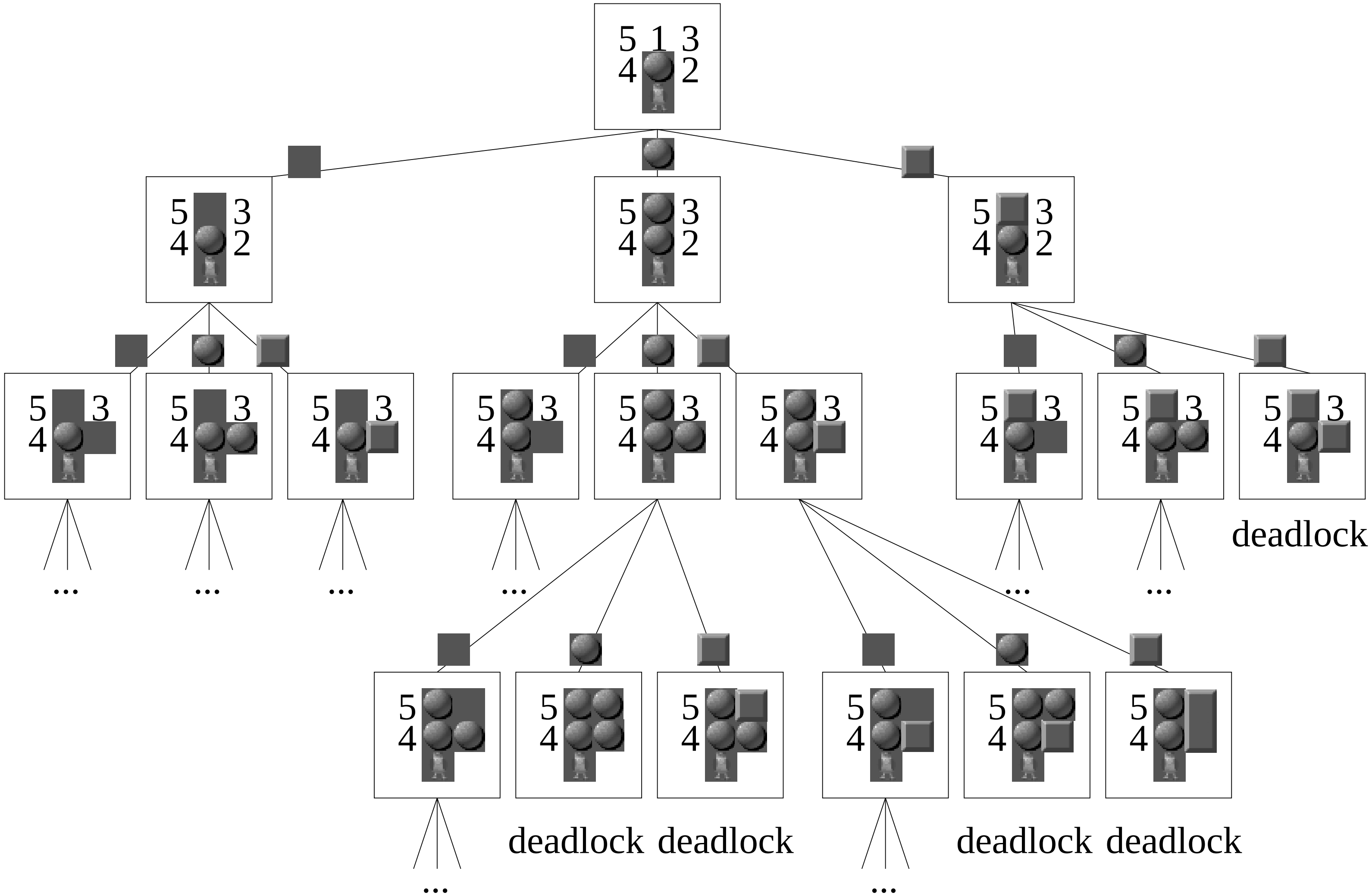


Figure C.1: Deadlock Tree

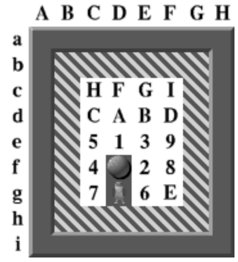


Figure C.2: Example Deadlock Table Query Order

Figure C.3: Goal Macro Example

- If neither placing a wall nor a stone on a particular square created a deadlock, placing an empty square there cannot create deadlock either.
- When placing a stone, we can check if the patterns computed so far can identify this position as a deadlock.
- The search of a deadlock pattern can be sped up by removing stones immediately when they reach a goal square.
- The search can use a cheap lower bound that sums the distances of all stones to their respective closest goal.

Even with all these enhancements, computing a deadlock table of approximately 5x4 takes several weeks of computation, since most interior nodes of the pattern tree represent a small search, averaging several hundred nodes. Pushing deadlock tables further would require an enormous number of CPU cycles and the effects would be limited (see Section 4.6.4).

