Near Optimal Hierarchical Path-Finding

Adi Botea Martin Müller
Jonathan Schaeffer
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{adib,mmueller,jonathan}@cs.ualberta.ca

Abstract

The problem of path-finding in commercial computer games has to be solved in real time, often under constraints of limited memory and CPU resources. The computational effort required to find a path, using a search algorithm such as A*, increases with size of the search space. Hence, path-finding on large maps can result in serious performance bottlenecks.

This paper presents HPA* (Hierarchical Path-Finding A*), a hierarchical approach for reducing problem complexity in path-finding on grid-based maps. This technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing each cluster are pre-computed and cached. At the global level, clusters are traversed in a single big step. A hierarchy can be extended to more than two levels. Small clusters are grouped together to form larger clusters. Computing crossing distances for a large cluster uses distances computed for the smaller contained clusters.

Our method is automatic and does not depend on a specific topology. Both random and real-game maps are successfully handled using no domain-specific knowledge. Our problem decomposition approach works very well in domains with a dynamically changing environment. The technique also has the advantage of simplicity and is easy to implement. If desired, more sophisticated, domain-specific algorithms can be plugged in for increased performance.

The experimental results show a great reduction of the search effort. Compared to a highly-optimized A*, HPA* is shown to be up to 10 times faster, while finding paths that are within 1% of optimal.

1 Introduction

The problem of path-finding in commercial computer games has to be solved in real time, often under constraints of limited memory and CPU resources. Hierarchical search is acknowledged as an effective approach to reduce the complexity of this problem. However, no detailed study of hierarchical path-finding in commercial games has been published. Part of the explanation is that game companies usually do not make their ideas and source code available.

The industry standard is to use A* [10] or iterative-deepening A*, IDA* [3]. A* is generally faster, but IDA* uses less memory. There are numerous enhancements to these algorithms to make them run faster or explore a smaller search tree. For many applications, especially those with multiple moving NPCs (such as in real-time strategy games), these time and/or space requirements are limiting factors.

In this paper we describe HPA*, a new method for hierarchical path-finding on grid-based maps, and present performance tests. Our technique abstracts a map into linked local clusters. At the local level, the optimal distances for crossing the cluster are pre-computed and cached. At the global level, an action is to cross a cluster in a single step rather than moving to an adjacent atomic location.

Our method is simple, easy to implement, and generic, as we use no application-specific knowledge and apply the technique independently of the map properties. We handle variable cost terrains and various topology types such as forests, open areas with obstacles of any shape, or building interiors—without any implementation changes.

For many real-time path-finding applications, the complete path is not needed. Knowing the first few moves of a valid path often suffices, allowing a mobile unit to start moving in the right direction. Subsequent events may result in the unit having to change its plan, obviating the need for the rest of the path. A* returns a complete path. In contrast, HPA* returns a complete path of sub-problems. The first sub-problem can be solved, giving a unit the first few moves along the path. As needed, subsequent sub-problems can be solved providing additional moves. The advantage here is that if the unit has to change its plan, then no effort has been wasted on computing a path to a goal node that was never needed.

The hierarchical framework is suitable for static and dynamically changing environments. In the latter case, first assume that local changes can occur on immobile topology elements (e.g., a bomb destroys a bridge). We recompute the information extracted from the modified cluster locally and keep the rest of the framework unchanged. Second, assume that there are many mobile units on the

map and a computed path can become blocked by another unit. We compute an abstract path with reduced effort and do not spend additional effort to refine it to the low-level representation. We quickly get the character moving in a proven good direction and refine parts of the abstract path as the character needs them. If the path becomes blocked, we replan for another abstract path from the current position of the character.

The hierarchy of our method can have any number of levels, making it scalable for large problem spaces. When the problem map is large, a larger number of levels can be the answer for reducing the search effort, for the price of more storage and pre-processing time.

Our technique produces sub-optimal solutions, trading optimality for improved execution performance. After applying a path-smoothing procedure, our solutions are within 1% of optimal.

1.1 Motivation

Consider the problem of traveling by car from Los Angeles, California, to Toronto, Ontario. Specifically, what is the minimum distance to travel by car from 1234 Santa Monica Blvd in Los Angles to 4321 Yonge Street in Toronto? Given a detailed roadmap of North America, showing *all* roads annotated with driving distances, an A* implementation can compute the optimal (minimum distance) travel route. This might be an expensive computation, given the sheer size of the roadmap.

Of course, a human travel planner would never work at such a low level of detail. They would solve three problems:

1. Travel from 1234 Santa Monica Boulevard to a major highway leading out of Los Angeles.

2. Plan a route from Los Angeles to Toronto.

3. Travel from the incoming highway in Toronto to 4321 Yonge Street.

The first and third steps would require a detailed roadmap of each city. Step (2) could be done with a high-level map, with roads connecting cities, abstracting away all the detail within the city. In effect, the human travel planner uses abstraction to quickly find a route from Los Angles to Toronto. However, by treating cities as black boxes, this search is not guaranteed to find the shortest route. For example, although it may be faster to stay on a highway, for some cities where

the highway goes around the city, leaving the highway and going through the city might be a shorter route. Of course, it may not be a faster route (city speeds are slower than highway speeds), but in this example we are trying to minimize travel distance.

Abstraction could be taken to a higher level: do the planning at the state/province level. Once the path reaches a state boundary, compute the best route from state to state. Once you know your entrances and exits from the states, then plan the inter-state routes. Again, this will work but may result in a sub-optimal solution.

Taken to the extreme, the abstraction could be at the country level: travel from the United States to Canada. Clearly, there comes a point where the abstraction becomes so coarse as to be effectively useless.

We want to adopt a similar abstraction strategy for computer game path-finding. We could use A* on a complete 1000×1000 map – but that represents a potentially huge search space. Abstraction can be used to reduce this dramatically. Consider each 10×10 block of the map as being a "city". Now we can search in a map of 100×100 cities. For each city, we know the city entrances and the costs of crossing the city for all the entrance pairs. We also know how to travel between cities. The problem then reduces to three steps:

- Start node: Within the block containing the start node, find the optimal path to the borders of the block.
- Search at the block level (100×100 blocks) for the optimal path from the block containing the start node to the block containing the goal node.
- Goal node: Within the block containing the goal node, find the optimal path from the border of the block to the goal.

The result is a much faster search giving nearly optimal solutions. Further, the abstraction is topology independent; there is no need for a level designer to manually break the grid into high-level features or annotate it with way-points.

1.2 Contributions

The contributions of this paper include:

1. HPA*, a new hierarchical path-finding algorithm (including pseudo-code and source code) that is domain-independent and works well for static and dynamic terrain topologies.

2. Experimental results for hierarchical search on a variety of games mazes (from BioWare's BALDUR'S GATE), showing up to a 10-fold speed improvement in exchange for a 1% degradation in path quality.

3. Variations on the hierarchical search idea appear to be in use by several game companies, although most of their algorithmic details are not public. To the best of our knowledge, this is the first scientific study of using hierarchical A* in the domain of commercial computer games.

