State Space Search

Chapter 2

Problem solving basically involves doing the right thing at the right time. Given a problem to solve the task is to select the right moves that would lead to the solution.

Consider the situation in a football game you are playing. You are shown as player *P* in Figure 2.1 below. You have the ball with you and are running towards the opponent's goal. There are some other players in the vicinity also shown in the figure along with the directions of their movement. You have a problem to solve. What should you do next? What move should you make? What is the "intelligent" thing to do?

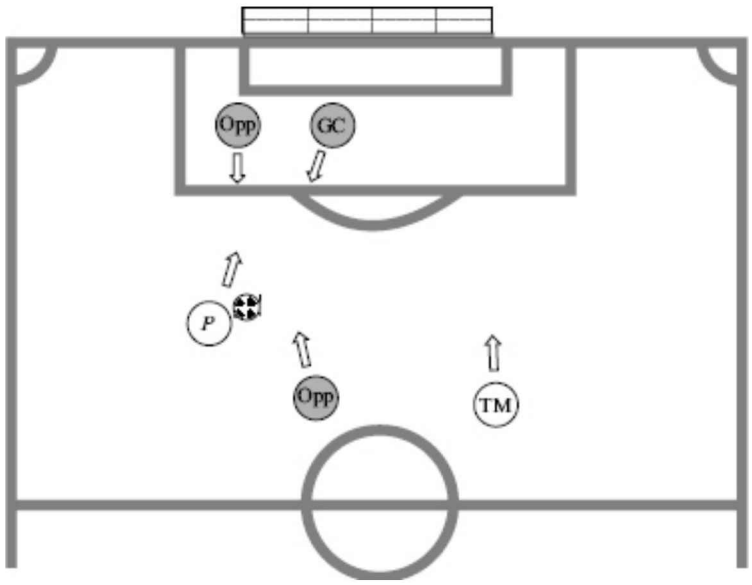


FIGURE 2.1 A problem to solve on the football field. What should player *P* do next? GC is the opponent goalkeeper, Opp are opponents, and TM is teammate. Arrows show the direction of each player's movement.

To solve such a problem on a computer, one must first create a representation of the domain; in this case, a football game being the problem, the decision to make next, the set of alternatives available, and the moves one can make. The above example is a problem in a multi-agent scenario. There are many players and the outcome depends on

their moves too. We will begin, in the initial chapters, with a much simpler scenario, in which there is only one player who can make changes in the situation. We will look at algorithms to analyse the complete problem and find a solution. In later chapters, we will explore ways to deal with multi-agent scenarios as well.

We will also use the term *agent* to refer to the problem solver. The problem solver, or the agent, operates on a *representation* of the *space* in which the problem has to be solved and also a representation of the *moves* or decisions that the agent has to choose from to solve the problem.

Our endeavour will be to write the algorithms in a domain-independent manner. Then the algorithms will be general purpose in nature and could be adapted to solve problems in different domains as depicted in Figure 2.2. A user would only have to implement the domain description and call the domain specific functions from the general program.

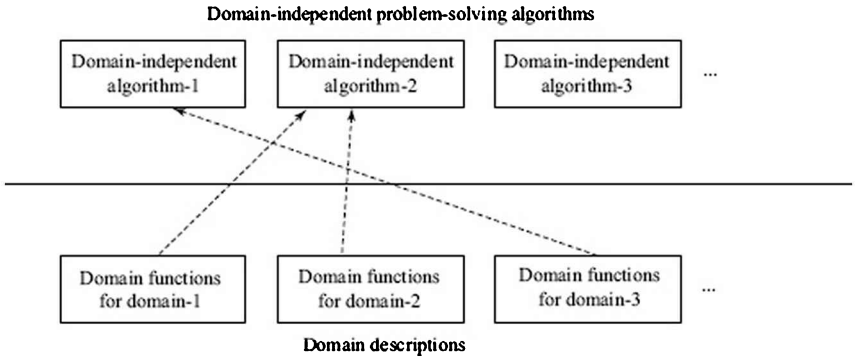


FIGURE 2.2 The goal is to develop general problem solving algorithms in a domain-independent form. When a problem is to be solved in a domain then the user needs to create a domain description and plug it in a suitable problem solving algorithm.

We model the problem solving process as traversing a *state space.* The state space is a space in which each element is a state. A state is a description of the world in which the problem solver operates. The given situation is described by a state called the *START* state. The desired or the goal situation is described by one or more *GOAL* states. In any given state, an action or a decision by the agent changes something and the agent makes a *move* to a new state. The task is to make a sequence of moves, such that the agent ends up being in a goal state.

The set of choices available to us *implicitly define* the space in which the decision making process operates. We do not assume that the entire state space is represented in some data structures. Instead, only the states actually *generated* by the agent exist explicitly. The unseen states are only implicit. In this implicit space, we visualize the problem as follows. Initially, we are in some given state, or the *START* state. We desire to be in some state that we will call the *GOAL* state, as shown in Figure 2.3. The desired state may be described completely, identifying

the state, or it could be described partially by some desirable properties; in which case there may be more than one goal state satisfying the properties. The given state being the current state is described completely.

This transformation from the start state to the goal state is to be made by a sequence of moves that are available to us in the domain of problem solving. In the beginning, the search algorithm (or agent) can only "see" the start state. It has access to some move generation operators that determine which states are reachable in one step (see Figure 2.6). The algorithm has to choose one of these moves. Each move applied to a given state transforms it into another state. Our task is to find those sequences of moves that will transform our current *(START)* state into the desired *(GOAL)* state. The solution is depicted in Figure 2.4.

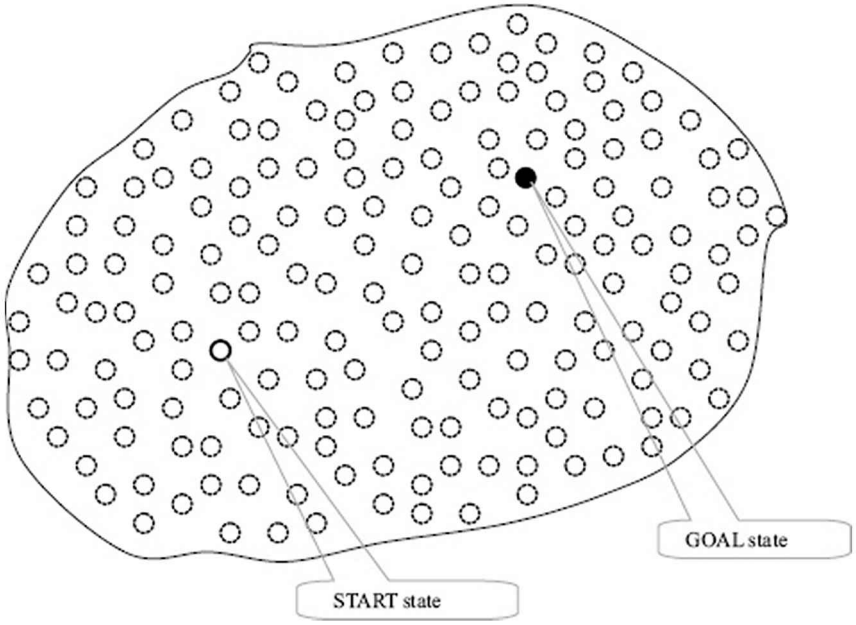


FIGURE 2.3 The state space.