C.2 Goal Macros

C.2.1 Goal Rooms

A goal room is a vague concept. Humans rarely define hard boundaries as are needed by a program trying to precompute goal macros. The procedure described here should not be viewed as the ultimate answer to the problem of goal room detection.

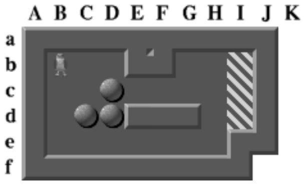


Figure C.4: Two Kinds of Entrances

First, all goal squares that are direct neighbors are included in the goal room. If less than three goal squares are found together, no goal room is created, since the possible savings do not outweight the risk of producing unsafe goal macros (see Section 4.7.2). Then, using the goal squares as a start state, a highly pruned depth-first branch-and-bound search is executed that searches through the search space of goal room configurations for the "best" goal room. "Best" is defined as follows:

- include as few stones as possible,
- leave as few entrances as possible, and
- include as many squares as possible.

The primary concern is to identify a goal room with a minimum number of entrances, and then, if possible, to maximize the number of squares in the goal room. At each node in this goal-room search, successor states are goal-room configurations increased by one square through which a stone can enter the current goal-room configuration. To improve efficiency, a transposition table is used to prevent duplicating work. The result of this search for the example problem in Figure C.3 is a goal room with the entrances at the squares *Dc* and *Dd*. Note, that the squares *Gc* and *Gd* would also form a goal room with two entrances, but with fewer squares inside.

To determine goal rooms is extremely difficult, because many problems influence it. The larger the goal room the larger the potential gains, but also the higher the risk of creating goal macros that cut off solutions. Fewer entrances are generally preferred, since many entrances increase the risk of blocking communication channels. If goal rooms are too small however, the procedures described in the following sections might not be able to find solutions because stones have to temporarily leave the goal area.

C.2.2 Entrances

There are generally two types of entrances.

Man Entrance: an entrance through which only the man can enter, and

Stone Entrance: an entrance through which stones (and man) can enter.

If we talk about "entrance" without specifying if it is a man or stone entrance, we will assume it is a stone entrance. For example, assume the entrances to the goal room in Figure C.4 are *Ec* and *Ee*, then *Ec* is a stone entrance, since stones can reach goal squares from *Ec*. However, the entrance at *Ee* is a man entrance only; no stone can reach a goal from this entrance.

C.2.3 Goal-Macro Trees

Having identified a goal room, another off-line search now creates a goal-macro tree. We call this search *goal-macro tree generation* and it is discussed in detail in Section C.2.5. Figure C.5 shows an example of a goal-macro tree[1]. Each node in this tree represents a specific configuration of stones in the goal room. Edges between the nodes represent macro moves. Each edge is labeled with the number of pushes required by the macro it represents. A macro move is defined by the entrance square and the final goal square the stone is pushed to. The root of the tree represents the empty goal area at the beginning of the search. If at any point in time during the search a stone is pushed to the entrance of a goal area, the goal-macro tree is consulted as to which macro(s) should be tried by looking up the node that represents the current stone configuration in the goal room. To speed up the process of finding the correct node in the goal-macro tree, a pointer is kept that points to the node in the goal-macro tree that represents the current stone configuration in the goal area. This pointer is updated every time a goal-macro move is made or undone.

C.2.4 Target Squares

Given a certain stone configuration in the goal area, the goal-macro generation has to solve the problem of which square(s) should the next stone be pushed to. These squares are called target squares. There is a potentially different set of target squares for each entrance.

Several properties of the empty goal squares are considered. Figure C.6 shows an example goal room that we will be using to explain the following concepts. There are five entrance-independent properties used:

FIXED: The stone would be fixed if placed on this square. Squares *Ih*, *Kh* and *Ke* have this property.

DEAD: Placing a stone on this square would render one or more other empty goal squares immediately inaccessible, essentially creating a deadlock. The squares *Jg* and *Jf* have this property. Note that the emphasis is on *immediately*, a one-move look-ahead. Placing a stone on *If* creates a deadlock as well, but only deeper look-ahead is able to verify that.

NONOBSTRUCT: The stone would not obstruct any path to any of the other squares, meaning if a square is reachable from some entrance under some conditions, it still is. The squares *Ih,Jh* and *Kh* are such squares. However, the

[1] In effect, we treat goal-macro trees as graphs for efficiency reasons. We still call it a tree, because this is more intuitive.

