6 Automated Planning

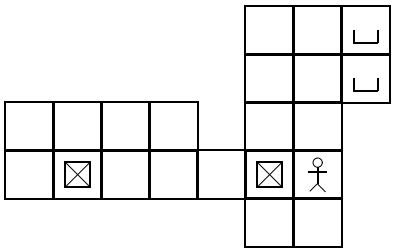
A complex form of problem solving. Same as problem solving search, except

- we now assume the states and actions have exploitable structure

Specifically, we assume each action only affects a small part of the state.

6.1 Example: Sokoban (game)

Move boxes around in a grid world by pushing them from behind. The goal is to get all of the boxes into containers, starting from the given initial state.



The actions are: move up, down, left, and right. However, there are constraints on how the boxes can be moved:

- the boxes must be pushed directly from behind
- can only push one box at a time

Note that a state is a complex description. However, the actions affect only a small portion of the state description at any one time.

How hard is Sokoban?

Really hard! In fact, it is PSPACE-complete (when problem size is measured with respect to the size of the state *description*, not the size of the state *space*).

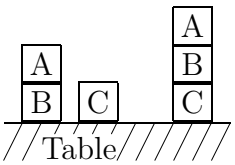
It is currently believed that PSPACE-complete is much harder than NP-complete, mainly because it encodes strictly more general problems. For example, alternating quantified Boolean satisfiability is PSPACE-complete:

- Does $\forall v_1 \exists v_2 \forall v_3 \cdots \exists v_n$ *CNF-formula*$(v_1, ..., v_n)$ evaluate to true?

This seems clearly harder than regular Boolean satisfiability:

- Does $\exists v_1 \exists v_2 \exists v_3 \cdots \exists v_n$ *CNF-formula*$(v_1, ..., v_n)$ evaluate to true?

6.2 Another example: Blocks world planning



- Actions: put box X on Y
  - can only move one box at a time
  - can only move top box from a stack
  - can always put block on table
- Initial state: full configuration
- Goal: not necessarily full configuration, can be just partial description. (E.g. get A on B and C on D.)

Finding an arbitray solution is easy: just put every block on the table and then stack them up again. However, finding the *shortest* solution is NP-hard.

6.3 General propositional planning

Planning is like a simple form of automated programming: find a sequence of instructions that transforms a given initial machine state to a goal state. However, it is the simplest possible form of automated programming: no loops, conditionals, recursion, subroutine calls, etc. But even still, general planning is hard: it is PSPACE-complete!

6.3.1 State

State: represented by an assignment of truth values to primitive propositions.

E.g. *Blocks world*

We can choose primitive propositions to be: AonT (A on table), AonB, AonC, BonT, BonA, etc. If we use 1 to denote true, and 0 to denote false, then the above initial state can be represented as vector of 0s and 1s:

A	A	A	B	B	B	C	C	C
T	B	C	T	A	C	T	A	B
0	1	0	1	0	0	1	0	0

6.3.2 Goal

Goal: a compound proposition γ. For example, AonB $\wedge$ BonC

If γ is a conjunction of primitive propositions or negations of primitive propositions, it can be represented as a vector of 0s, 1s, and '*'s ("don't care"s).

E.g. The goal AonB $\wedge$ BonC can be represented as

A T	A B	A C	B T	B A	B C	C T	C A	C B
*	1	*	*	*	1	*	*	*

6.3.3 Action

Action a: defined by a precondition π_a and a postcondition α_a.
If we assume that π_a and α_a are conjunctions of primitive propositions or their negations, then preconditions and postconditions can be represented as vectors (patterns). So, an action can be represented as a pair of vectors.

E.g. The action "put A on the table" can be represented as

	A T	A B	A C	B T	B A	B C	C T	C A	C B
pre-condition	*	*	*	*	0	*	*	0	*
post-condition	1	0	0	*	*	*	*	*	*

To apply an action, the preconditions have to be satisfied. The result of applying an action is determined by the postcondition and by the "frame assumption" (propositions that are not specified in the postcondition do not change their truth value).

6.4 Planning search

Any problem solving search algorithm from before can be applied to planning problems (e.g., DFS, A*, etc). Doing so requires the ability to determine which actions are applicable at a given state, and the ability to calculate the results of applying a particular action at a given state. Note however, that the standard problem solving search algorithms do not necessarily exploit any particular structure in the states, nor the fact that actions affect limited portions of the state description (but we will address this later). First, let us consider applying standard search techniques.

6.4.1 Forward search

In forward search we start with the initial state and apply actions in a *forward* direction. That is, we perform *state progression*: starting with a complete state description that matches the action's precondition, determine the resulting state that satisfies the strongest postcondition. Since the previous state is complete, the successor state is also guaranteed to be complete (and unique).

For example, if we use the initial state given previously, and apply the action "put A on table" (also illustrated above), we obtain

AT	AB	AC	BT	BA	BC	CT	CA	CB	
0	1	0	1	0	0	1	0	0	pre-state
$*$	$*$	$*$	$*$	0	$*$	$*$	0	$*$	pre-condition
1	0	0	$*$	$*$	$*$	$*$	$*$	$*$	post-condition
1	0	0	1	0	0	1	0	0	post-state

6.4.2 Backward search

However, one need not only search forward in these problems. One can also search backwards! To conduct a backward search, one starts with the final goal and applies actions in the reverse direction. The search concludes successfully if a sub-goal is reached that matches the initial state.