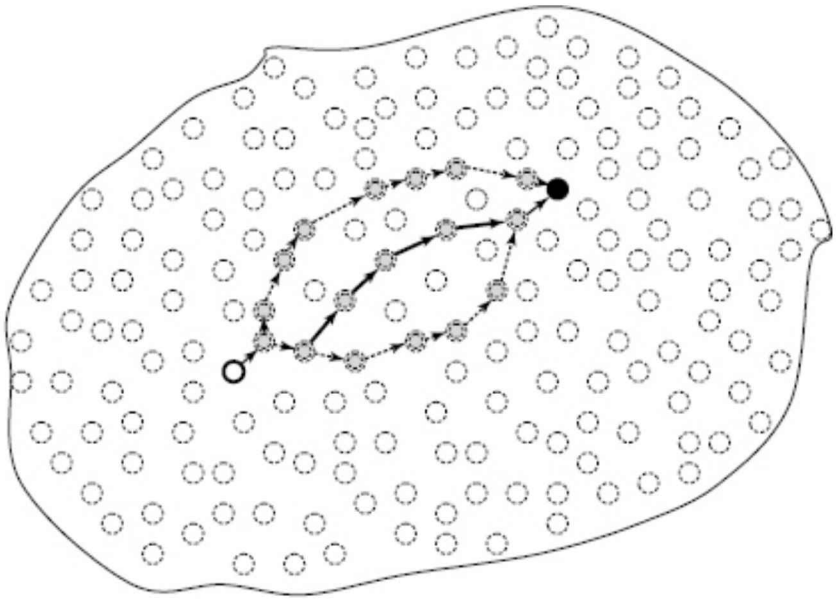


FIGURE 2.4 The solution is a sequence of moves that end in the GOAL state. There may be more than one solution to a problem. The figure shows one solution with thick arrows, and two alternatives with dotted arrows.

2.1 Generate and Test

Our basic approach will be to search the state space looking for the solution. The high level search algorithm has two components; one, to generate a candidate from the state space, and two, to test whether the candidate generated is the solution. The high level algorithm is given below in Figure 2.5.

```
Generate And Test()
   1 while  more candidates exist
   2    do   Generate a candidate
   3       Test   whether it is a solution
   4 return FAILURE
```

FIGURE 2.5 A high level search algorithm.

The rest of the chapter will be devoted to refining the above algorithm. We assume that the problem domain has functions defined that allow us to operate in the domain. At the moment, we need two functions to be defined on the domain. They are the following:

moveGen(State) Takes a state as input and returns a set of states that are reachable in one step from the input state, as shown in Figure 2.6.

We call the set of states as *successors* or *children* of the input state. The input state is the parent of the children.

goalTest(State) Returns *true* if the input state is the goal state and *false* otherwise.

goalTest(State, Goal) Returns *true* if *State* matches *Goal,* and *false* otherwise.

Observe that we may have either of the above goal-test functions. The former is used when the goal is described by some properties that are checked by the function. The latter takes an explicit goal state and uses that to compare with the candidate state.

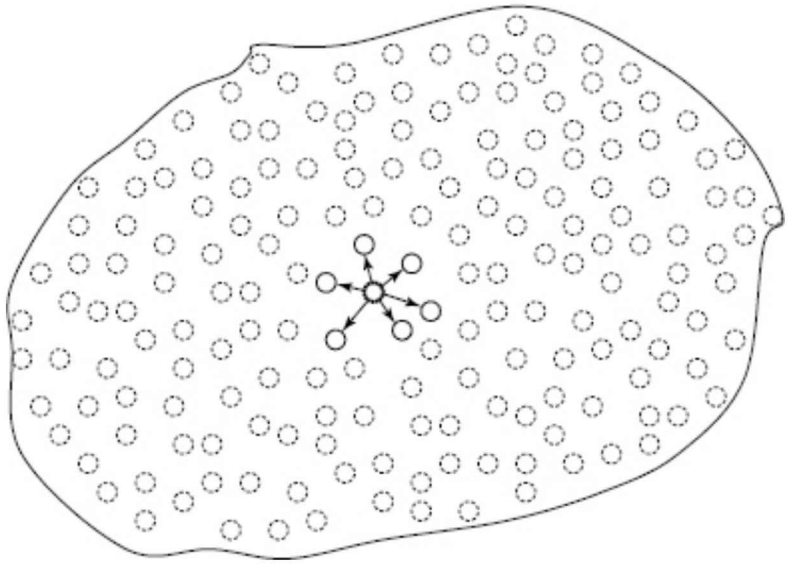


FIGURE 2.6 The moveGen function returns a set of states that are reachable in one move from a given state.

Our search algorithms should be able to operate in any domain for which the above functions are provided. We view the set of states as 'nodes' in the state space, and the set of moves as the edges connecting these nodes. Thus, our view of the state space will be that of a graph that is defined *implicitly* by the domain function moveGen. Figure 2.7 depicts the part of the state space generated and explored by the *Generate&Test* algorithm. The generated space is known as the *search tree* generated by the search program.

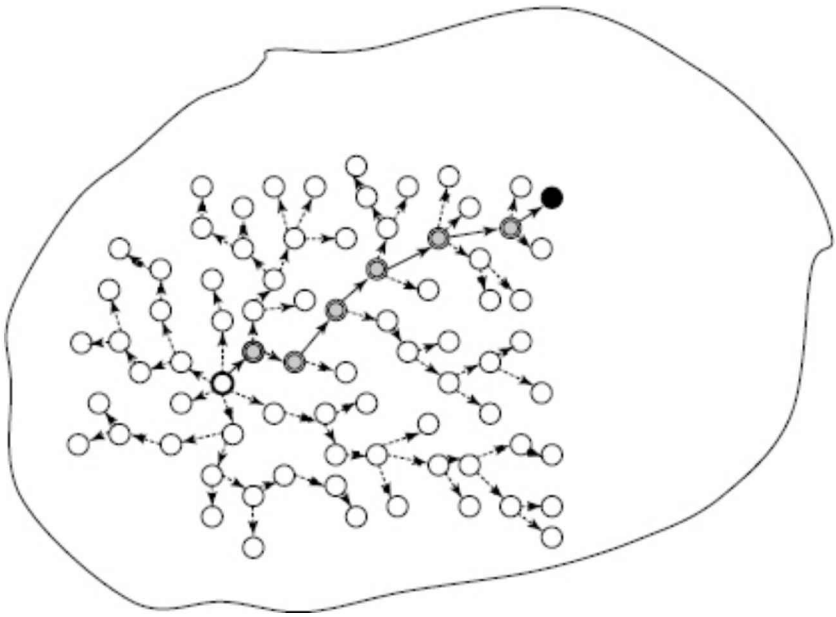


FIGURE 2.7 The nodes visited by a search algorithm, form a search tree shown by empty circles and dotted arrows. The solution found is shown with shaded nodes and solid arrows.

Before looking at the algorithms in detail, let us look at a couple of problem examples . While many real world problems can indeed be posed as search problems, we prefer simpler problems often posed as puzzles. Small, well defined problems are easy to visualize and easy to implement, and serve to illustrate the algorithms. Real problems on the other hand would be complex to represent, and divert from the understanding of search methods. Nevertheless, the reader is encouraged to pose real world problems as state space search problems. A few suggestions are given in the exercises.

Common sorts of puzzles are river-crossing puzzles. In these puzzles, a group of entities need to cross the river, but all of them cannot cross at the same time due to the limited capacity of the boat or the bridge. In addition, there may be constraints that restrict the combination of entities that can stay in one place.

One example of a river-crossing puzzle is the missionaries and cannibals problem stated as follows. There are three missionaries and three cannibals who want to cross the river. There is a boat with a capacity to seat two. The constraint is that if on any bank the cannibals outnumber the missionaries, they will eat them[1]. How do all the six people get across safely? In the Text Box 2.1, we look at another river-crossing problem, and also design the domain functions required.

Box 2.1: The Man, Lion, Goat, Cabbage Problem

A man (M) needs to transport a lion (L), a goat (G), and a cabbage (C) across a river. He has a boat (B) in which he can take only one of them at a time. It is only his presence that prevents the lion from eating the goat, and the goat from eating the cabbage. He can neither leave the goat alone with the lion, nor the cabbage with the goat. How can he take them all across the river?

