

Chapter 11

Search Strategies

In AI, we often use *state space* to represent the process of solving problems.

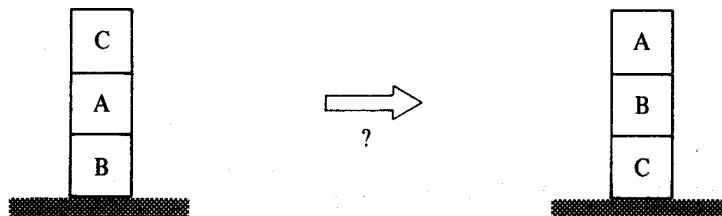
More specifically, a state space shows various situations a problem could be, and the ways to move in between. Then, solving a problem is reduced to find a path in the state space from the given problem to its solution.

Once such a representation is made, what's left is 1) to find such a path 2) as quickly as possible.

Various search strategies are used to achieve the second objective. We will discuss three such strategies in this part: depth-first search, breadth-first search, and iterative deepening; and a more sophisticated approach in the next chapter.

A classic example

The following picture gives the start configuration of the *block world*.

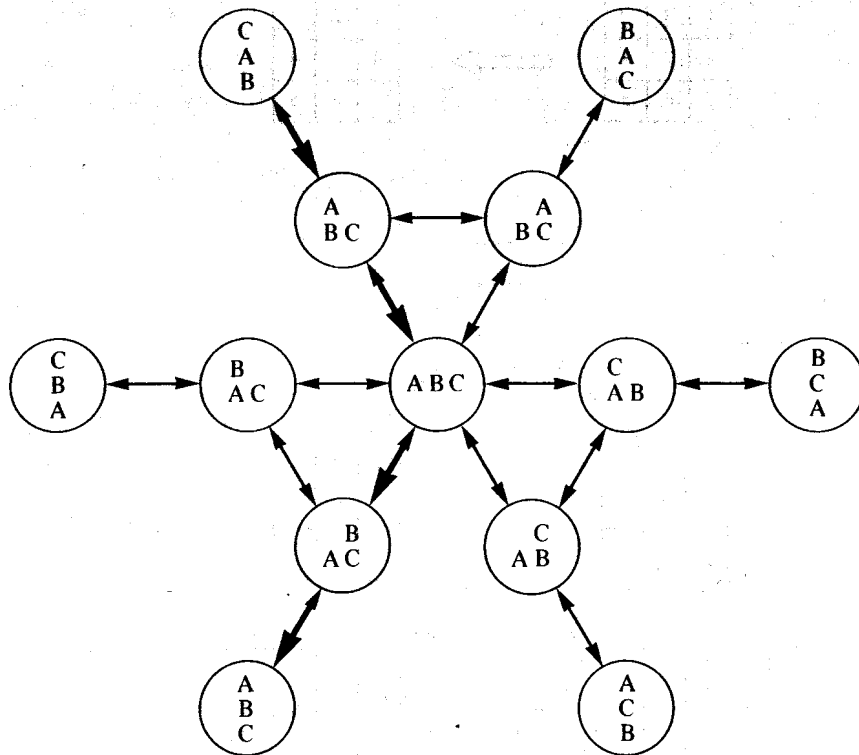


We can move only one block at a time. At the very beginning, we can only move block C onto the table. After that, we have three alternative actions to take: either put A on table; or put A on C; or put C back to A.

For such a problem, we need to keep a record of the current situation (*state*) and a record about the transformations between any pair of states when we take certain legal actions. The combination of such records constitutes a state space for a problem.

An example

Below gives a state space for the block world problem, starting with three stacks, consisting of A, B and C, respectively.



What to do?

In such a graph, the nodes refer to problem situations, and the bidirectional arcs represent the transitions (moves) between states.