Section 2 contains a brief overview of the background literature. Section 3 presents our new approach to hierarchical A*, and its performance is evaluated in Section 4. Section 5 presents our conclusions and topics for further research. Appendix A provides the pseudo-code for our algorithm.

2 Literature Review

The first part of this section summarizes hierarchical approaches used for path-finding in commercial games. The second part reviews related work in a more general context, including applications to other grid domains such as robotics.

Path-finding using a two-level hierarchy is described in [5]. The author provides only a high-level presentation of the approach. The problem map is abstracted into clusters such as rooms in a building or square blocks on a field. An abstract action crosses a room from the middle of an entrance to another. This method has similarities to our work. First, both approaches partition the problem map into clusters such as square blocks. Second, abstract actions are block crossings (as opposed to going from one block center to another block center). Third, both techniques abstract a block entrance into one transition point (in fact, we allow either one or two points). This leads to fast computation but gives up the solution optimality. There are also significant differences between the two approaches. We extend our hierarchy to several abstraction levels and do this abstraction in a domain independent way. We also pre-compute and cache optimal distances for block crossing, reducing the costs of the on-line computation.

Another important hierarchical approach for path-finding in commercial games uses *points of visibility* [6]. This method exploits the domain local topology to define an abstract graph that covers the map efficiently. The graph nodes represent the corners of convex obstacles. For each node, edges are added to all the nodes that can be seen from the current node (i.e., the can be connected with a straight line).

This method provides solutions of good quality. It is particularly useful when the number of obstacles is relatively small and they have a convex polygonal shape (i.e., building interiors). The efficiency of the method decreases when many obstacles are present and/or their shape is not a convex polygon. Consider the case of a map containing a forest, which is a dense collection of small size obstacles. Modeling such a topology with points of visibility would result in a large graph (in terms of number of nodes and edges) with short edges. Therefore, the key idea of traveling long distances in a single step wouldn't be efficiently exploited. When the problem map contains concave or curved shapes, the method either has poor performance or needs sophisticated engineering to build the graph efficiently. In fact, the need for algorithmic or designer assistance to create the graph is one of the disadvantages of the method. In contrast, our approach works for many kinds of maps and does not require complex domain analysis to perform the abstraction.

The *navigation mesh* (aka. NavMesh) is a powerful abstraction technique useful for 2D and 3D maps. In a 2D environment, this approach covers the unblocked area of a map with a (minimal) set of convex polygons. A method for building a near optimal NavMesh is presented in [11]. This method relaxes the condition of the minimal set of polygons and builds a map coverage much faster.

Besides commercial computer games, path-finding has applications in many research areas. Path-finding approaches based on topological abstraction that have been explored in robotics domains are especially relevant for the work described in this paper. *Quadtrees* [8] have been proposed as a way of doing hierarchical map decomposition. This method partitions a map into square blocks with different sizes so that a block contains either only walkable cells or only blocked cells. The problem map is initially partitioned into 4 blocks. If a block contains both obstacle cells and walkable cells, then it is further decomposed into 4 smaller blocks, and so on. An action in this abstracted framework is to travel between the centers of two adjacent blocks. Since the agent always goes to the middle of a box, this method produces sub-optimal solutions.

To improve the solution quality, quadtrees can be extended to *framed quadtrees* [1, 12]. In framed quadtrees, the border of a block is augmented with cells at the highest resolution. An action crosses a block between any two border cells. Since this representation permits many angles of direction, the solution quality improves significantly. On the other hand, framed quadtrees use more memory than quadtrees.

Framed quadtrees are more similar to our work than quadtrees, since we also use block crossings as abstract actions. However, we don't consider *all* the cells on the block border as entrance points. We reduce the number of block entrance

points by abstracting an entrance into one or two such points. Moreover, our approach allows blocks to contain obstacles. This means that the distance between two transition points is not necessarily linear. For this reason we have to compute optimal paths between entrance points placed on the border of the same block.

A multi-level hierarchy has been used to enhance the performance of multiple goal path-planning in a MDP (Markov Decision Process) framework [4]. The problem posed is to efficiently learn near optimal policies $\pi^*(x, y)$ to travel from x to y for all pairs (x, y) of map locations. The number of policies that have to be computed and stored is quadratic in the number of map cells. To improve both the memory and time requirements (for the price of losing optimality), a multi-level structure is used—a so called *airport hierarchy*. All locations on the problem map are *airports* that are assigned to different hierarchical levels. The strategy for travelling from x to y is similar to traveling by plane in the real world. First, travel to bigger and bigger airports until we reach an airport that is big enough to have a connection to the area that contains the destination. Second, go down in the hierarchy by travelling to smaller airports until the destination is reached. This approach is very similar to the strategy outlined in Section 1.1.

An analysis of the nature of path-finding in various frameworks is performed in [7]. The authors classify path-finding problems based on the type of the results that are sought, the environment type, the amount of information available, etc. Challenges specific to each problem type and solving strategies such as replanning and using dynamic data structures are briefly discussed.

A hierarchical approach for shortest path algorithms that has similarities with HPA* is analysed in [9]. This work decomposes an initial problem graph into a set of fragment sub-graphs and a global boundary sub-graph that links the fragment sub-graphs. Shortest paths are computed and cached for future use, similarly to the caching that HPA* performs for cluster traversal routes. The authors analyse what shortest paths (i.e., from which sub-graphs) to cache, and what information to keep (i.e., either complete path or only cost) for best performance when limited memory is available.

Another technique related to HPA* is Hierarchical A* [2], which also uses hierarchical representations of a space with the goal of reducing the overall search effort. However, the way that hierarchical representations are used is different in these two techniques. While our approach uses abstraction to structure and enhance the representation of the search space, Hierarchical A* is a method for automatically generating domain-independent heuristic state evaluations. In single-agent search, a heuristic function that evaluates the distance from a state to the goal is used to guide the search process. The quality of such a function greatly

affects the quality of the whole search algorithm. Starting from the initial space, Hierarchical A* builds a hierarchy of abstract spaces until an abstract one-state space is obtained. When building the next abstract space, several states of the current space are grouped to form one abstract state in the next space. In this hierarchy, an abstract space is used to compute a heuristic function for the previous space.

3 Hierarchical Path-finding

Our hierarchical approach implements the strategy described in Section 1.1. Searching for an abstract solution in our hierarchical framework is a three step process called *on-line search*. First, travel to the border of the neighborhood that contains the start location. Second, search for a path from the border of the start neighborhood to the border of the goal neighborhood. This is done using on an abstract level, where search is simpler and faster. An action travels across a relatively large area, with no need to deal with the details of that area. Third, complete the path by traveling from the border of the goal neighborhood to the goal position.

The abstracted graph for on-line search is built using information extracted from the problem maze. We discuss in more detail how the framework for hierarchical search is built (pre-processing) and how it is used for path finding (on-line search). Initially we focus on building a hierarchy two levels: one low level and one abstract level. Adding more hierarchical levels is discussed at the end of this section. We illustrate how our approach works on the small 40×40 map shown in Figure 1 (a).