The state representation and move generation functions are interrelated. Let us say that we represent the state as a list of two lists, one for each bank of the river, say the left bank L and the right bank R. In each list, we name the entities on that bank.

```
start state S = ((M G L C B) ())
goal state G = (() (M G L C B))
```

The move generation could first generate all possible moves and then filter out illegal moves. For example, from the start state, the following states are reachable:

```
((G L C) (M B)), ((L C) (M G B)), ((G C) (M L B)), and ((G L) (M C B))
```

Of these, only the second state ((L C) (M G B)) is a legal state. Notice that the moveGen has transferred some elements from the first list to the second. In the next move it will need to transfer elements in the opposite direction. Also, there is redundant information in the representation. When one knows what is on the left bank, one also knows what is on the right bank. Therefore, one of the lists could be removed. To solve the problem of alternating directions, we could choose a representation that lists the entities on the bank where the boat is, and also which bank it is on, as shown below.

```
start state S = (M L G C Left)
goal state G = (M L G C Right)
```

where Left denotes that the boat is on the left bank and Right on the right bank. Note that M is redundant, because the man is where the boat is, and could have been removed.

The *moveGen*(*N*) function could be as follows.

```
Initialize set of successors C to empty set.
Add M to the complement of given state N to get new state S.
If given state has Left, then add Right to S, else add Left.
If legal(S) then add S to set of successors C.
For each other-entity E in N
     make a copy S' of S,
     add E to S',
     If legal (S'), then add S' to C.
Return (C).
```

The complement of a state is with respect to the set {M L G C}.

The function “legal (state:S)” will return “no” or “false” if either both G and C or both L and G are missing from S, otherwise it will return “yes” or “true”.

The following figure shows the well known *Eight Puzzle* used extensively in one of the earliest textbooks on artificial intelligence (Nilsson, 1971). The goal is to find a path to a desired arrangement of tiles, for example as shown in Figure 2.8.

The idea in *weak methods* or general methods in artificial intelligence (AI) is that we develop a general algorithm that can be applied to many domains. In the above examples, we have illustrated how problems can be posed as search problems. Let us return to the design of the search algorithms. The following refinement of *Generate&Test* falls in the category of forward state space search. Here we begin with the *START* state and search until we reach the *GOAL* state.

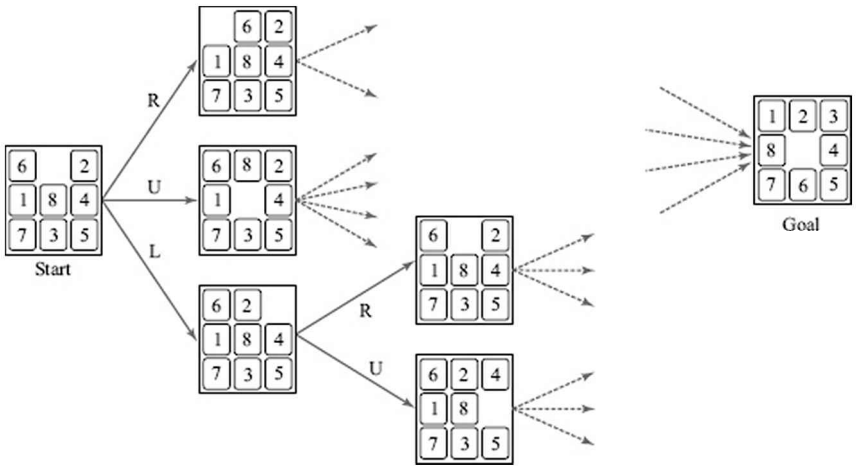


FIGURE 2.8 The *Eight Puzzle* consists of eight tiles on a 3 × 3 grid. A tile can slide into an adjacent location if it is empty. A move is labelled R if a tile moves right, and likewise for up (U), down (D) and left (L).

2.2 Simple Search 1

We assume a bag data structure called *OPEN* to store the candidates that we have generated. The algorithm is given below.

```
SimpleSearch1()
  1 open ← {start}
  2 while open is not empty
  3    do pick some node n from open
  4       open ← open \ {n}
  5       if GoalTest(n) = TRUE
  6       then return n
  7       else open ← open ∪ MoveGen(n)
  8 return FAILURE
```

FIGURE 2.9 Algorithm *SimpleSearch1*.

Box 2.2: Hercules, Hydra and CombEx

In Greek mythology, one of the monsters that Hercules found himself fighting was Hydra. Hydra was not an easy monster to deal with. For every head that Hercules chopped off, many more grew in its place. The problem faced by our search program is not unlike the one faced by Hercules. For every node that Simple-Search-1 picks to inspect, many more are added in the bag.

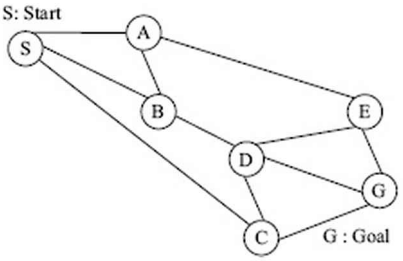
In mathematical terms, the search tree generated by the program grows exponentially. And yet, as Hercules demonstrated in his fight with Hydra, the monster can eventually be overcome.

Almost all the research we see in artificial intelligence can be seen as a battle against CombEx, the monster that makes our problems grow with a 'COMBinatorial Explosion'.

Our battle with CombEx will require two abilities. The first will be the basic skills to navigate and explore the search space. This is akin to learning how to handle the sword. But as we will see, this will not be enough. We will also need to know where to strike. Without knowledge to guide the exploration, search will be hopelessly outnumbered by the exploding possibilities.

The first is necessary because CombEx will appear in new forms as we strive to solve new problems. But as we wage the battle, we must be able to learn the strengths and weaknesses of CombEx in every domain. We must acquire knowledge both through our own experience and the accumulated experience of other problem solvers.

Let us look at a small, synthetic problem to illustrate our search algorithms. Figure 2.10 depicts the state space graph for a tiny search problem.



Simple Search 1 simply picks a node N from OPEN and checks if it is the goal.
- **If Yes, it returns the node N**
- **If No, it adds all children of N to OPEN**

FIGURE 2.10 A tiny search problem.

There are two problems with the above program. The first is that the program could go into an infinite loop. This would happen when the state space is a graph with loops in which it would be possible to come back to the same state. In the *Eight Puzzle* described above, the simplest loop would be when a tile is moved back and forth endlessly. For the tiny search problem of Figure 2.10, a possible trace of the search algorithm is shown below in Figure 2.11, in which the same nodes are visited again and again.

Mazes are a very good example where infinite loops lurk dangerously. They also illustrate the nature of difficulty in the search. See the accompanying

```
(S)
(ABC)
(SACD)
(ABCACD)
(SACACDACD) ...
```

FIGURE 2.11 Simple search may go into loops. One possible evolvement of OPEN is shown above. The node picked at each stage is underlined.

Box 2.3: The Problem of Visibility in Search

Anyone who has gone trekking in the mountains would have experienced the following. You are at a peak looking down towards a village in the valley. You can clearly chart a path through the woods on the way down, past those set of boulders, back into the woods, and then following a path along the sunflower fields. But when you start descending and enter the woods, the perspective and clarity you had from the top curiously vanishes.

A similar problem is faced by people navigating a network of roads. It is all very clear when you are poring over a map. But down at the next intersection, it becomes a problem which road to take.

The search algorithms we are trying to devise do not have the benefit of perspective and global viewpoints. They can only see a node in the search space, and the options available at that node. The situation is like when you are in a maze. You can see a number

of options at every junction, but you do not have a clue about the choice to make. This is illustrated in Figure 2.12.

In the next chapter, we will investigate methods to take advantage of clues obtained from the domain. If for example, one of the paths in the maze has a little more light than the others, it could be leading to an exit.