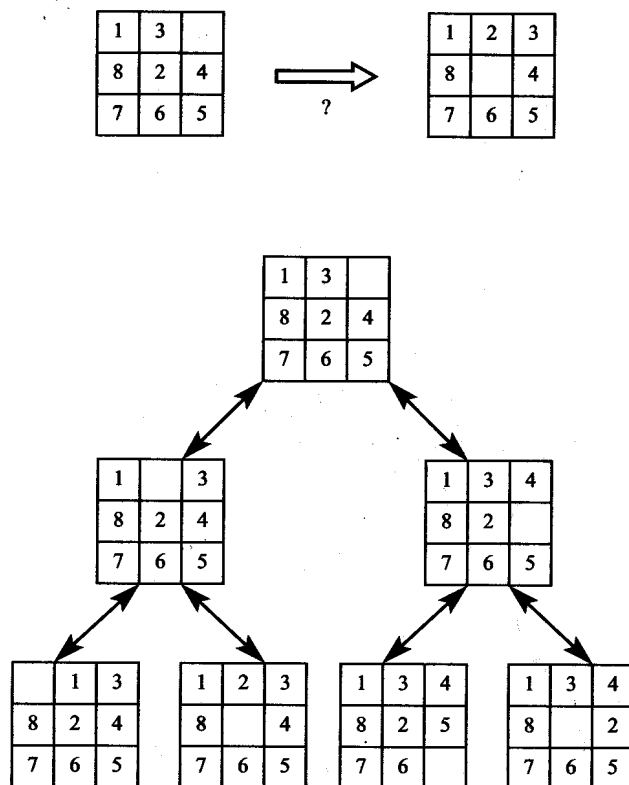
The problem of finding a solution, i.e., a collection of moves, is equivalent to finding a path from the given initial state, e.g., (ABC) to some specified state, called a *goal* state, e.g.,

$\begin{matrix} A \\ B \\ C \end{matrix}$.

Another example

Below shows a partial state space for an eight puzzle problem, where one of the cells is always empty, and any adjacent tile can move into it.

We want to start with some configuration, and eventually move into a given goal configuration.



State space in general

This idea of state space can be used to describe the solution process of other problems as well, such as the Tower of Hanoi, and the traveling salesman problem.

The common theme of using state space model to solve a problem is to define the problem as 1) a state space, 2) a start node, and 3) a goal condition that specify a set of nodes.

Incidentally, we can also attach costs to the moves. In the block world case, we can use cost to indicate some blocks are harder to move than the others. Then, we would be interested in finding a *minimum cost solution*.

Knowledge representation

The state space description can be represented with $s(X,Y)$, which is true iff there is a legal move from X to Y . If there are costs associated with moves, we can use $s(X,Y,C)$.

Although this relation can be explicitly specified with a bunch of facts, in reality, we would have to implicitly state them by providing some rules to find out the successors.

To represent a node, we want to make it compact, but easy to manipulate to generate succeeding nodes, calculate costs, etc..

For example, in the block world case, we can use a list of stacks. Thus, the initial state is $[[c,a,b], [], []]$, and the goal node could be either $[a,b,c], [], []]$, $[], [a,b,c], []]$, or $[], [], [a,b,c]$.

Solving block world

A move in the game is to move one block from the top of one stack, and put it at the top of another. This can be coded in Prolog as follows:

```
s(Stacks, [Stack1, [Top|Stack2] | Others]) :-  
  del([Top|Stack1], Stacks, Stacks1),  
  del(Stack2, Stacks1, Others).
```

It says, to change `Stacks`, consisting of `[Top|Stack1]`, `Stack2`, etc., to `[Stack1, [Top|Stack2] | Others]`, we go through three steps: 1) remove `[Top|Stack1]` to get `Stacks1`;

2) remove `Stack2` from `Stacks1` to get `Others`;
and

3) Add `Stack1`, and `[Top|Stack2]` back to `Others`.

Wrap it up

A goal node can be specified as follows:

```
goal(Situation):-  
    member([a,b,c],Situation).
```

We will program a search algorithm as a relation `solve(Start,Solution)`, where `Solution` is a path(a collection of moves) between the start state and one of the goal states. In this case, we will call

```
?-solve([[c,a,b],[],[ ]],Solution).
```

As you can imagine, it makes a big difference as how to program this `solve` relation.

Depth-first search

Given a state space specification of a problem, there could be a few different ways to search for a solution. Since state space is basically a graph, we can use the basic graph search algorithms to look for the path, i.e., DFS, and BFS. We start with DFS.