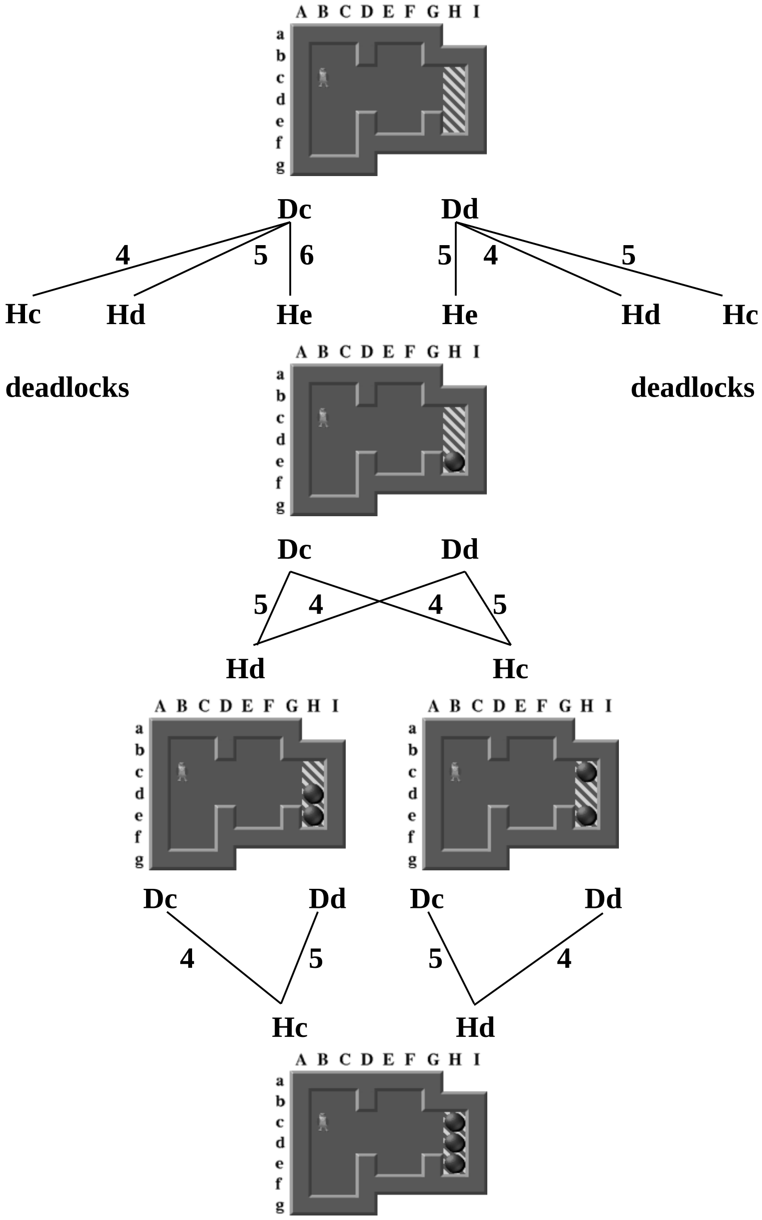


Figure C.5: Goal-Macro Tree Example

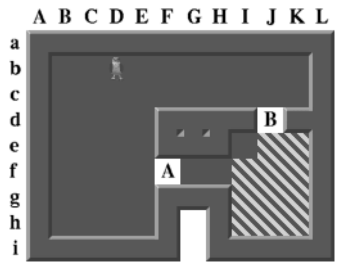


Figure C.6: Target Squares

square *Ig* is not. It obstructs the reachability of square *Ih* from entrance *A*. A longer path is needed to get the stone to *Ih*.

ACCESS: Placing a stone on this square would render some square inaccessible from some entrance. Squares *If* and *Je* are example squares with this property.

COMMUNICATION: Placing a stone on a square with this property does not cut off any communication paths between stone and man entrances. If possible, one should avoid placing stones on squares that cutoff certain areas of the maze.

The following square properties are with respect to a specific entrance:

OPTIMAL: The stone can reach the target square with an optimal number of pushes; no other stone is placed such that we have to make a detour. All squares have this property for each of the two entrances in our empty example maze. The man is allowed to leave the goal area.

INSIDE: When pushing the stone to a target square with this property, the man does not have to leave the goal area. The squares *Ig* and *Ih* don't have this property for entrance *A*, neither has square *Ke* for entrance *B*, given that optimality is required.

STRICT: This property is a combination of OPTIMAL and INSIDE. Squares *Ig, Ih* and *Ke* do not have this property. Either the man needs to leave the goal room or the macro move will require a non-optimal push sequence.

LOOSEST: A target square with this property is only reachable with the man leaving the goal area and the stone taking a non-optimal path.

CLOSEST: CLOSEST is trying heuristically to guess where stones should go if they come through a specific entrance. Since entrances can be arbitrarily far away from the first goal square (see entrance *A* for example), CLOSEST is with respect to the closest goal square to that entrance. For entrance *A* the squares *If,Ig* and *Ih* are closest, so are the squares *Je* and *Ke* for entrance *B*.

A heuristic function evaluates each of the target squares using the properties described above. These values and properties are used to order the target squares to allow for a more efficient goal-macro tree generation.

C.2.5 Goal-Macro Tree Generation

Goal-macro tree generation is a search that traverses the highly pruned search space of stone configurations in the goal area to find possible ways to pack the stones into the goal room. Stones can enter through all stone entrances and in any order. The search saves its results in a goal-macro tree, such that the IDA* search can reuse the knowledge found by the goal-macro tree generation.