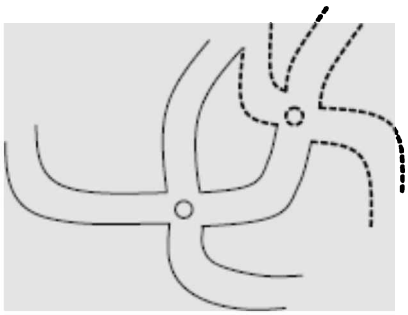


FIGURE 2.12 In a maze, one can only see the immediate options. Only when you choose one of them do the further options reveal themselves.

Box 2.3 for a note on the limited visibility in search. The key to not getting lost endlessly in a maze is to mark the places where one has been. We can follow a similar approach by maintaining a list of seen nodes. Such a list has traditionally been called *CLOSED*. It contains the list of states we have tested, and should not visit again. The algorithm *Simple Search 2* (SS-2) in Figure 2.13 incorporates this check. It does not add any seen nodes to *OPEN* again. To prune the search tree further, one could also specify that it does not add any successors that are already on *OPEN* as well. This would lead to a smaller search tree in which each node occurs exactly once.

```
SimpleSearch2()
  1 open ← {start}
  2 closed ← {}
  3 while    open is not empty
  4   do Pick some node n from open
  5      open ← open \ {n}
  6      closed ← closed ∪ {n}
  7      if GoalTest(n) = TRUE
  8        then return n
  9        else open ← open ∪ {MoveGen(n) \ closed}
 10 return FAILURE
```

FIGURE 2.13 Algorithm *SimpleSearch2*.

The search tree generated by one possible execution of the algorithm is shown below. The *CLOSED* list is depicted by shaded nodes and the

OPEN list by dotted circles.

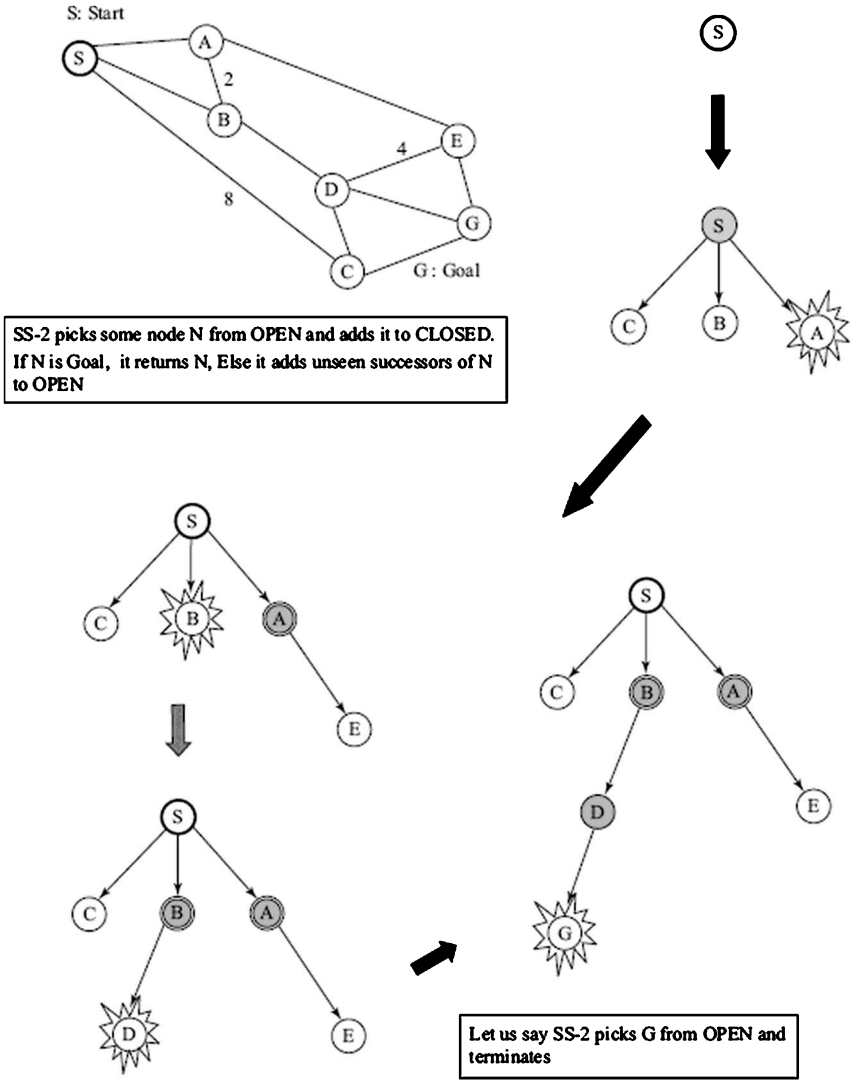


FIGURE 2.14 SS-2 visits each node only once and finds the goal.

The second problem with the above program is that it returns the goal state when it finds it. This is not always problematic though. There *are* problems in which we are only interested in finding a state satisfying some properties. For example, the *n-queens* problem in which we simply need to show a placement of n queens on an $n \times n$ chessboard such that no queen attacks (as defined by rules of the game) any other queen. We can call such problems as *configuration problems.* The other kinds of problems, that we call *planning problems,* are those in which we need a

sequence of moves or the path to the goal state. Clearly, our program is inadequate for the latter kind of problems.

One simple approach is to explicitly keep track of the complete path to each new state. We can do this by modifying the state representation (in the search algorithm only) to store a list of nodes denoting the path. In order to work with the same domain functions *moveGen* and *goalTest,* we need to modify our algorithm. Simple-Search-3 (SS-3) incorporates this change (see Figure 2.15). After picking the node from *OPEN*, it extracts the current state *H*. It tests *H* with *goalTest* and generates its successors using *moveGen*. When the goal node is found, all that the algorithm needs to do is to reverse the node to get the path in the proper order. The function *mapcar* in Figure 2.15 takes a list of lists as an argument and returns a list containing the heads of the input lists. The function *cons* adds a new element to the head of a list.

In some problems, the path information can be represented as part of the state description itself. An example of this is the knight's chessboard-tour problem given in the exercises. One could start by representing the state as a 10 × 10 array in which the centre 8 × 8 sub-array, initialized to say 0, would be the chessboard. The starting state would be a square on the chessboard labelled with 1, and subsequent squares could be labelled in the order in which the knight visits them.

```
SimpleSearch3()
  1  open ← {(start)}
  2  closed ← {}
  3    while open is not empty
  4      do Pick some node n from open
  5        h ← Head(n)
  6        if GoalTest(h) = TRUE
  7          then return Reverse(n)
  8          else closed ← closed ∪ {h}
  9               successors ← {MoveGen(h) \ closed}
  10              successors ← {successors \ Mapcar(open)}
  11              open ← {open \ {n}}
  12              for each s in successors
  13                do Add Cons(s,n) to open
  14 return FAILURE
```

FIGURE 2.15 Algorithm SS-3 stores the path information at every node in the search tree.

The path information stored in the node could also be exploited to check for looping. All that the algorithm would need to do is to check if the new candidate states are not in the path already. While the pruning of nodes will not be as tight as our algorithm above, it would require lesser storage since the set *CLOSED* will no longer be needed. We also remove those successors that are already on *OPEN*. Figure 2.16 shows the *OPEN* and *CLOSED* list for the tiny search problem for the algorithms *SS-2* and *SS-3.*

The search tree as seen by SS-2		SS-3 maintains entire path information at each node on OPEN in the search tree.	
		The search tree as seen by SS-3	
OPEN	CLOSED	OPEN	CLOSED
(S)	()	((S))	()
(ABC)	(S)	((AS) (BS) (CS))	(S)
(BCE)	(SA)	((BS) (CS) (EAS))	(AS)
(CDE)	(SAB)	((CS) (DBS) (EAS))	(BAS)
(CGE)	(DABS)	((CS) (GDBS) (EAS))	(DBAS)
SS-2 terminates when G is picked and found to be the goal.		Again, the search terminates when G is picked.	
The program returns G.		But now the program reverses the path and returns SBDG, which is the path found from S to G.	