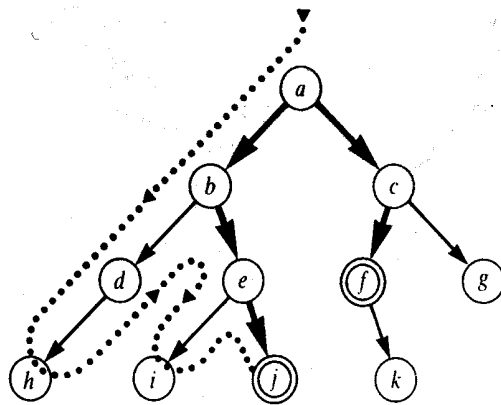
The basic idea of applying depth-first search to look for a solution path, S_{o1} , from a given node N to some goal node is 1) if N itself is a goal node, we are done. 2) Otherwise, if there is a successor node, $N1$, of N such that there is a path S_{o11} from $N1$ to a goal node, then $S_{o1}=[N|S_{o11}]$. Thus,

```
solve(N, [N]) :- goal(N).  
solve(N, [N|So11]) :- s(N, N1), solve(N1, So11).
```

Why DFS?

We discuss DFS first since it is very appropriate for the recursive style of programming, and also it is the way Prolog does its business when it looks for alternative solutions when backtracking.

Below gives an example of DFS.



The order in which the DFS visits nodes is a, b, d, h, e, i, j. One of the solution is a, b, d, h, e, i, j. Another solution is also found during backtracking: a, c, f.

Eight queens in DFS

A state-space formulation of the eight queens problem can be the following:

1. Nodes are board positions with any number of queens placed in consecutive columns of the board.
2. A successor node is obtained by placing another queen into the next column so that she does not attack any of the existing queens.
3. The start node is the empty chessboard.
4. A goal node is any configuration with eight queens.

The essential code

If we add the following problem specific code into the general DFS solver,

```
s(Queens, [Queen|Queens]) :-  
    member(Queen, [1,2,3,4,5,6,7,8]),  
    noattack(Queen, Queens).
```

```
goal([_,_,_,_,_,_,_,_,_]).
```

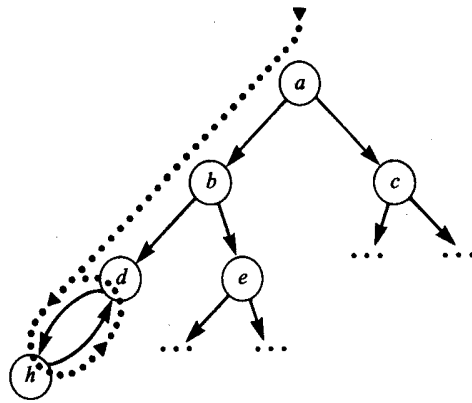
and then call

```
?-solve([], Solution).
```

A problem

Sometimes, the DFS strategy can run into troubles. Particularly, for some state space, it may get into infinite loop.

Below shows such a case, where the DFS gets stuck between d and h : a, b, d, h, d, h, \dots



This is also the issue we could run into with the block world problem, if we want to go from $[bac]$ to $[abc]$.

A simple minded solution

```
%accessory function
del(X, [X|L], L).
del(X, [Y|L], [Y|L1]):- del(X, L, L1).

%legal moves
s(Stacks, [Stack1, [Top1|Stack2] |OtherStacks]):-
  del([Top1|Stack1], Stacks, Stacks1),
  del(Stack2, Stacks1, OtherStacks).

%goal condition
goal(Situation):-
  member([a, b, c], Situation).

%general solution
solve(N, [N]):-goal(N).
solve(N, [N|Sol1]):-s(N, N1), solve(N1, Sol1).
```

This one leads to spave overflow.

A solution

An obvious solution is not to consider any node that has already been visited.

Let `depthfirst(Path,Node,Solution)` be a relation, where `Node` is the current node, `Path` collects all the nodes visited so far from the start node to the predecessor of `Node`, and `Solution` extends `Path` to a goal node, via `Node`.

```
solve(Node,Solution):-
    depthfirst([],Node,Solution).

depthfirst(Path,Node,[Node|Path]):-
    goal(Node).

depthfirst(Path,Node,Sol):-
    s(Node,Node1),
    not(member(Node1,Path)),
    depthfirst([Node|Path],Node1,Sol).
```

An example

If we apply this modified program to solve the block world problem, with the given start node being `[[b, a, c], [], []]`, i.e.,

```
?- solve([[b, a, c], [], []], S).
```

and the goal being `[[a, b, c], [], []]`, we will find out that it takes 20 moves to get to the end, with the last few moves being the following, in a reversed order.

```
[[], [a, b, c], []],  
[[a], [b, c], []],  
[[b, a], [c], []],  
[[], [c, b, a], []],  
[[c], [b, a], []],  
...  
[[b, a, c], [], []]
```

Another problem and its solution

This improved strategy will not fall into infinite loop through cycles, but it may still get into an infinite path in the state space.