Each node the search tries a set of target squares for each entrance. It places a stone on each of the squares in turn and recursively calls itself. The recursive call

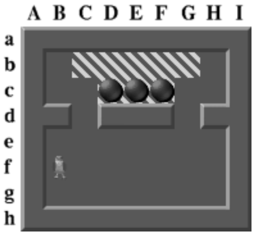


Figure C.7: Pivot Point Example

returns successfully if at least one possible way was found to pack all the stones into the goal area. In that case, the goal macro to the target square is added to the current node. The search attempts to satisfy each of the following properties with at least one target square: CLOSEST, OPTIMAL, INSIDE and NONOBSTRUCT. If the search cannot find any successful target square at a node, it returns with failure.

C.2.6 Pivot Points

If all of the target squares have the ACCESS property (they cut off some square from some entrance), we call the position a *pivot* position and all squares are included as macros. This is necessary because at pivot points in the search there is no way of knowing how many stones will be pushed through which entrance. Figure C.7 shows one such position. Placing the stone at any of the remaining goal squares divides the goal area into parts accessible only from one entrance. Since the goal-macro generation cannot know what happens during the IDA* search, any guess might be wrong. Hence all squares have to be included.

C.2.7 Included Stones

The goal-macro tree creation assumes an empty goal area at the start of the search. If stones were included in the goal area of the start configuration, then the goal-macro tree that is created, and the pointer the search has into it representing the current stone configuration in the goal area, do not correspond. Whenever a move is generated for a stone inside the goal area during IDA*, a special routine is called that tries to put the stone in the closest goal square that the current goal-macro tree node offers.

C.2.8 Parking

If the first attempt to build a goal-macro tree fails, it was most likely because a parking maneuver is needed that the search cannot handle. A second attempt is started, and this time the search is allowed to keep nodes in the goal macro tree that have no successor. It is assumed that at that point, stones need to be parked and that the IDA* search will be able to solve the parking problem. Since parking happens mostly late in the goal packing, we can get most of the benefits of goal macros without rendering the problems unsolvable.

C.3 Customizing IDA* for Pattern Searches

If the pattern searches used the same IDA* procedure and lower-bound estimator as in *Rolling Stone*, the search would be prohibitively large and slow. Instead, we use a special version of IDA* (PIDA*) that is customized for pattern searches, allowing for additional optimizations that dramatically improve the search efficiency. By relaxing the rules of Sokoban and introducing new goal criteria, the resulting search is more efficient and still returns an admissible lower bound on the solution.

C.3.1 Stone Removal

One enhancement is to remove stones from the test maze once they reach a goal square. For deadlock PIDA* searches, stones are also removed when they are pushed onto a man-reachable square. This comes from the observation that most deadlocks result in a number of stones getting *crowded* together. Hence, if a stone "breaks free", we assume we no longer need to consider it in that search subtree.

C.3.2 Multiple Goal States

Another optimization is to relax what we consider a goal state. In this relaxation, goal states are also positions where the man can reach all squares, and at least one conflict with the current StonePath has been found. Penalty PIDA* searches do not use this simplification.

These shortcuts simplify the search leading to large savings in the cost of a pattern search. However, this comes at the expense of possibly missing a penalty or deadlock. In practice, the reduced search effort more than compensates for the few missed opportunities.

C.3.3 Efficient Lower Bound

Since stones get removed from the board when they reach a goal square, the Min-matching lower-bound heuristic is not appropriate. A cheaper heuristic can be used: the sum of the shortest distances of each stone to its closest goal. When a stone moves, this lower bound is easily updated. This results in large savings in the cost per node compared to the original $O(n^3)$ lower bound. Since the number of stones is small in a pattern search, most search-related routines are fast, because their cost depends on the number of stones in the maze.

C.3.4 Transposition Table Entries

Usually, if an IDA* search is started, the transposition table has to be cleared, since old entries are not valid for the current search. Since multiple PIDA* searches are run on the same problem just with different stone configurations, we can potentially reuse transposition table entries from previous PIDA* searches within the same IDA* search. However, special care has to be taken when updating the transposition table at the end of an aborted search. Reusing entries from previous searches can drastically reduce the overhead for the PIDA* searches.

C.4 Relevance Cuts

C.4.1 Influence Table

Our implementation runs a shortest-path finding algorithm to find the largest influence between any pair of squares. The first is referred to as the *start* square; the second as the *destination* square. Each square on a path between the start and destination squares contributes points depending on how it influences that path. The more points are associated with a pair of squares, the less the squares influence each other. The exact numbers used to calculate influence are the following:

Alternatives: A square s on a path will have two neighboring squares that are *not* on the path. For each of the neighboring squares n, the following points are added: 2 points if it is possible to push a stone (if present) from s to n; 1 point if it is only possible to move a man from s to n; and 0 if n is a wall. Thus, the maximum number of points that one square can contribute for alternatives is 4.

Goal-Skew: However, if s is on an optimal path from the start square to any of the goals in the maze, then the alternative points are divided by two.