FIGURE 2.16 The search trees as seen by SS-2 and SS-3.

In *SS-3, OPEN* contains paths and *CLOSED* contains states. Next, we modify our search function such that both *OPEN* and *CLOSED* store node pairs, representing a node and its parent node in the search tree. Now, all nodes in the search tree have the same structure. We will, however, need to do more work to return the path found. Each node visited has a back pointer to its parent node. The algorithm *reconstructPath* below reconstructs the path by tracing these back pointers until it reaches the start node which has *NIL* as back pointer.

```
ReconstructPath(nodePair, closed)
  1 path ← List(Head(nodePair))
  2 parent ← Second(nodePair)
  3 while parent is not NIL
  4   dopath ← Cons(parent, path)
  5        nodePair ← FindLink(parent, closed)
  6        parent ← Second(nodePair)
  7 return path

FindLink(child, closed)
  1 if child = Head(Head(closed))
  2   then return Head(closed)
  3   else return FindLink(child, Tail(closed))
```

FIGURE 2.17 Reconstructing the path from the list of back pointers of each node involves retrieving the parent of the current node all the way back to start whose parent, by definition, is *NIL*. *nodePair* is a pair that contains the goal *g* and its parent node. The functions *List, Head, Second, Tail* and *Cons* for the list data structure.

In the algorithms in Figures 2.17, 2.18 and 2.20, we also move from the *set* representation to a *list* representation. It calls a function *removeSeen* to prune the list of successors, and *makeNodes* to prepare the successors in the form for adding them to *OPEN*. We also make it deterministic by picking the new candidate from the head of the list *OPEN,* since this is an easy operation for most data structures.

2.3 Depth First Search (DFS)

The algorithm *DFS* given below (Figure 2.18) treats *OPEN* like a *stack*. It adds the new candidates at the head of the list. The reason it is called *depth first* is that from the search tree, it selects a node from the *OPEN* list that is *deepest* or farthest from the start node. The candidates are inspected in the last-in-first-out order. This way its focus is on the newer arrivals first, and consequently its characteristic behaviour is that it dives headlong into the search space. Figure 2.19 illustrates the search tree as seen by *DFS,* represented by the two lists *OPEN* and *CLOSED.*

```
DepthFirstSearch()
  1 open ← ((start NIL))
  2 closed ← ()
  3 while not Null(open)
  4   do nodePair ← Head(open)
  5      node ← Head(nodePair)
  6      if GoalTest(node) = TRUE
  7        then return ReconstructPath(nodePair, closed)
  8        else closed ← Cons(nodePair, closed)
  9             children ← MoveGen(node)
 10             noLoops ← RemoveSeen(children, open, closed)
 11             new ← MakePairs(noLoops, node)
 12             open ← Append(new, Tail(open))
 13 return "No solution found"

RemoveSeen(nodeList, openList, closedList)
  1 if Null(nodeList)
  2   then return ()
  3     else n ← Head(nodeList)
  4       if (OccursIn(n, openList) OR OccursIn(n, closedList))
  5         then return RemoveSeen(Tail(nodeList), openList, closedList)
  6         else return Cons(n,RemoveSeen(Tail(nodeList), openList, closedList)

OccursIn(node, listOfPairs)
  1 if Null(listOfPairs)
  2   then return FALSE
  3   else if n = Head(Head(listOfPairs)
  4          then return TRUE
  5          else return OccursIn(node, Tail(listOfPairs))

MakePairs(list, parent)
  1 if Null(list)
  2   then return ()
  3   else return Cons(MakeList(Head(list), parent),
                                          MakePairs(Tail(list), parent))
```

FIGURE 2.18 DFS treats *OPEN* like a stack adding new nodes at the head of the list. The function *RemoveSeen* removes any nodes that are already on *OPEN* or *CLOSED.* The function *MakePairs* takes the nodes returned by *RemoveSeen* and constructs node pairs with the parent node, which are then pushed onto *OPEN.*

DFS treats *OPEN* like a stack.

The search tree as seen by *DFS*

OPEN	*CLOSED*
((S, Nil))	( )
((A, S)(B, S)(C, S))	((S, Nil))
((E, A)(B, S)(C, S))	((A, S)(S, Nil))
((D, E)(G, E)(B, S)(C, S))	((E, A)(A, S)(S, Nil))
((G, E)(B, S)(C, S))	((D, E)(E, A)(A, S)(S, Nil))

Again, the search terminates when G is picked.

The program reconstructs the path and returns SAEG, which is the path found from S to G. The back pointers are:

G→E, E→A, A→S

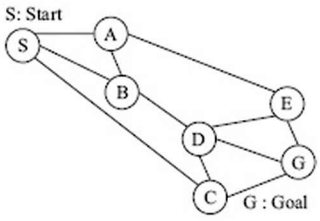


FIGURE 2.19 The search tree as seen by Depth First Search.

2.4 Breadth First Search (BFS)

Breadth First Search (Figure 2.20), on the other hand, is very conservative. It inspects the nodes generated on a first come, first served basis. That is, the nodes form a queue to be inspected. The data structure Queue implements the first in, first out order. The *BFS* algorithm is only a small modification of the *DFS* algorithm. Only the order in the append statement where *OPEN* is updated needs to be changed.

```
BreadthFirstSearch()
  1  open ← ((start, NIL))
  2  closed ← ()
  3  while not Null(open)
  4    do nodePair ← Head(open)
  5       node ← Head(nodePair)
  6       if GoalTest(node) = TRUE
  7          then return ReconstructPath(nodePair, closed)
  8          else closed ← Cons(nodePair, closed)
  9               children ← MoveGen(node)
  10              noLoops ← RemoveSeen(children, open, closed)
  11              new ← MakePairs(noLoops, node)
  12              open ← Append(Tail(open), new)
  13 return "No solution found"
```

FIGURE 2.20 In *BFS*, the order in append is reversed. OPEN becomes a QUEUE.

This small difference in the order of adding nodes to *OPEN* produces radically different behaviours from the two search methods. *DFS* dives down expanding the search tree. Traditionally, we draw it going down the leftmost side first. *BFS* on the other hand pushes into the tree, examining it layer by layer. Consider the following small search tree (Figure 2.21) that would be built if every node had three successors, and the total depth was three.

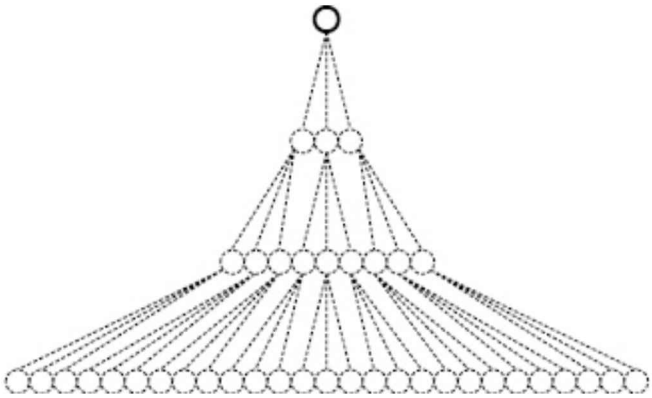


FIGURE 2.21 A tiny search tree.

Let us look at the tree generated by search at the point when the search is about to examine the fifth node, shown in black for both *DFS* and *BFS* in Figure 2.22.