3.1 Pre-processing a Grid

The first step in building the framework for hierarchical search defines a topological abstraction of the maze. We use this maze abstraction to build an abstract graph for hierarchical search.

The topological abstraction covers the maze with a set of disjunct rectangular areas called *clusters*. The bold lines in Figure 1 (b) show the abstract clusters used for topological abstraction. In this example, the 40×40 grid is grouped into 16 clusters of size 10×10. Note that no domain knowledge is used to do this abstraction (other than, perhaps, tuning the size of the clusters).

For each border line between two adjacent clusters, we identify a (possibly empty) set of entrances connecting them. An entrance is a maximal obstacle-free

segment along the common border of two adjacent clusters c_1 and c_2, formally defined as below. Consider the two adjacent lines of tiles l_1 and l_2, one in each cluster, that determine the border edge between c_1 and c_2. For a tile $t \in l_1 \cup l_2$, we define symm(t) as being the symmetrical tile of t with respect to the border between c_1 and c_2. Note that t and symm(t) are adjacent and never belong to the same cluster. An entrance e is a set of tiles that respects the following conditions:

- The border limitation condition: $e \subset l_1 \cup l_2$. This condition states that an entrance is defined along and cannot exceed the border between two adjacent clusters.
- The symmetry condition: $\forall t \in l_1 \cup l_2 : t \in e \Leftrightarrow$ symm$(t) \in e$.
- The obstacle free condition: an entrance contains no obstacle tiles.
- The maximality condition: an entrance is extended in both directions as long as the previous conditions remain true.

Figure 2 shows a zoomed picture of the upper-left quarter of the sample map. The picture shows details on how we identify entrances and use them to build the abstracted problem graph. In this example, the two clusters on the left side are connected by two entrances of width 3 and of width 6 respectively. For each entrance, we define one or two *transitions*, depending on the entrance width. If the width of the entrance is less than a predefined constant (6 in our example), then we define one transition in the middle of the entrance. Otherwise, we define two transitions, one on each end of the entrance.

We use transitions to build the abstract problem graph. For each transition we define two nodes in the abstract graph and an edge that links them. Since such an edge represents a transition between two clusters, we call it an *inter-edge*. Inter-edges always have length 1. For each pair of nodes inside a cluster, we define an edge linking them, called an *intra-edge*. We compute the length of an intra-edge by searching for for an optimal path inside the cluster area.

Figure 2 shows all the nodes (light grey squares), all the inter-edges (light grey lines), and part of the intra-edges (for the top-right cluster). Figure 3 shows the details of the abstracted internal topology of the cluster in the top-right corner of Figure 2. The data structure contains a set of nodes as well as distances between them. We define the distance as 1 for a straight transition and 1.42 [1] for a diagonal

[1]The generic path-finding library that we used in our experiments utilizes this value for approximating $\sqrt{2}$. A slightly more appropriate approximation would probably be 1.41.

transition. We only cache distances between nodes and discard the actual optimal paths corresponding to these distances. If desired, the paths can also be stored, for the price of more memory usage. See Section 3.2.2 for a discussion.

Figure 4 (a) shows the abstract graph for our running example. The picture includes the result of inserting the start and goal nodes S and G into the graph (the dotted lines), which is described in the next sub-section. The graph has 68 nodes, including S and G, which can change for each search. At this level of abstraction, there are 16 clusters with 43 inter-connections and 88 intra-connections. There are 2 additional edges that link S and G to the rest of the graph. For comparison, the low-level (non-abstracted) graph contains $1,463$ nodes, one for each unblocked tile, and $2,714$ edges.

Once the abstract graph has been constructed and the intra-edge distances computed, the grid is ready to use in a hierarchical search. This information can be pre-computed (before a game ships), stored on disk, and loaded into memory at game run-time. This is sufficient for static (non-changing) grids. For dynamically changing grids, the pre-computed data has to be modified at run-time. When the grid topology changes (e.g., a bridge blows up), the intra- and inter-edges of the affected local clusters need to be re-computed.

3.2 On-line Search

The first phase of the on-line search connects the starting position S to the border of the cluster containing S. This step is completed by temporarily inserting S into the abstract graph. Similarly, connecting the goal position G to its cluster border is handled by inserting G into the abstract graph.

After S and G have been added, we use A* [10] to search for a path between S and G in the abstract graph. This is the most important part of the on-line search. It provides an abstract path, the actual moves from S to the border of S's cluster, the abstract path to G's cluster, and the actual moves from the border of G's cluster to G.

The last two steps of the on-line search are optional:

1. Path-refinement can be used to convert an abstract path into a sequence of moves on the original grid.

2. Path-smoothing can be used to improve the quality of the path-refinement solution.

The abstract path can be refined in a post-processing step to obtain a detailed path from S to G. For many real-time path-finding applications, the complete path is not needed—only the first few moves. This information allows the character to start moving in the right direction towards the goal. In contrast, A* must complete its search and generate the entire path from S to G before it can determine the first steps of a character.

Consider a domain where dynamic changes occur frequently (e.g., there are many mobile units travelling around). In such a case, after finding an abstract path, we can refine it gradually as the character navigates towards the goal. If the current abstract path becomes invalid, the agent discards it and searches for another abstract path. There is no need to refine the whole abstract path in advance.

3.2.1 Searching for an Abstract Path

To be able to search for a path in the abstract graph, S and G have to be part of the graph. The processing is the same for both start and goal and we show it only for node S. We connect S to the border of the cluster c that contains it. We add S to the abstract graph and search locally for optimal paths between S and each of the abstract nodes of c. When such a path exists, we add an edge to the abstract graph and set its weight to the length of the path. In Figure 4 we represent these edges with dotted lines.

In our experiments we assume that S and G change for each new search. Therefore, the cost of inserting S and G is added to the total cost of finding a solution. After a path is found, we remove S and G from the graph. However, in practice this computation can be done more efficiently. Consider a game when many units have to find a path to the same goal. In this case, we insert G once and re-use it. The cost of inserting G is amortized over several searches. In general, a cache can be used to store connection information for popular start and goal nodes.

After inserting S and G, the abstract graph can be used to search for an abstract path between S and G. We run a standard single-agent search algorithm such as A* on the abstract graph.

3.2.2 Path Refinement

Path refinement translates an abstract path into a low-level path. Each cluster crossing in the abstract path is replaced by an equivalent sequence of low-level moves.

If the cluster pre-processing cached these move sequences attached to the intra-edges, then refinement is simply a table look-up. Otherwise, we perform small searches inside each cluster along the abstract path to re-discover the optimal local paths. There are two factors that limit the complexity of the refinement search. First, abstract solutions are guaranteed to be correct, provided that the environment does not change after finding an abstract path. This means that we never have to backtrack and re-plan for correcting the abstract solution. Second, the initial search problem has been decomposed into several very small searches (one for each cluster on the abstract path), with low complexity.