Connection: The connection between consecutive squares along a path is used to modify the influence. If a stone can be pushed in the direction of the destination square, then 1 point is added. If only the man can traverse the connection between the squares (moving towards the destination square), then 2 points are added.

Tunnel: If the previous square on a path is in a tunnel, 0 points are added, regardless of the above properties.

Figure C.8 is used to illustrate influence. For a subset of squares in the figure, Table C.1 shows the influence numbers. In this example, the program automatically determines that an influence relationship > 8 implies that two squares are *distant* with respect to each other. How this threshold is determined is described in the next section.

In this example, square A is influencing squares B and C. However, only B is influencing A (the non-symmetric property). The table shows that there are several

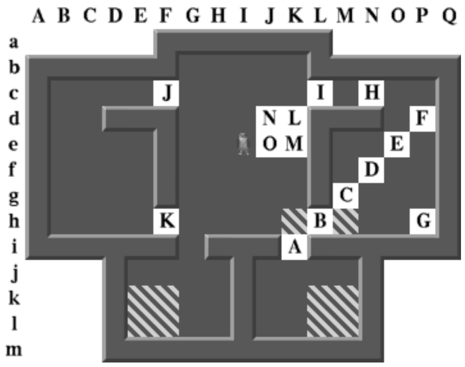


Figure C.8: Example Squares

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
A	1	6	10	18	19	21	13	17	17	24	12	12	10	18	16
B	4	1	5	13	14	18	8	16	16	22	10	11	9	17	15
C	7	4	1	9	10	15	9	15	15	25	13	14	12	20	18
D	11	8	5	3	9	14	12	14	14	29	17	18	16	24	22
E	13	10	7	7	2	7	9	7	7	26	19	12	14	18	20
F	23	19	17	18	10	2	13	6	6	25	34	11	13	17	19
G	12	7	9	15	14	11	1	15	15	34	23	19	17	25	23
H	10	10	14	17	9	5	16	1	1	11	16	3	4	7	8
I	10	10	14	17	9	5	16	1	1	11	16	3	4	7	8
J	16	16	20	27	19	15	23	11	11	1	10	11	12	11	14
K	10	10	14	22	23	26	17	22	22	16	1	17	15	23	21
L	8	8	12	20	14	10	15	6	6	21	14	1	2	7	8
M	7	7	11	19	16	12	14	8	8	23	13	3	1	9	7
N	12	12	16	24	18	14	19	10	10	16	18	5	6	3	6
O	11	11	15	23	20	16	18	12	12	18	17	7	5	8	3

Table C.1: Example Influence Values

regions with high locality, whereas most of the entries indicate non-local relationships. Given the high percentage of non-local entries in the table, one might expect relevance cuts to eliminate most of the search tree. This is not quite true, in that a sequence of local moves can result in the start and end squares of the move sequence not being local with respect to each other.

Consider calculating the influence between squares A and C, as well as C and A (see Table C.2). The table entries correspond to the contribution of each of the influence properties. The table indicates the influence scores for the squares A, B, C, and the intermediate squares p and q, as well as for the connection between the squares (indicated by the arrows). Each line modifies the previous line (adding new values or changing existing values). The final influence, the sum of the preceding columns, is shown in the last column.

$InfluenceTable[C, A]$	C	$\rightarrow$	q	$\rightarrow$	B	$\rightarrow$	p	$\rightarrow$	A	influence
alternatives	1	0	2	0	0	0	4	0	0	
connection	1	1	2	1	0	1	4	1	0	
tunnel	1	1	2	1	0	0	4	1	0	
goal-skew	1	1	1	1	0	0	2	1	0	**7**
$InfluenceTable[A, C]$	A	$\rightarrow$	p	$\rightarrow$	B	$\rightarrow$	q	$\rightarrow$	C	influence
alternatives	2	0	4	0	0	0	1	0	1	
connection	2	1	4	1	0	1	1	2	1	
tunnel	2	0	4	1	0	0	1	2	1	
goal-skew	1	0	4	1	0	0	1	2	1	**10**

Table C.2: Example Influence Calculation

C.4.2 Parameter Settings

To reflect differences in mazes, the parameters $infthreshold$ and m are set at the beginning of the search. The maximal influence distance, $infthreshold$, is computed as follows:

1. Compute the average value for all entries $InfluenceTable[x, y]$ satisfying the condition that square y is on an optimal path from x to any goal.
2. The average is too high. Scale it back by dividing it by two.
3. To ensure that the cuts are not too aggressive, $infthreshold$ is not allowed to be less than 6.

The length of the history used, m, is calculated as follows:

1. Compute the average value for all entries $InfluenceTable[x, y]$ satisfying the condition that a stone on square y can be pushed to a goal (e.g. in Figure C.8, squares F and G would not be included).
2. To ensure that the cuts are not too aggressive, m is not allowed to be more than 10.