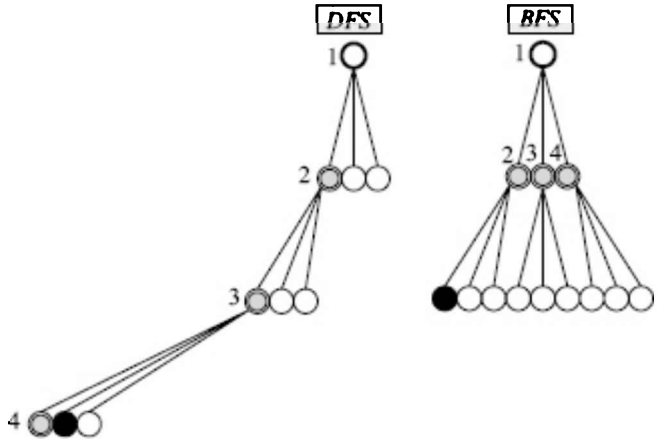


FIGURE 2.22 The search trees generated by *DFS* and *BFS* after four expansions.

DFS dives down the left part of the tree. It will examine the fifth and the sixth nodes and then *backtrack* to the previous level to examine the sibling of the third node it visited. *BFS,* on the other hand, looks at all the nodes at one level before proceeding to the next level. When examining its fifth node, it has started with level two.

How do the two search methods compare? At this point, we introduce the basis for comparison of search methods. The search algorithm is evaluated on the following criteria:

- **Completeness:** Does the algorithm always find a solution when there exists one? We also call a complete search method *systematic.* By this we mean that the algorithm explores the entire search space before reporting failure. If a search is not systematic

then it is *unsystematic,* and therefore incomplete.

- **Time Complexity:** How much time does the algorithm run for before finding a solution? In our case, we will get a measure of time complexity by the number of nodes the algorithm picks up before it finds the solution.
- **Space Complexity:** How much space does the algorithm need? We will use the number of nodes in the *OPEN* list as a measure of space complexity. Of course, our algorithm also needs to store *CLOSED,* and it does grow quite rapidly; but we will ignore that for the time being. There are two reasons for ignoring *CLOSED.* Firstly, it turns out that on the average it is similar for most search methods. Secondly, we also look at search methods which do not maintain the list of seen nodes.
- **Quality of Solution:** If some solutions are better than others, what kind of solution does the algorithm find? A simple criterion for quality might be the length of the path found from the start node to the goal node. We will also look at other criteria in later chapters.

2.5 Comparison of BFS and DFS

How do Depth First Search and Breadth First Search compare? Let us look at each of the criteria described above.

2.5.1 Completeness

Both Depth First Search and Breadth First Search are complete for finite state spaces. Both are systematic. They will explore the entire search space before reporting failure. This is because the termination criterion for both is the same. Either they pick the goal node and report success, or they report failure when *OPEN* becomes empty. The only difference is where the new nodes are placed in the *OPEN* list. Since for every node examined, all unseen successors are put in *OPEN*, both searches will end up looking at all reachable nodes before reporting failure. If the state space is infinite, but with finite branching then depth first search may go down an infinite path and not terminate. Breadth First Search, however, will find a solution if there exists one. If there is no solution, both algorithms will not terminate for infinite state spaces.

2.5.2 Time Complexity

The time complexity is not so straightforward to estimate. If the search space is finite then in the worst case, both the search methods will search all the nodes before reporting failure. When the goal node exists then the time taken to find it depends upon where the goal node is in the state space. Both *DFS* and *BFS* search the space in a predefined manner. The

time taken depends upon where the goal happens to be in their path. Figure 2.23 shows the progress of both the searches on our state space.

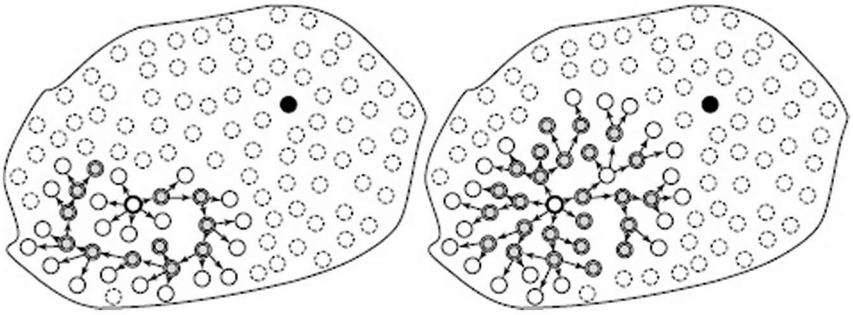


FIGURE 2.23 *DFS* on the left dives down some path. *BFS* on the right pushes slowly into the search space. The nodes in dark grey are in *CLOSED,* and light grey are in *OPEN.* The goal is the node in the centre, but it has no bearing on the search tree generated by *DFS* and *BFS.*

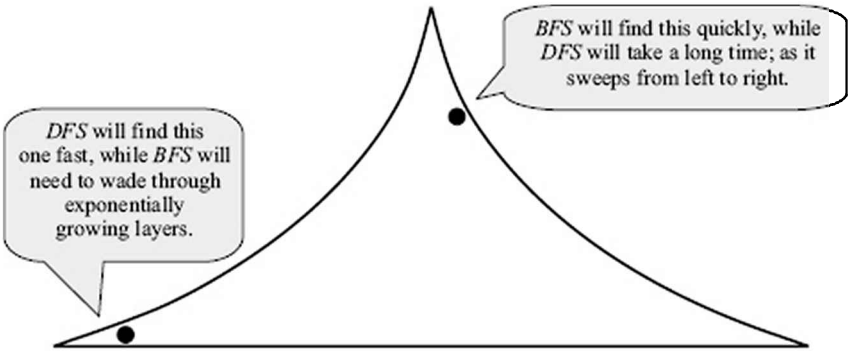


FIGURE 2.24 *DFS* vs. *BFS* on a finite tree.

If the state space is infinite then *DFS* could dive down an infinitely long path and never recover! It would then make no sense to talk of time complexity. Let us assume, for analysis, a domain where the search tree is of bounded depth and of constant branching factor. The tree is shown in Figure 2.24, and depicts two cases that are specifically good for the two searches respectively. A goal node on the bottom left will be found rather quickly by a *DFS* that dives into that area[2] when it gets going. A goal node in the shallow part of the tree on the right would be found faster by *BFS* that explores the tree level by level. Basically, the *DFS* finds nodes on the left side in the search tree faster, while *BFS* finds nodes closer to the starting node faster.

For quantitative analysis, let us look at a search tree in which the goal is always at the same depth *d* in a search tree with a fixed branching factor *b.* Of course, this may not be true in real problems but making this assumption will give us some idea of the complexity of the two algorithms for finding goals at depth *d.* We further assume that the search tree is of

bounded depth *d*. An example of such a domain is the *n-queens* domain, where in each row a queen can be placed in any of the *n* columns, and a solution will have queens placed in all the *n* rows. The goal node can be anywhere from the extreme left to the extreme right.

The root node of the search tree is the start node. At level (depth) 1, there are *b* nodes. At depth 2, there are *b*x*b* nodes, and so on. At level *d* there are b^d nodes. Also, given a tree of depth *d*, the number of internal nodes *I* is given by

$$I = 1 + b - b^2 + \ldots + b^{d-1}$$
$$= (b^d - 1) / (b - 1)$$

Table (2.1) illustrates the numbers for $b = 10$, and gives us a good idea of how rapidly the search tree grows. At depth = 13, there are 10^{13} leaves and about 1.1×10^{12} internal nodes. An interesting point to note is that the number of internal nodes is significantly smaller than the number of leaves. This means that most of the work by the search is done on the deepest level. This is, of course, a characteristic of an exponentially growing space.