The principle behind implementing a backward search is to determine the sub-goal that, if satisfied, allows one to reach the target goal in one step by applying the chosen action. That is, we perform *goal regression*: starting with a target goal that matches the action's postcondition, determine the previous sub-goal that satisfies the weakest precondition. That is, determine the most general sub-goal such that, if the action is applied in the forward direction from any state that satisfies the sub-goal, the resulting post-state is guaranteed to satisfy the target goal.

For example, if we apply the action "put A on B" in the reverse direction to the goal $\gamma = \text{AonB} \wedge \text{BonC}$ we obtain

AT	AB	AC	BT	BA	BC	CT	CA	CB	
$*$	$*$	$*$	$*$	0	1	$*$	0	0	sub-goal
$*$	$*$	$*$	$*$	0	$*$	$*$	0	0	pre-condition
0	1	0	$*$	$*$	$*$	$*$	$*$	$*$	post-condition
$*$	1	$*$	$*$	$*$	1	$*$	$*$	$*$	target-goal

That is, we obtain the new sub-goal

A	A	A	B	B	B	C	C	C
T	B	C	T	A	C	T	A	B
∗	∗	∗	∗	0	1	∗	0	0

So from any state that satisfies this sub-goal, if we apply the chosen action "put A on B", the resulting state is guaranteed to satisfy the original target goal $\gamma = \text{AonB} \wedge \text{BonC}$.

In order to determine which approach, forward or backward search, is more promising, it is useful to analyze the branching factor. If the closest solution is at distance d from the initial state, and the branching factor is b, a typical search algorithm will spend $O(b^d)$ time to find the solution.

E.g. For blocks world, if there are k stacks in a configuration, then there are k^2 possible forward actions (we can remove the top block from any of the stacks and put it on one of the remaining $k-1$ stacks or on the table). For backward search, if we assume that a goal contains k positive atomic propositions and no negative propositions (vector of 1s and '*'s), then there are k actions that can be applied backward. For example, for goal AonB $\wedge$ ConD we can apply either 'put A on B' or 'put C on D' in backward direction. Hence, the branching factor is only k. This is not a strict analysis, but it does give us some intuition that the branching factor for backward search might be smaller than the branching factor for forward search, so backward search may be more efficient.

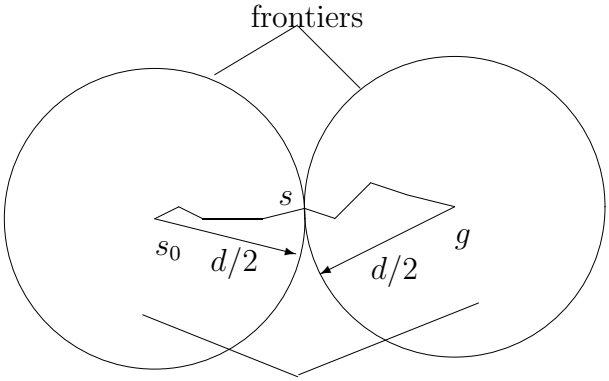
On the other hand, there are more goal vectors than state vectors (3^n versus 2^n) which suggests that there might be a trade-off—since intuitively, a larger search space could result in a slower search.

6.4.3 Bidirectional search

Forward and backward search can be done in parallel, in a method in a method called *bidirectional search.* In bidirectional search, two frontiers are expanded—one from the initial state, and the other from the target goal—until an intersection of the two frontiers is found. The final solution is constructed as the path $s_0 \rightarrow\rightarrow g$, where s_0 is the initial state, and g is the target goal.

If we assume that the branching factor for forward search and for backward search is b, and the shortest distance to a solution is d, then both forward and backward search need $O(b^d)$ time to find the solution. However,

the following figure illustrates that time needed for bidirectional search is $O(b^{d/2} + b^{d/2}) = O(b^{d/2})$ time.



Hence, when the forward and backward branching factors are equal, bidirectional search may be a good approach.

A problem: In bidirectional search, at least one search frontier has to be in memory, so it consumes a lot of space.

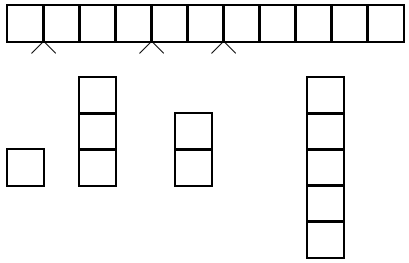
Readings

Russell and Norvig: Chapter 11
Dean, Allen, Aloimonos: Chapter 7

Appendix: Number of states in blocks world

Assume there are N blocks. Let us first ignore block labels.

In order to count number of configurations having k stacks ($1 \leq k \leq N$), we have to choose $k-1$ out of $N-1$ places to "cut" a long stack of blocks into k stacks (as in the following picture):



We see that we can make $\binom{10}{3}$ choices to split 11 blocks into 4 stacks ($N = 11$, $k = 4$).

Stack permutations are not important, so the number of distinct unlabeled configurations of k stacks is $\frac{1}{k!}\binom{N-1}{k-1}$.

Now, we can add labels to blocks: there are $N!$ label permutations; so the size of the state space for the blocks world is:

$$N!\left[\binom{N-1}{0} + \frac{1}{2}\binom{N-1}{1} + \frac{1}{3!}\binom{N-1}{2} + \ldots + \frac{1}{N!}\binom{N-1}{N-1}\right]$$

One can also note that the upper bound on the shortest solution is $2N-2$: We can always flatten the configuration to stacks of height 1 (at most $N-1$ moves), and build any configuration we want in not more than additional $N-1$ moves. Hence, the size of the search tree is not larger than $(N^2)^{2N-2} = N^{4N-4}$.