3.2.3 Path Smoothing

The topological abstraction phase defines only one transition point per entrance. While this is efficient, it gives up the optimality of the computed solutions. Solutions are optimal in the abstract graph but not necessarily in the initial problem graph.

To improve the solution quality (i.e., length and aesthetics), we perform a post-processing phase for path smoothing. Our technique for path smoothing is simple, but produces good results. The main idea is to replace local sub-optimal parts of the solution by straight lines. We start from one end of the solution. For each node in the solution, we check whether we can reach a subsequent node in the path in a straight line. If this happens, then the linear path between the two nodes replaces the initial sub-optimal sequence between these nodes.

3.3 Experimental Results for Example

The experimental results for our running example are summarized in the first two rows of Table 1. L-0 represents running A* on the low-level graph (we call this level 0). L-1 uses two hierarchy levels (i.e., level 0 and level 1), and L-2 uses three hierarchy levels (i.e., level 0, level 1, and level 2). The meaning of the last row, labeled L-2, is described in Section 3.5.

Low-level (original grid) search using Manhattan distance as the heuristic has poor performance. Our example has been chosen to show a worst-case scenario. Without abstraction, A* will visit all the unblocked positions in the maze. The search expands $1,462$ nodes. The only factor that limits the search complexity is the maze size. A larger map with a similar topology represents a hard problem for A*.

The performance is greatly improved by using hierarchical search. When inserting S into the abstract graph, it can be linked to only one node on the border of the starting cluster. Therefore we add one node (corresponding to S) and one edge that links S to the only accessible node in the cluster. Finding the edge cost uses a search that expands 8 nodes. Inserting G into the graph is identical.

A* is used on the abstracted graph to search for a path between S and G. Searching at level 1 also expands all the nodes of the abstract graph. The problem is also a worst-case scenario for searching at level 1. However, this time the search effort is much reduced.

The main search expands 67 nodes. In addition, inserting S and G expands 16 nodes. In total, finding an abstract path requires 83 node expansions. This effort is enough to provide a solution for this problem—the moves from S to the edge of its cluster and the abstract path from the cluster edge to G. If desired, the abstract path can be refined, partially or completely, for additional cost. The worst case is when we have to refine the path completely and no actual paths for intra-edges were cached. For each intra-edge (i.e., cluster crossing) in the path, we perform a search to compute a corresponding low-level action sequence. There are 12 such small searches, which expand a total of 145 nodes.

3.4 Adding Levels of Hierarchy

The hierarchy can be extended to several levels, transforming the abstract graph into a *multi-level* graph. In a multi-level graph, nodes and edges have labels showing their level in the abstraction hierarchy. We perform path-finding using a combination of small searches in the graph at various abstraction levels. Additional levels in the hierarchy can reduce the search effort, especially for large mazes. See Appendix A.2.2 for details on efficient searching in a multi-level graph. To build a multi-level graph, we structure the maze abstraction on several levels. The higher the level, the larger the clusters in the maze decomposition. The clusters for level l are called l-clusters. We build each new level on top of the existing structure. Building the 1-clusters has been presented in Section 3.1. For $l \geq 2$, an l-cluster is obtained by grouping together $n \times n$ adjacent $(l-1)$-clusters, where n is a parameter.

Nodes on the border of a newly created l-cluster update their level to l (we call these l-nodes). Inter-edges that make transitions between l-clusters also increase their level to l (we call these l-inter-edges).

We add intra-edges with level l (i.e., l-intra-edges) for pairs of communicating l-nodes placed on the border of the same l-cluster. The weight of such an edge

is the length of the shortest path that connects the two nodes within the cluster, using only $(l-1)$- nodes and edges. More details are provided in Section A.2.2.

Inserting S into the graph iteratively connects S to the nodes on the border of the l-cluster that contains it, with l increasing from 1 to the maximal abstraction level. Searching for a path between S and a l-node is restricted to level $l-1$ and to the area of the current l-cluster that contains S. We perform an identical processing for G too.

The way we build the abstract graph ensures that we always find the same solution, no matter how many abstract levels we use. In particular, adding a new level $l \geq 2$ to the graph does not diminish the solution quality. Here we provide a brief intuitive explanation rather than a formal proof of this statement. A new edge added at level l corresponds to an existing shortest path at level $l-1$. The weight of the new edge is set to the cost of the corresponding path. Searching at level l finds the same solution as searching at level $l-1$, only faster.

In our example, adding an extra level with $n = 2$ creates 4 large clusters, one for each quarter of the map. The whole of Figure 2 is an example of a single 2-cluster. This cluster contains 2×2 1-clusters of size 10×10. Besides S, the only other 2-node of this cluster is the one in the bottom-left corner. Compared to level 1, the total number of nodes at the second abstraction level is reduced even more. Level 2, where the main search is performed, has 14 nodes (including S and G). Figure 4 (b) shows level 2 of the abstract graph. The edges pictured as dotted lines connect S and G to the graph at level 2.

Abstraction level 2 is a good illustration of how the pre-processing solves local constraints and reduces the search complexity in the abstract graph. The 2-cluster shown in Figure 2 is large enough to contain the large dead end "room" that exists in the local topology. At level 2, we avoid any useless search in this "room" and go directly from S to the exit in the bottom-left corner.

After inserting S and G, we are ready to search for a path between S and G. We search only at the highest abstraction level. Since start and goal have the highest abstraction level, we will always find a solution, assuming that one exists. The result of this search is a sequence of nodes at the highest level of abstraction. If desired, the abstract path can repeatedly be refined until the low-level solution is obtained.

3.5 Experimental Results for Example with 3-Level Hierarchy

The third row of Table 1 shows numerical data for our running example with a 3-Level hierarchy (i.e., with three levels: $L-0$, $L-1$, and $L-2$).

As shown in Section 3.3, connecting S and G to the border of their 1-clusters expands 16 nodes in total. Similarly, we now connect S and G to the border of their 2-clusters. These searches at level 1 expand 3 nodes for S and 22 nodes for G.

The main search at level 2 expands only 7 nodes. No nodes other than the ones in the abstract path are expanded. This is an important improvement, if we consider that search in the level 1 graph expanded all nodes in the graph. In total, finding an abstract solution in the extended hierarchy requires 48 nodes.

It is worth to remark that, after adding a new abstraction level, the cost for inserting S and G dominates the main search cost. This illustrates the general characteristic of the method that the cost for inserting S and G increases with the number of levels, whereas the main search becomes simpler. Finding a good trade-off between these searches is important for optimizing the performance.

Table 1 also shows the costs for complete solution refinement. Refining the solution from level 2 to level 1 expands 16 nodes and refining from level 1 to level 0 expands 145 nodes, for a total of 161 nodes.