Table 2.1 The number of leaves and internal nodes in a search tree

Depth	*Leaves*	*Internal nodes*
0	1	0
1	10	1
2	100	11
3	1000	111
4	10000	1111
5	100000	11111
6	1000000	111111
7	10000000	1111111
8	100000000	11111111
9	1000000000	111111111
10	10000000000	1111111111
11	100000000000	11111111111
12	1000000000000	111111111111
13	10000000000000	1111111111111

Let us now compare the time taken by our two search methods to find a goal node at the depth *d*. The time is estimated by looking at the size of the *CLOSED* list, which is the number of nodes examined before finding the goal. This assumes that the algorithms take constant time to examine each node. This is strictly not true. Remember that just to check whether a node has been seen earlier, one has to look into the *CLOSED* list. And for that, one has to look into *CLOSED* which grows as more and more nodes are added to it. This is likely to become costlier as the search progresses. Nevertheless, we are only interested in relative estimates, and will adopt the simpler approach of counting the nodes examined before termination. Also, in practice, one might use a *hash table* to store the nodes in *CLOSED*, and then checking for existence could in fact be done in constant time.

DFS

If the goal is on the extreme left then *DFS* finds it after examining *d* nodes. These are the ancestors of the goal node. If the goal is on the extreme right, it has to examine the entire search tree, which is b^d nodes. Thus, on the average, it will examine N_{DFS} nodes given by,

$$\begin{aligned} N_{DFS} &= [(d+1) + (b^{d-1} - 1)/(b-1)] / 2 \\ &= (b^{d-1} - bd + b - d - 2) / 2(b-1) \\ &= b^d/2 \text{ for large } d \end{aligned}$$

BFS

The search arrives at level *d* after examining the entire subtree above it. If the goal is on the left, it picks only one node, which is the goal. If the goal is on the right, it picks it up in the end (like the *DFS),* that is, it picks b^d nodes at the level *d*. On an average, the number of nodes N_{BFS} examined by *BFS* is

$$\begin{aligned} N_{BFS} &= (b^d - 1)/(b-1) + (1 - b^d)/2 \\ &\approx (b^{d-1} + b^d + b - 3) / 2(b-1) \\ &\approx b^d (b-1) / 2b \text{ for large } d \end{aligned}$$

Thus, we can see that the *BFS* does marginally worse than *DFS.* In fact,

$$N_{BFS} / N_{DFS} = (b+1)/b$$

so that for problems with a large branching factor, both tend to do the same work.

Box 2.4: Combex: Unimaginably Large Numbers

Even simple problems can pose a number of choices that we find difficult to comprehend. Consider some search problem where the branching factor is ten, and the solution is twenty moves long. Notice that even the Rubik's cube generates a larger search space. Real world problems will have larger branching factors.

To find the solution, the algorithm may have to examine 10^{20} nodes. How long will this take? Assume that you have a fast machine on which you can check a million nodes a second. Thus, you will need $10^{(20-6)} = 10^{14}$ seconds. Assume conservatively that there are 100000 seconds in a day (to simplify our numbers). Assume a thousand days to a year, and you would need $10^{14-8} = 10^6$ years or a million years.

Surely, you are not willing to wait this long!

In an interesting observation, Douglas Hofstadter (Hofstadter

1986) has talked about our inability to comprehend large numbers. Anything beyond a few thousands is "huge". Very often two million and two billion look "similar" to us. And numbers like 10^{20} and 10^{30} don't give us the feel of looking at two numbers, one of which is ten billion times larger than the other. Also, interestingly, George Gamow had written a book called "One, two, three, … Infinity" (Gamow 71) to highlight a similar observation about comprehending large numbers.

2.5.3 Space Complexity

For assessing space complexity, we analyse the size of the *OPEN* list. That is, the number of candidates that the search keeps pending for examination in the future. Here again, we are ignoring the size of *CLOSED,* which as we know, does grow exponentially, but our interest is in relative measures. Moreover, we will later look at ways to prune the *CLOSED* list in more detail. Figure 2.25 shows the *OPEN* and the *CLOSED* lists at different stages of search in a depth-bounded tree of branching factor 3. The double circle shaded nodes are the ones on *CLOSED,* and the unfilled nodes are the ones on *OPEN*.

DFS

In a general search tree, with a branching factor *b*, *DFS* dives down some branch, at each level, leaving (b – 1) nodes in *OPEN.* When it enters the depth *d,* it has at most O_{DFS} nodes in the *OPEN* list, where

$$O_{\text{DFS}} = (b-1)(d-1) + b$$
$$= d(b-1) + 1$$

Thus, the size of *OPEN* is *linear* with depth. In fact, as the search progresses from the left to right, the number of candidates at each level decreases, as shown in Figure 2.26.

Note that the process of backtracking happens automatically when search reaches a dead end. If the examined node has no successors, the next node in *OPEN* is then picked up. This could be a sibling of the node just examined, or if it has no siblings left, a sibling of its parent. Thus, as search exhausts some deeper part of the tree, it automatically reverts to shallower nodes waiting in the *OPEN* list.

BFS

Breadth First Search pushes into the tree level by level. As it enters each level at depth *d,* it sees all the b^d nodes ahead of it in *OPEN.* By the time it finishes with these nodes, it has generated the entire next level and stored them in *OPEN.* Thus, when it enters the next level at depth (*d* + 1),

it sees $b^{(d+1)}$ nodes in the *OPEN* list.

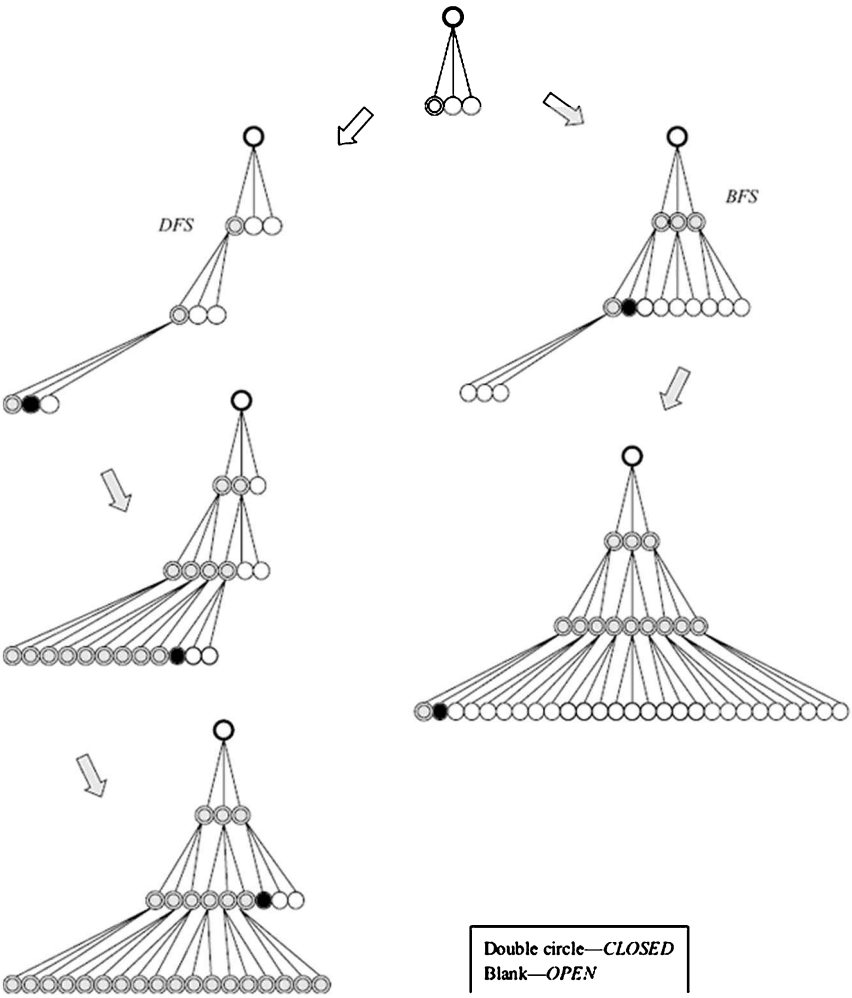


FIGURE 2.25 The *OPEN* and *CLOSED* for *DFS* and *BFS* on the tiny search tree.