Appendix D

Failed Ideas

We had no shortage of "good" ideas during the Sokoban project, many of which did not produce the expected results. This appendix contains a collection of the most promising of these ideas that we tried but did not lead to significant improvements. Even though this might not be considered as a contribution of the thesis (hence the location in the appendix), it might prove valuable for the keen beginners in Sokoban to caution themselves.

We have found discussions about Sokoban invariably leading into certain directions, suggesting things to try and ideas to pursue. Unfortunately, we then have to point out that we have tried many of the suggested ideas with little or no success. Some of the ideas are saving search nodes, but come at a cost that prohibits their use because the overall runtime increases. Other methods work, but only for so few problems that we did not want them included because we were afraid they could do more harm than good.

Wherever possible, we try to identify the problems that need to be solved in order to make some technique feasible. This, of course, is not necessarily a complete list, other problems might exist that we are not aware of. The reader might also detect a sarcastic tone in the description of some of our attempts - unfortunately that is what often remains. After months of design, implementation, tuning and debugging, redesign, reimplementation and tuning and debugging again, and again, the insights gained are often demoralizing in nature. The search-tree size is one of these concepts we still fail to appreciate completely. We are faced with such an incredibly large search space that searching in it for solutions seems like the proverbial search for the "needle in the haystack" - except that we probably face an even more daunting task.

This appendix might therefore seem like a turnoff to many, but it is not meant like that. We are convinced that a public record of "ideas that failed" is needed, not just for projects like Sokoban, but for Computing Science in general. This appendix is an attempt to start such a record for the domain of Sokoban and single-agent search. It might help to spark and/or further the discussion into the merits of these ideas in general, ultimately possibly leading to interesting publications in their own right. The potential benefits are manifold and not restricted to avoiding duplicated efforts, but include focusing future research, improving experimental work and increasing the communication on hard problems.

Figure D.1: Development Chain of Success

Why is it so hard to make an idea work? Figure D.1 shows what it takes for us to consider an idea a success. We need to make three critical transitions.

1. We need to take the idea that was conceived (and often has a vague nature) and form hard concepts around it. It is often easy to utter fuzzy ideas, but the moment one has to be more concrete, it is much more complicated to capture what was meant when one was thinking about it. For example, what do you mean with "avoid difficult stone configurations"? Maybe the concept of crowding is more concrete: "Many stones in a restricted area must be avoided."

2. The second step is to develop a concrete algorithm that represents an idea. How can a hard concept be put into a concrete algorithm? For example, how can crowding be calculated in a Sokoban maze? What is "many stones" for a "restricted area" and what does it mean to "avoid" such situation in the search: cutting them off, or postponing them?

3. Assuming we haven't failed so far, now we have to find a way to implement the algorithm efficiently. Saving 50% of the nodes is not good enough if each node searched becomes 10 times more expensive.

Often, even though all three hurdles were passed successfully, one finds out that a certain idea is redundant with another idea already present in the solver and only small additional gains are possible. This is especially prevalent in high-performance solvers that are very efficient already: improving on their performance is often incredibly challenging.

Worse are interacting features. Even though the new search enhancement works like a charm, it hinders another one and they conflict in such a way that the overall performance drops.

We have also seen what we call logical bugs. After developing and implementing an idea, thorough testing might reveal that pathological cases or exceptions exist. To handle those exceptions well is often the difference between failure and success of a method.

And then there are implementation errors - or bugs for short...

We have encountered all these problems–and more–while working on the Sokoban project. The following list of failed ideas provides some insight into our efforts in tackling the "Go of single-agent search": Sokoban. We focus here on the description of the high-level ideas that failed. Many ideas took weeks of effort to convince ourselves of their futility.

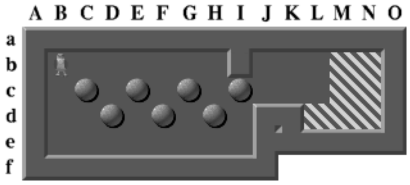


Figure D.2: A Problem Where Backward Search Wins

D.1 Means End Analysis

Even before we started seriously thinking about exhaustive search, we invested several weeks in trying to make Means End Analysis (MEA) work. The problem we encountered is how to restrict and order all the possible choices of moves. Interacting subgoals do not allow for easy ordering of the choices. A lot of domain knowledge would be needed to solve this problem using MEA.

D.2 Backward Search

As already pointed out in Chapter 2, there is nothing forcing us to search from the initial position forward to find a goal state. We could also start from a goal state and search backwards. There are several problems we face when implementing such an approach:

- Multiple start states: Because only the stone/goal locations are defined, the man can potentially be in different places in the goal position. For push-optimal solutions that is a small number, like 1 to usually not more than 4, but for move-optimal solutions this can be quite a bit more. Of course, that increases the search-tree size.

- We trade forward deadlocks for backward deadlocks. Backward deadlocks are usually easier to detect; the man runs out of moves, because it is "compressing" its own space. But, there are backward deadlocks that are just as hard to detect as forward deadlocks. Those are the ones where the man compresses an area of the maze, but can escape to do other (futile) work, leaving a few stones in a locked position. We found with the pattern searches a way to detect forward deadlocks, but we would have to change them to detect backward deadlocks.