3.6 Storage Analysis

Besides the computational speed, the amount of storage that a method uses for path-finding is another important performance indicator. Two main factors influence the amount of memory that our hierarchical approach uses: the size of the problem graph and the size of the open list used by A*. We discuss these two factors in more detail in the rest of this section. For the graph storage, we include both an empirical analysis and a worst-case theoretical discussion.

3.6.1 Graph Storage Requirements

Table 2 shows the average size of the problem graph for our BALDUR'S GATE test suite. See Section 4 for details about this data set and settings such as cluster sizes, or edge definition in the original problem graph. We compare the original low-level graph to the abstract graphs in hierarchies with one, two, and three abstract levels (not counting level 0). In the table we show the number of nodes N, the number of inter-edges E_1, and the number of intra-edges E_2. For the multi-level graphs, we show both the total numbers and the numbers for each level L_i, $i \in \{1, 2, 3\}$.

The data show that the storage overhead of the abstract graph is small compared to the size of the original problem graph. Adding a new graph level updates

the level of some existing nodes and inter-edges without creating any new objects of these types. The only overhead consists of the new intra-edges that a level creates. In our data set, we add at most $1,846$ intra-edges (when three abstract levels are defined) to an initial graph having $4,469$ nodes and $16,420$ edges. Assuming that a node and an edge occupy about the same amount of memory, we obtain an overhead of 8.83%.

The way that the overhead translates in terms of memory bytes is highly dependant on factors such as implementation, compiler optimizations, or size of the problem map. For instance, if the map size is at most 256×256, then storing the coordinates of a graph node takes two bytes. More memory is necessary for larger maps.

Since abstract nodes and edges are labeled by their level, the memory necessary to store an element might be larger in the abstract graph than in the initial graph. This additional requirement, called the level overhead, can be as little as 2 bits per element, corresponding to a largest possible number of levels of 4. Since most compilers round the bit-size of objects to a multiple of 8, the level overhead could actually not exist in practice.

The storage utilization can be optimized by keeping in memory (e.g., the cache) only those parts of the graph that are necessary for the current search. In the hierarchical framework, we need only the sub-graph corresponding to the level and the area where the current search is performed. For example, when the main abstract search is performed, we can drop the low-level problem graph, greatly reducing the memory requirements for this search.

The worst case scenario for a cluster is when blocked tiles and free tiles alternate on the border, and any two border nodes can be connected to each other. Assume the size of the problem maze is $m \times m$, the maze is decomposed into $c \times c$ clusters, and the size of a cluster is $n \times n$. In the worst case, we obtain $4n/2 = 2n$ nodes per cluster. Since each pair of nodes defines an intra-edge, the number of intra-edges for a cluster is $2n(2n-1)/2 = n(2n-1)$. This analysis is true for clusters in the middle of the maze. We do not define abstract nodes on the maze edges, so marginal clusters have a smaller number of abstract nodes. For the cluster in a maze corner, the number of nodes is n and the number of intra-edges is $n(n-1)/2$. For a cluster on a maze edge, the number of nodes is $1.5n$ and the number of intra-edges is $1.5n(1.5n-1)/2$. There are 4 corner clusters, $4c-8$ edge clusters, and $(c-2)^2$ middle clusters. Therefore, the total number of abstract nodes is $2m(c-1)$. The total number of intra-edges is $n(c-2)^2(2n-1) + 2(n-1) + 3(c-2)(1.5n-1)$. The number of inter-edges is $m(c-1)$.

3.6.2 Storage for the A* Open List

Since hierarchical path-finding decomposes a problem into a sum of small searches, the average size of open in A* is smaller in hierarchical search than in low-level search. Table 3 compares the average length of the open list in low-level search and hierarchical search. The average is performed over all searches described in Section 4.1, without refining the results after the solution length. The data shows a three-fold reduction of the list size between the low-level search and the main search in the abstracted framework.

4 Experimental Results

4.1 Experimental Setup

Experiments were performed on a set of 120 maps extracted from BioWare's game BALDUR'S GATE, varying in size from 50×50 to 320×320. For each map, 100 searches were run using randomly generated S and G pairs where a valid path between the two locations existed.

The atomic map decomposition uses *octiles*. Octiles are tiles that define the adjacency relationship in 4 straight and 4 diagonal directions. The cost of vertical and horizontal transitions is 1. Diagonal transitions have the cost set to 1.42. We do not allow diagonal moves between two blocked tiles. Entrances with width less than 6 have one transition. For larger entrances we generate two transitions.

The code was implemented using the University of Alberta Path-finding Code Library (`http://www.cs.ualberta.ca/~games/pathfind`). This library is used as a research tool for quickly implementing different search algorithms using different grid representations. Because of its generic nature, there is some overhead associated with using the library. All times reported in this paper should be viewed as generous upper bounds on a custom implementation.

The timings were performed on a 800 MHz Pentium III with 3 GB of memory. The programs were compiled using gcc version 2.96, and were run under Red Hat Linux version 7.2.

4.2 Analysis

Figure 5 compares low-level A* to abstract search on hierarchies with the maximal level set to 1, 2, and 3. The left graph shows the number of expanded nodes and the right graph shows the time. For hierarchical search we display the total effort,

which includes inserting S and G into the graph, searching at the highest level, and refining the path. The real effort can be smaller since the cost of inserting S or G can be amortized for many searches, and path refinement is not always necessary. The graphs show that, when complete processing is necessary, the first abstraction level is good enough for the map sizes that we used in this experiment. We assume that, for larger maps, the benefits of more levels would be more significant. The complexity reduction can become larger than the overhead for adding the level. As we show next, more levels are also useful when path refinement is not necessary and S or G can be used for several searches.

Even though the reported times are for a generic implementation, it is important to note that for any solution length the appropriate level of abstraction was able to provide answers in less than 10 milliseconds on average. Through length 400, the average time per search was less than 5 milliseconds on a 800 MHz machine.

A* is slightly better than HPA* when the solution length is very small. A small solution length usually indicates an easy search problem, which A* solves with reduced effort. The overhead of HPA* (e.g., for inserting S and G) is in such cases larger than the potential savings that the algorithm could achieve. A* is also better when S and G can be connected through a "straight" line on the grid. In this case, using the Euclidian distance as heuristic provides perfect information, and A* expands no nodes other than those that belong to the solution.

Figure 6 shows how the total effort for hierarchical search is composed of the abstract effort, the effort for inserting S and G, and the effort for solution refinement. The cost for finding an abstract path is the sum of only the main cost and the cost for inserting S and G. When S or G are reused for many searches, only part of this cost counts for the abstract cost of a problem. Considering these, the figure shows that finding an abstract path becomes easier in hierarchies with more levels.

Figure 7 shows the solution quality. We compare the solutions obtained with hierarchical path-finding to the optimal solutions computed by low-level A*. We plot the error before and after path-smoothing. The error measures the overhead in percents and is computed with the following formula:

$$e = \frac{hl - ol}{ol} \times 100$$

where hl is the length of the solution found with HPA*, and ol is the length of the optimal solution found with A*. The error is independent of the number of hierarchical levels. The only factor that generates sub-optimality is not considering all

the possible transitions for an entrance.