One way to overcome this issue is to add another argument, `Maxdepth`, which specifies how deep we are willing to dig.

```
depthfirst2(Node, [Node], _) :- goal(Node).
```

```
depthfirst2(Node, [Node|Sol], Maxdepth) :-  
    Maxdepth > 0, s(Node, Node1),  
    Max1 is Maxdepth - 1,  
    depthfirst2(Node1, Sol, Max1).
```

Since, we could not predict beforehand what should be the proper value of `Maxdepth`, we will lose completeness of the search strategy.

Better solutions

An improvement is to start from a low value for the depth, then gradually increase it until a solution is found. This is called *iterative deepening*.

Let `path(N1,N2,Path)` be such that `Path` gives, in the reverse order, a path from `N1` to `N2`. It can be defined as follows:

```
path(N,N,[N]).
path(First,Last,[Last|Path]):-
    path(First,OneButLast,Path),
    s(OneButLast,Last),
    not(member(Last,Path)).
```

This relation will find out, for a given start node, all the acyclic paths of increasing length. This is exactly what we want in implementing the iterative deepening idea.

Final words

Therefore, the iterative deepening idea can be coded in Prolog as follows:

```
depth_first_ID(Node,Solution):-  
    path(Node,GoalNode,Solution),  
    goal(GoalNode).
```

If we add in the block world specific rules, we will get the following solution in just 5 steps.

```
[[[] , [a, b, c], []],  
 [ [] , [b, c], [a]],  
 [[c], [a], [b]],  
 [[a, c], [b], []],  
 [[b, a, c], [], []]]
```

The whole thing

```
%goal specification
goal(Situation):-member([a, b, c], Situation).

%to delete a member from a list.
del1(X, [X|Tail], Tail).
del1(X, [Y|Tail], [Y|Tail1]):-del1(X, Tail, Tail1).

%transition rule
s(Stacks, [Stack1, [Top|Stack2] |Others]):-
    del1([Top|Stack1], Stacks, Stacks1),
    del1(Stack2, Stacks1, Others).

%This is for DFS Iterative depth
depth_first_ID(Node, Solution):-
    path(Node, GoalNode, Solution), goal(GoalNode).

%This is the program as shown in pp. 248.
path(N, N, [N]).
path(First, Last, [Last|Path]):- p
path(First, OneButLast, Path),
    s(OneButLast, Last), not(member(Last, Path)).

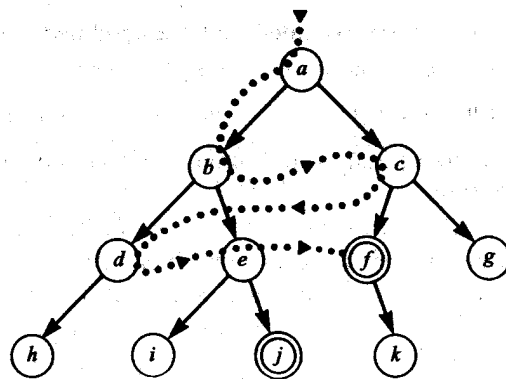
%a sample query
%?-depth_first_ID([[b, a, c], [], []], Solution).
```

Breadth-first search

BFS is just the opposite of DFS: It is slow, but guarantees a solution if there is one. More specifically, it will check out all the closer nodes before checking those further away. It is really backed up by a queue, not a stack. Thus, it is a bit more difficult to code.

Technically speaking, we have to maintain a set of alternative nodes from which a path can be further extended.

The relation `breadthfirst(Path, Solution)` means some path from a candidate in `Path` can be extended to a goal node. `Solution` keeps that extended path. Below shows the process.