- Usually, goals are in goal areas (forward searches), but when searching backwards, they are scattered throughout the maze, making it hard to establish orders in which you want to put stones in. Goal macros, one of our most valuable search enhancements, are useless.

We have a maze in which our backward search beats our forward search, but we had to specially design it to get the effect. Figure D.2 shows the maze. The forward search needs 99,829 nodes to solve the maze; the backward search needs only 10,244.

This comparison is not quite fair, since the backward search is not using pattern searches and thus does not create their corresponding overhead, nor can it appreciate any benefits. However, turning off pattern searches in the forward search is an even bigger looser (364,006 nodes) against the backward search.

The feature in the problem in Figure D.2 that makes this problem amenable to backward search is that the backward search detects early on (high up in the search tree) that an extra move is needed to get all the stones out of the goal area. The forward search detects this only deep in the search, resulting in a much larger search tree before it switches into the second iteration at which point good move ordering results in a quick successful termination of the search.

However, in general, the backward searches were much larger than the forward searches, mostly because of the lost opportunity to apply the goal macros.

D.3 Bidirectional Search

Let's assume we solve the problems with the backward search. We could combine forward and backward search to form a bidirectional approach. When using iterative deepening for forward and backward searches, one can easily alternate both directions. Since both directions can have different search-tree sizes, it seems natural to exploit this fact. We tried to deepen the search direction that had a smaller tree in the previous iteration.

The general idea is the following: Let's say the lower-bound function estimates the length of a solution (distance to the goal from the root node) to be D. We can start the forward search using normal IDA* and give it an additional hard depth limit of, say, $X \leq D$. Then we start the backward search and give it an additional depth limit of $D - X$. If there was a solution with length D, then the search frontiers should meet at depths X and $D - X$. If the frontiers did not meet, no solution exists with length D. Consequently, we increase the target solution length and start either a forward or a backward search, now with increased threshold D and increased additional depth limit X or $D - X$. If of the two initial searches the forward search had the smaller tree, it is probably a good idea to use forward search for the next step.

However, the meeting of the search frontiers is still the most important problem. Traditionally, this is regarded as the main drawback of bidirectional search. The trouble is that the search frontiers can be so large that one needs lots of memory to store them. We changed the transposition table code to detect that, but the likelihood of overwriting entries is high. Table replacement schemes usually prefer entries from deep searches, and shallow entries are thrown away. We changed that to save the frontier nodes from clobbering. Then, the table is flooded with all the frontier nodes.

X is an important variable here. Setting X to about half of the solution length will keep both trees about the same size, maximizing the theoretical savings of bidirectional search. Using X biased towards cutting one search shorter than the other will keep at least one search frontier small, allowing it to be stored with practical amounts of memory.

We faced two major problems with our implementation. The first was predicting which direction was more profitable to search. The previous search sizes are only a weak predictor of the size of the next iteration, because iterations grow rather erratically. Second, what should X be set to? We tried an iterative approach, increasing X for the backward search for the same threshold D, but that is not cost effective because large portions of the tree are re-searched.

Future directions might include searching both directions unbounded, assuming that the first iterations will fail to make the frontiers meet, and to record how deeply they penetrated the tree to make a more informed decision as to which search direction to grow next and where to set the depth limit. It becomes obvious here how important control functions are. They could control the setting of X and D and switch the search directions, basically controlling the search using information gathered by previous searches.

D.4 Real-Time Search

The search now spends all its allocated time to find a solution. What if we had to control a robot that had to move every n seconds? If n is small enough, such that we cannot find a complete solution, we have to commit to a move without knowing if that move leads to a solution. Much worse, and unique to domains with directed search spaces such as Sokoban, by making a move, we might introduce a deadlock and thus never be able to solve the problem.

Our attempt at real-time search tried to minimize the risk of being trapped in a deadlock by executing the following steps:

1. Spend about 25% of the allocated time to order moves by small searches and using the search results to estimate which move is best. If, by chance, we find a solution, we are done, the rest of the problem is easy. Else, go to step 2.
2. Check if the currently best move is reversible (use 25% of effort). If the best move is reversible, go to step 4. Else, go to step 3.
3. Check if we can find a deadlock (use 25% of effort). If so, goto step 1. Else, go to step 4.
4. Execute best move. Go to step 1 to find next move.

The last 25% are "banked" for the cases when we have to return to step 1 because a deadlock was found in step 3. This procedure usually gets caught in a loop, because it finds reversible moves attractive. Some measure of progress is needed to guide the real-time search. Otherwise the threat of deadlock forces the program into a "safe", but unproductive choice.

D.5 General Pattern Databases

After a search finishes, the patterns found by the pattern searches are just forgotten. We implemented a scheme where these patterns are saved and then reused in later, different mazes.