The cluster size is a parameter that can be tuned. We ran our performance tests using 1-clusters with size 10×10. This choice at level 1 is supported by the data presented in Figure 8. This graph shows how the average number of expanded nodes for an abstract search changes with varying the cluster size. While the main search reduces with increasing cluster size, the cost for inserting S and G increases faster. The expanded node count reaches a minimum at cluster size 10.

For higher levels, an l-cluster contains 2×2 $(l-1)$-clusters. We used this small value since, when larger values are used, the cost for inserting S and G increases faster than the reduction of the main search. This tendency is especially true on relatively small maps, where smaller clusters achieve good performance and the increased costs for using larger clusters may not be amortized. The overhead of inserting S and G results from having to connect S and G to many nodes placed on the border of a large cluster. The longer the cluster border, the more nodes to connect to. We ran similar tests on randomly generated maps. The main conclusions were similar but, because of lack of space, we do not discuss the details in this paper.

5 Conclusions and Future Work

Despite the importance and the amount of work done in path-finding, there are not many detailed publications about hierarchical path-finding in commercial games.

In this paper we have presented a hierarchical technique for efficient near-optimal path-finding. Our approach is domain-independent, easy to apply and works well for different kinds of map topologies. The method adapts to dynamically changing environments. The hierarchy can be extended to several abstraction levels, making it scalable for large problem spaces. We tested our program using maps extracted from a real game, obtaining near-optimal solutions significantly faster than low-level A*.

We have many ideas for future work in hierarchical path-finding. We plan to optimize the way that we insert S and G into the abstract graph. As Figure 6 shows, these costs increase significantly with adding a new abstraction layer. One strategy for improving the performance is to connect S only to a sparse subset of the nodes on the border, maintaining the completeness of the abstract graph. For instance, if each "unconnected" node (i.e., a node on the border to which we did not try to connect S) is reachable in the abstract graph from a "connected" node (i.e., a node on the border to which we have connected S), then the completeness

is preserved. Another idea is to consider for connection only border nodes that are on the direction of G. However, this last idea does not guarantee the completeness and it is hard to evaluate the benefits beforehand. If the search fails because of the graph incompleteness, we have to perform it again with the subset of border nodes gradually enlarged.

The clustering method that we currently use is simple and produces good results. However, we also want to explore more sophisticated clustering methods. An application-independent strategy is to automatically minimize some of the clustering parameters such as number of abstract clusters, cluster interactions, and cluster complexity (e.g., the percentage of internal obstacles).

6 Acknowledgement

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE). We thank all members of the Path-finding Research Group at the University of Alberta. Markus Enzenberger and Yngvi Björnsson wrote a generic path-finding library that we used in our experiments. BioWare kindly gave us access to the BALDUR'S GATE maps.

References

[1] D. Z. Chen, R. J. Szczerba, and J. J. Urhan Jr. Planning Conditional Shortest Paths Through an Unknown Environment: A Framed-Quadtree Approach. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and System Human Interaction and Cooperation*, volume 3, pages 33–38, 1995.

[2] R. Holte, M. Perez, R. Zimmer, and A. MacDonald. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *Proceedings AAAI-96*, pages 530–535, 1996.

[3] R. Korf. Depth-first Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 97:97–109, 1985.

[4] A. Moore, L. Baird, and L. Kaelbling. Multi-Value-Functions: Efficient Automatic Action Hierarchies for Multiple Goal MDPs. In *Proceedings*

of the International Joint Conference on Artificial Intelligence (IJCAI '99), 1999.

[5] S. Rabin. A* Aesthetic Optimizations. In Mark Deloura, editor, *Game Programming Gems*, pages 264–271. Charles River Media, 2000.

[6] S. Rabin. A* Speed Optimizations. In Mark Deloura, editor, *Game Programming Gems*, pages 272–287. Charles River Media, 2000.

[7] B. Reese and B. Stout. Finding a pathfinder. `http://citeseer.nj.nec.com/reese99finding.html`.

[8] H. Samet. An Overview of Quadtrees, Octrees, and Related Hierarchical Data Structures. NATO ASI Series, Vol. F40, 1988.

[9] S. Shekhar, A. Fetterer, and B. Goyal. Materialization Trade-Offs in Hierarchical Shortest Path Algorithms. In *Symposium on Large Spatial Databases*, pages 94–111, 1997.

[10] B. Stout. Smart Moves: Intelligent Pathfinding. *Game Developer Magazine*, October/November 1996.

[11] P. Tozour. Building a Near-Optimal Navigation Mesh. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 171–185. Charles River Media, Inc., 2002.

[12] A. Yahja, A. Stentz, S. Singh, and B. Brummit. Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments. In *Proceedings, IEEE Conference on Robotics and Automation, (ICRA)*, Leuven, Belgium, May 1998.

A APPENDIX

In this appendix we provide low-level details about our hierarchical path-finding technique, including the main functions in pseudo-code. The code can be found at the web site `http://www.cs.ualberta.ca/~adib/`. First we adress the pre-processing, and next the on-line search.

A.1 Pre-processing

Figure 9 summarizes the pre-processing. The main method is $preprocessing()$, which abstracts the problem maze, builds a graph with one abstract level and, if desired, adds more levels to the graph.

A.1.1 Abstracting the Maze and Building the Abstract Graph

At the initial stage, the maze abstraction consists of building the 1-clusters and the entrances between clusters. Later, when more levels are added to the hierarchy, the maze is further abstracted by computing clusters of superior levels. In the method $abstractMaze()$, $C[1]$ is the set of 1-clusters, and E is the set of all entrances defined for the map.

The method $buildGraph()$ creates the abstract graph of the problem. First it creates the nodes and the inter-edges, and next builds the intra-edges. The method $newNode(e, c)$ creates a node contained in cluster c and placed at the middle of entrance e. For simplicity, we assume we have one transition per entrance, regardless of the entrance width. The methods $getCluster1(e, l)$ and $getCluster2(e, l)$ return the two adjacent l-clusters connected by entrance e. We use the methods $addNode(n, l)$ to add node n to the graph and set the node level to l, and $addEdge(n_1, n_2, w, l, t)$ to add an edge between nodes n_1 and n_2. Parameter w is the weight, l is the level, and $t \in \{\text{INTER}, \text{INTRA}\}$ shows the type (i.e., inter-edge or intra-edge) of the edge.

The last part of the method $buildGraph()$ adds the intra-edges. The method $searchForDistance(n_1, n_2, c)$ searches for a path between two nodes and returns the path cost. This search is optimized as shown in Section A.2.2.

A.1.2 Creating Additional Graph Levels

The hierarchical levels of the multi-level abstract graph are built incrementally. Level 1 has been built at the previous phase. Assuming that the highest current

level is $l-1$, we build level l by calling the method $addLevelToGraph(l)$. We group clusters at level $l-1$ to form a cluster at level l (the method $buildClusters(l)$, $l > 1$). $C[l]$ is the set of l-clusters. The last part of the method $addLevelToGraph()$ adds new intra-edges to the graph.