We will use a list to collect these candidate paths, each of which is represented as a list. Initially, this set contains just `[[StartNode]]`. In general, if the head of the first path is a goal node, we are done. Otherwise, remove the head from the first path, and generate the set of all the possible one-step extension of this first path, add this extension set to the *end* of the candidate set, then apply BFS recursively to the updated candidate set. Thus,

```
solve(Start,Solution):-
    breadthfirst([[Start]],Solution).

breadthfirst([[Node|Path]|_],[Node|Path]):-
    goal(Node).

breadthfirst([Path|Paths],Solution):-
    extend(Path,NewPaths),
    conc(Paths,NewPaths,Paths1),
    breadthfirst(Paths1,Solution).
```

```

extend([Node|Path],NewPaths):-
  bagof([NewNode,Node|Path],
    (s(Node,NewNode),
     not(member(NewNode,[Node|Path]))),
    NewPaths),!.
extend(Path,[]).

```

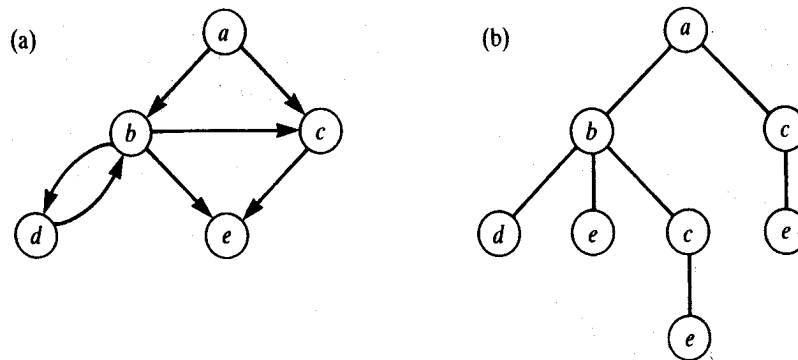
As an example, if we start with `[[a]]`, in the next step we will generate `[[b,a],[c,a]]`. Then we remove the first path, i.e., `[b,a]`, and extend it into `[[d,b,a],[e,b,a]]`, and then put it back and we have `[[c,a],[d,b,a],[e,b,a]]`. We then remove `[c,a]`, extend it, put it back to the candidate path to have a new candidate path set: `[,[d,b,a],[e,b,a],[f,c,a],[g,c,a]]`.

One more step later, `[f,c,a]` becomes the first path, and the search is complete, since `f` is a goal node.

Homework: Exercises 11.6, and 11-11.

Some analysis

Recall that both BFS and DFS not only work for trees, but work for general graphs as well, although it may duplicate part of the graph as shown in the following graphs.



By the very definition, BFS generates all the solution paths according to their lengths. This is important if we want to have optimal solutions in terms of path length. Although the general DFS does not do this, the iterative deepening does. On the other hand, these algorithms do not consider the overall cost. We will address this issue in the next chapter.

Time complexity

If we apply BFS to a tree with *branching factor*, i.e. the maximum number of children in that tree, being b , and the length of the shortest solution path being d , then the total number of nodes the BFS has to explore is at most $1 + b + b^2 + \dots + b^{d-1} = O(b^d)$. Thus, the time complexity of BFS is $O(b^d)$.

A general DFS may miss completely the shortest solution and get into an infinite loop. For the version coming with a maximum depth d_{max} such that $d \leq d_{max}$, we would have its time complexity being $O(b^{d_{max}})$, when it has to explore all these nodes. But, its space complexity is only $O(d_{max})$ since it only keeps the path being explored.

Iterative DFS

For the iterative deepening version with the maximum depth being d , the algorithm looks for all at depth 0, then all at depth no more than 1, ..., finally all at depth no more than d .

Thus, it visits the start node $d + 1$ times, its b children d time, those at depth 2, b^2 of them, $d - 1$ times, ..., and those at depth d precisely one time.

Hence, in the worst case, the number of nodes it visits could be

$$\begin{aligned} T(d) &= (d + 1) * 1 + d * b + \dots + 1 * b^d \\ &= O(b^d). \end{aligned}$$

Space wise, it takes only d .

The best of both worlds

It can be shown that the ratio of nodes generated between the iterative deepening version of DFS and BFS is about $b/(b - 1)$, which is small, when $b \gg 2$.

Thus, the iterative deepening version of DFS seems to combine the best of both DFS in terms of its efficiency and BFS in terms of its guaranteed solution.

Below shows the comparison of these methods.

	Time	Space	Completeness
BFS	b^d	b^d	yes
DFS	$b^{d_{max}}$	d_{max}	no
DFS(ID)	b^d	d	yes

Question: Have you checked the project page recently?