We restricted the patterns saved to areas of size 6x6 and filtered all patterns to remove "loose" walls. These are walls that, when removing the context outside the 6x6 area, are not connected to any other wall and do not neighbor any stone. The resulting patterns are then appended to a dynamically growing database of such patterns. When starting a search, all patterns are matched in all possible ways in the maze (rotating, mirroring) and verified by a small search to make sure that the penalties still hold in the new maze. The approved patterns are then entered into the database.

The overheads at the end and the beginning of the searches are negligible. Even the verification searches, limited to 100 nodes each, are fast. What kills this approach is the pattern matching overhead. Too many patterns are entered in the on-line database, and matching during the search slows the program down significantly. Moreover, the matching rate is small. The majority of the patterns are never matched and only cost overhead without efficiency gain. That is a typical instance of the utility problem [Min88].

After we abandoned this idea we implemented the pattern limits. With the pattern limits, this method might work better, but additional investigation is needed.

D.6 Stone Reachability

One of the most exciting, but failed, ideas we pursued is the idea of stone reachability. This idea came up in several serious discussions with interested people and was suggested in slightly different variations. Don Beal called it *roaming*, Bart Massey calls them *equivalence classes*, Neil Burch's version was named *stone reach* and Dave Gomboc suggested it as *canonical form*.

The idea is roughly the following: Keep pushing a stone until the reachability of other stones is effected. Reachability of stones is defined as the squares the man can push stones to without pushing other stones in between. Alternatively, one can think of it as the area in which a stone can be pushed around reversibly.

The idea is that one could create pseudo macro moves: as long as no stone reachabilities are changed, keep pushing this stone and do not consider any alternative pushes. However, Sokoban proves itself more difficult than foreseen, again. There are frequently occurring cases when this heuristic fails and truncates solutions. Second-order reachability considerations are of importance. Often, by moving a stone to a certain square, stone reachability is changed, but only if another stone is moved first. We call this shadowing. Moving stone A would change the reachability of stone B, but stone C shadows (restricts) stone B's reachability such that the effect is not immediately visible. Only after removing stone C, one can now see that moving stone A was indeed changing stone B's reachability.

We tried a two-step version of the initial idea. If the first-order stone reachability was not effected, a second test was performed. Now, all stones, except the two in question (A and B in the previous example) are removed and it is determined if stone A's move changes stone B's reachability. If not, we can treat the sequence of moves of stone A as a macro. That works quite well, the search trees get reduced by about 50%, however, the cost of computing stone reachability increases the cost of computation for each node by about 10 times! The net loss is about 5 times longer runtimes. The reader is cautioned to assume this might be an implementation inefficiency. We reduced the cost of naively computing stone reachability by many clever enhancements by at least one order of magnitude and closer examination reveals why this cost is so high. To compute stone reachability, one has to compute man reachability many times over (at least around the stone to be pushed) and that is an expensive computation (depth- or breadth-first search).

It is hard to accept the fact that a beautiful idea fails on something as trivial as computation cost of a constant factor. Unfortunately, high-performance problem solvers cannot conveniently ignore these constants and thus, even nice ideas are often retired after months of intense intellectual, programming, tuning and debugging efforts. This is especially frustrating because one can never be sure when to stop these efforts. *The* brilliant idea might be just around the corner to save the method, or more realistically, one might find *the* bug that caused the implementation to fail. After all, admitting defeat also means giving up on a lot of effort spent.

D.7 Super Macros

Another exciting idea we pursued was that of *super* macros. When a penalty search fails to produce a pattern because there are no more stones conflicting with the current StonePath and ManPath, then this is a hint that the set of stones just considered can be pushed to goals independently of other stones. In principle, the penalty search has just proven that there exists a solution for an independent subproblem in the maze: a set of stones.

The knowledge about the independence of a subset of stones can be used to restrict the IDA* search to this subset, until this entire subset of stones is pushed onto goals. Hence the name *super macros*. This idea was implemented and proven to work, but the savings are small, usually less than 5%. We decided not to use it because there are a few problems with the above reasoning. Pattern searches assume that a stone is going to its closest goal. What if that assumption is wrong? We might not have an independent set of stones; the pattern search could be wrong. While we did not witness such adverse effects, the risk involved seemed too high to ignore compared to the possible savings.

Why are the savings so small? These independent sets of stones are usually close to the goal areas. Usually they are few stones and optimal solutions can be found for pushing them to goals. With our move ordering, IDA* will try these optimal moves close to the goal area first anyways. If these moves lead to goal macros, the goal cuts are already removing alternatives to these moves in case no solution was found.

Super macros are an example of an idea that is almost entirely subsumed by an array of other search enhancements, and adding it on top does not improve the search any further.

D.8 Conclusions

To put it into one sentence:

> Ideas are cheap; making them work is expensive.

Every one of the ideas described in this appendix is interesting, even promising. Most of the time, the reasons behind their failure are not obvious. Future research, hopefully motivated by new insights, might find ways to turn some of these ideas into successful methods. However, it is unlikely to be easy to overcome the problems we encountered.