A.2 On-line Search

A.2.1 Finding an Abstract Solution

Figure 10 summarizes the steps of the on-line search.

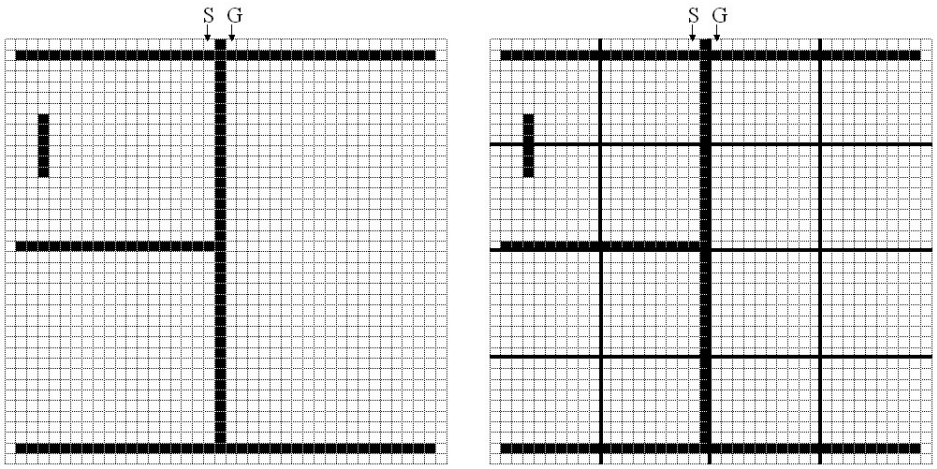
The main method is $hierarchicalSearch(S, G, maxLevel)$, which performs the on-line search. First we insert S and G into the abstract graph, using the method $insertNode(node, level)$. The method $connectToBorder(n, c)$ adds edges between node n and the nodes placed on the border of cluster c that are reachable from n. We insert S, G into the multi-level graph using the method $insertNode()$. $determineCluster(n, l)$ returns the l-cluster that contains node n.

The method $searchForPath(S, G, maxLevel)$ performs a search at the highest abstraction level to find an abstract path from S to G. If desired, we refine the path to a low-level representation using the method $refinePath(absPath)$. Finally, the method $smoothPath(llPath)$ improves the quality of the low-level solution.

A.2.2 Searching in the Multi-Level Graph

In a multi-level graph, search can be performed at various abstraction levels. Searching at level l reduces the search effort by exploring only a small subset of the graph nodes. The higher the level, the smaller the part of the graph that can potentially be explored. When searching at a certain level l, the rules that apply for node expansion are the following. First, we consider only nodes having level greater than or equal to l. Second, we consider only intra-edges having level l and inter-edges having level $\geq l$.

The search space can be further reduced by ignoring the nodes outside a given cluster. This is useful in situations such as connecting S or G to the border of their clusters, connecting two nodes placed on the border of the same cluster, or refining an abstract path.



(a) (b)

Figure 1: (a) The 40×40 maze used in our example. The obstacles are painted in black. S and G are the start and the goal nodes. (b) The bold lines show the boundaries of the 10×10 clusters.

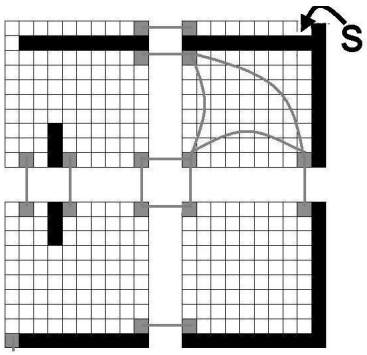
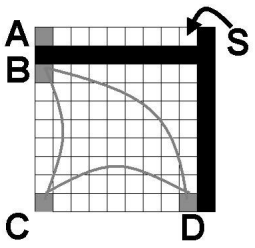
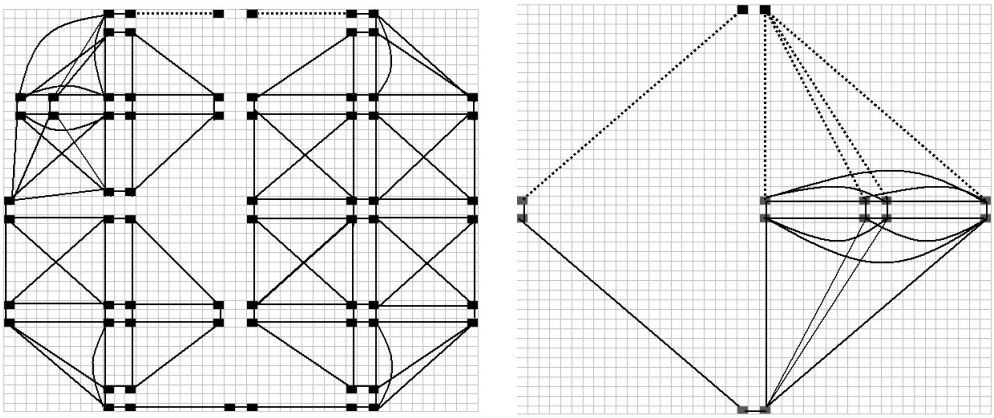


Figure 2: Abstracting the top-left corner of the maze. All abstract nodes and inter-edges are shown in light grey. For simplicity, intra-edges are shown only for the top-right cluster.



	A	B	C	D
A		∞	∞	∞
B			7.00	10.94
C				8.00
D				

Figure 3: Cluster-internal path information.



(a) (b)

Figure 4: (a) The abstract problem graph in a hierarchy with one low level and one abstract level. (b) Level 2 of the abstract graph in the 3-Level hierarchy. The dotted edges connect S and G to the rest of each graph.

Search Technique	SG	Main	**Abstract**	Refinement
L-0	0	1,462	**1,462**	0
L-1	16	67	**83**	145
L-2	41	7	**48**	161

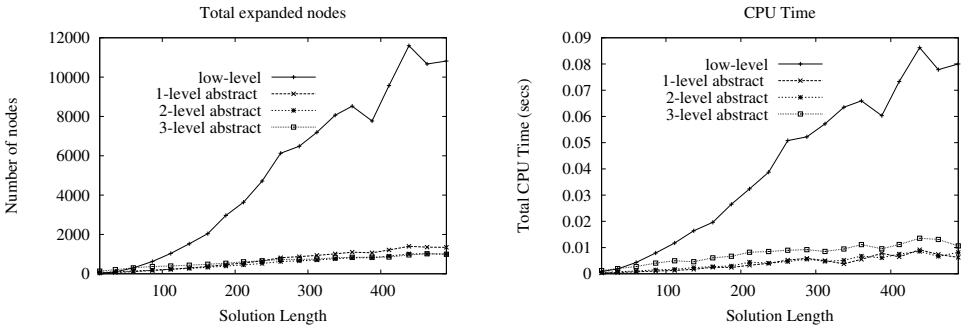
Table 1: Summary of results our running example. We show the number of expanded nodes. SG is the effort for inserting S and G into the graph. *Abstract* is the sum of the previous two columns. This measures the effort for finding an abstract solution. *Refinement* shows the effort for complete path-refinement.

	Graph 0	Graph 1		Graph 2			Graph 3			
		L_1	Total	L_1	L_2	Total	L_1	L_2	L_3	Total
N	**4,469**	367	**367**	186	181	**367**	186	92	89	**367**
E_1	**16,420**	198	**198**	100	98	**198**	100	50	48	**198**
E_2	**0**	722	**722**	722	662	**1,384**	722	622	462	**1,846**

Table 2: The average size of the problem graph in BALDUR'S GATE. Graph 0 is the initial low-level graph. Graph 1 represents a graph with one abstract level (L_1), Graph 2 has two abstract levels (L_1, L_2) , and Graph 3 has three abstract levels (L_1, L_2, L_3). N is the number of nodes, E_1 is the number of inter-edges, and E_2 is the number of intra-edges.

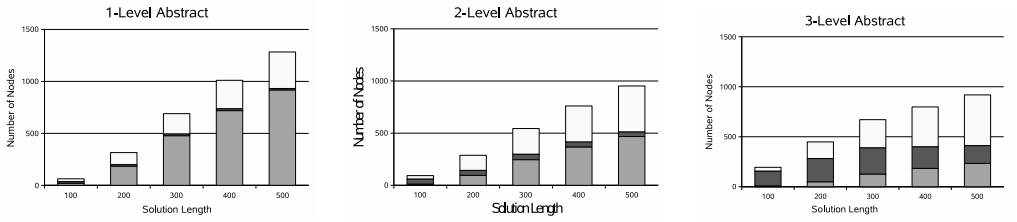
	Low level	Abstract		
		Main	SG	Refinement
Open Size	51.24	17.23	4.50	5.48

Table 3: Average size of the open list in A*. For hierarchical search, we show the average Open size for the main search, the SG search (i.e., search for inserting S and G into the abstract graph), and the refinement search.



(a) (b)

Figure 5: Low-level A* vs. hierarchical path-finding.



(a) (b) (c)

Figure 6: The effort for hierarchical search in hierarchies with one abstract level, two abstract levels, and three abstract levels. We show in what proportion the main effort, the SG effort, and the refinement effort contribute to the total effort. The gray part at the bottom of a data bar represents the main effort. The dark part in the middle is the SG effort. The white part at the top is the refinement effort.

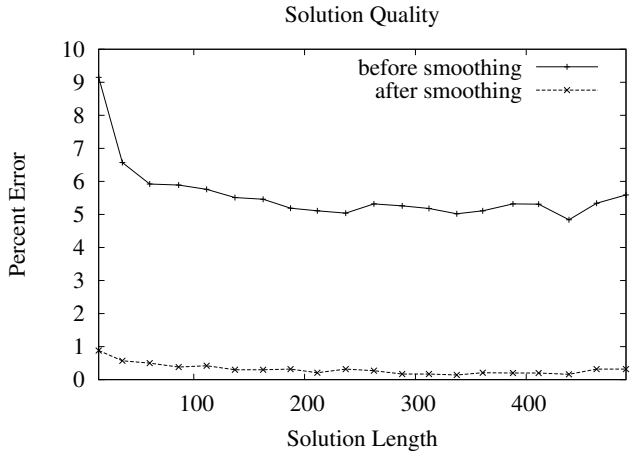


Figure 7: The solution quality.

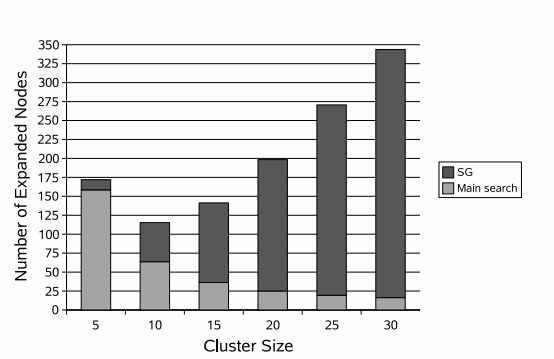


Figure 8: The seach effort for finding an abstract solution. SG represents the cost of inserting S and G. The main search finds an abstract path in the abstract graph.

```
void abstractMaze(void) {
   E = ∅;
   C[1] = buildClusters(1);
   for (each c1, c2 ∈ C[1]) {
      if (adjacent(c1, c2))
         E = E ∪ buildEntrances(c1, c2);
   }
}

void buildGraph(void) {
   for (each e ∈ E) {
      c1 = getCluster1(e, 1);
      c2 = getCluster2(e, 1);
      n1 = newNode(e, c1);
      n2 = newNode(e, c2);
      addNode(n1, 1);
      addNode(n2, 1);
      addEdge(n1, n2, 1, 1, INTER);
   }
   for (each c ∈ C[1]) {
      for (each n1, n2 ∈ N[c], n1 ≠ n2) {
         d = searchForDistance(n1, n2, c);
         if (d < ∞)
            addEdge(n1, n2, 1, d, INTRA);
      }
   }
}

void addLevelToGraph(int l) {
   C[l] = buildClusters(l);
   for (each c1, c2 ∈ C[l]) {
      if (adjacent(c1, c2) == false)
         continue;
      for (each e ∈ getEntrances(c1, c2)) {
         setLevel(getNode1(e), l);
         setLevel(getNode2(e), l);
         setLevel(getEdge(e), l);
      }
   }
   for (each c ∈ C[l])
      for (each n1, n2 ∈ N[c], n1 ≠ n2) {
         d = searchForDistance(n1, n2, c);
         if (d < ∞)
            addEdge(n1, n2, l, d, INTRA)
      }
}

void preprocessing(int maxLevel) {
   abstractMaze();
   buildGraph();
   for (l = 2; l ≤ maxLevel; l++)
      addLevelToGraph(l);
}
```

Figure 9: The pre-processing phase in pseudo-code. This phase builds the multi-level graph, except for S and G.

```
void connectToBorder(node s, cluster c) {
   l = getLevel(c);
   for (each n ∈ N[c])
      if (getLevel(n) < l)
         continue;
      d = searchForDistance(s, n, c);
      if (d < ∞)
         addEdge(s, n, d, l, INTRA);
}
void insertNode(node s, int maxLevel) {
   for (l = 1; l ≤ maxLevel; l++) {
      c = determineCluster(s, l);
      connectToBorder(s, c);
   }
   setLevel(s, maxLevel);
}
```

```
path hierarchicalSearch(node s, g, int l) {
   insertNode(s, l);
   insertNode(g, l);
   absPath = searchForPath(s, g, l);
   llPath = refinePath(absPath, l);
   smPath = smoothPath(llPath);
   return smPath;
}
```

Figure 10: The on-line processing in pseudo-code.