This then is the main drawback of *BFS.* The size of *OPEN* grows exponentially with depth. This stands out when one looks at *DFS,* managing its search with an *OPEN* list that grows only linearly with depth.

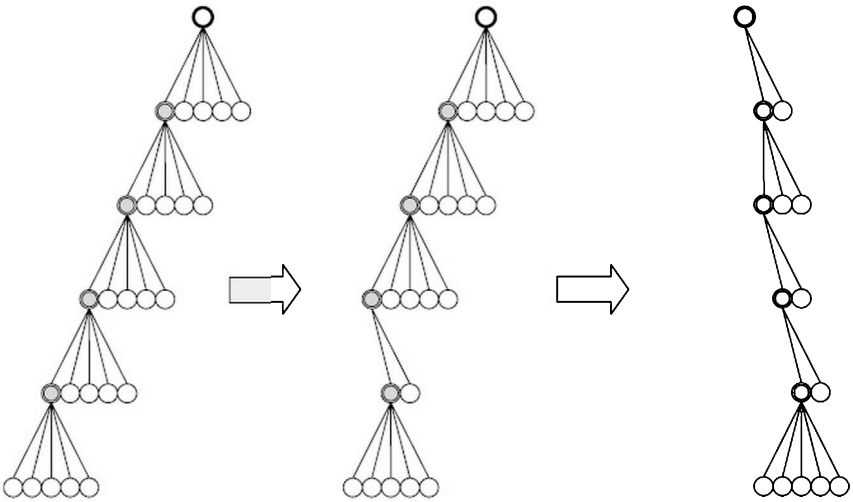


FIGURE 2.26 With a branching factor of 5, at depth 5, there are 5(5 – 1) +1 = 21 nodes in the OPEN to begin with. But as DFS progresses and the algorithm backtracks and moves right, the number of nodes on OPEN decreases.

2.6 Quality of Solution

Our formulation of problem solving does not include any explicit cost for the moves or choices made. Consequently, the only quality measure we can talk of is the length of the solution. That is, we consider solutions with smaller number of moves as better.

Where *BFS* loses out on space complexity, it makes up on the quality of the solution. Since it pushes into the search space level by level, the Breadth First Search inspects candidate solutions in increasing order of solution length. Consequently, it will always find the shortest solution. This is a major advantage in many domains.

Depth First Search on the other hand, dives down into the search tree. It backtracks if it reaches a dead end, and tries other parts of the search space. It returns the first solution found, which holds no guarantee that it will be the shortest one. Given a search tree with two goal nodes as shown in Figure 2.24, *DFS* will find the longer solution deeper in the search tree. Thus, it can find a non-optimal solution. On the other hand, since *BFS* pushes into the search tree, it will always find the shortest solution. So, this is one feature where *BFS* is better than *DFS*.

In fact, if the search space is infinite, there is a danger that *DFS* will go into a never-ending path and never return a solution! Consider the task of finding a pair of two integers <*m*,*n*> which satisfy *some* given property. Let the *moveGen* for each state <*x, y*> return two states <*x*+1,*y*> and <*x*,*y*+1>. Let the goal state be <46, 64> and the start state be <0,0>. The reader is encouraged to simulate the problem on paper and verify that *DFS* would get lost in an infinite branch.

One way to guard against this possibility is to impose a depth bound on the search algorithm. This means that the search space is prohibited from going beyond a specified depth. This could be done if for some reason one wanted solutions only within a certain length.

2.7 Depth Bounded *DFS (DBDFS)*

Figure 2.20 shows a depth bounded *DFS* algorithm. It is different from *DFS,* in that it takes a parameter for depth bound. Given any node, it calls moveGen only if it is within the given depth bound. The node representation is changed to include depth of node as the third parameter, apart from the parent link, which is the second parameter. The function *makePairs* is modified appropriately to compute the depth of a node, by adding one to the depth of its parent.

The astute reader would have observed that implementing *DBDFS* would have been simpler with the node representation that stored the complete path. One would just have to look at the length of the node to determine whether one is within the allowable bound. And in addition, the task of reconstructing the path is also simplified. Obviously, various design choices will have to be made while implementing solutions for specific problems.

```
DepthBoundedDFS(start, depthBound)
  1 open ← ((start, NIL, 0))
  2 closed ← ()
  3 while not Null(open)
  4   do nodePair ← Head(open)
  5   node ← Head(nodePair)
  6   if GoalTest(node) = TRUE
  7       then return ReconstructPath(nodePair, closed)
  8       else closed ← Cons(nodePair, closed)
  9          if Head(Rest(Rest(nodePair))) < depthBound
 10          then children ← MoveGen(node)
 11               noLoops ← RemoveSeen(children, open, closed)
 12               new ← MakePairs(noLoops, node, Head(Rest(Rest(nodePair))))
 13               open ← Append(new, Tail(open))
 14   return "No solution found"

MakePairs(list, parent, depth)
1 if Null(list)
2   then return ()
3   else return (Cons(MakeList(Head(list), parent, depth+1)),
4                 MakePairs(Tail(list), parent, depth))
```

FIGURE 2.27 Depth Bounded DFS generates new nodes only within a defined boundary.

Performance wise, DBDFS is like *DFS* on a finite tree. However, given that the depth bound is artificially imposed, the algorithm is not complete in the general case.

2.8 Depth First Iterative Deepening *(DFID)*

The algorithm Depth First Iterative Deepening *(DFID)* combines the best features of all the algorithms described above. It does a series of depth first searches with *increasing* depth bounds. Since in every cycle it does a *DFS* with bound incremented by one, whenever it finds a solution it would have found the shortest solution. In this respect, it is like *BFS*. New nodes are explored one level at a time. On the other hand, within each cycle it does a *DBDFS.* Therefore, its memory requirements are those of *DFS,* that is, memory requirements grow linearly with depth.

```
DepthFirstIterativeDeepening(start)
  1 depthBound ← 1
  2 while TRUE
  3    do DepthBoundedDFS(start, depthBound)
  4       depthBound ← depthBound + 1
```

FIGURE 2.28 *DFID* does a series of *DBDFSs* with increasing depth bounds.

The high level algorithm described above ignores the possibility of a finite search space with no solution. In that situation, the above algorithm will loop endlessly. The detailed algorithm given below keeps a count of the number of nodes examined in each call of *DBDFS.* If the count is the same in two successive cycles, that is no new nodes are generated, the algorithm *DFID* reports failure.

```
DepthFirstIterativeDeepening(start)
  1 depthBound ← 1
  2 previousCount ←  0
  3 newNodes  ← YES
  4 repeat
  5   count ←  0
  6   open  ← ((start, NIL, 0))
  7   closed  ← ()
  8   while not Null(open)
  9     do  nodePair ← Head(open)
 10         node ← Head(nodePair)
 11         if GoalTest(node) = TRUE
 12           then return ReconstructPath(nodePair, closed)
 13           else closed ← Cons(nodePair, closed)
 14             if Head(Rest(Rest(nodePair))) < depthBound
 15               then children ← MoveGen(node)
 16                 noLoops ← RemoveSeen(children, open, closed)
 17                 new ← MakePairs(noLoops, node,
 18                        Head(Rest(Rest(nodeP air))))
 19                 open ← Append(new, Tail(open))
 20                 count ←  count + Length(new)
 21   if previousCount = count
 22     then newNodes ←  NO
 23   previousCount ←  count
 24   depthBound ← depthBound + 1
 25 until newNodes = NO
 26 return "No solution found"
```

FIGURE 2.29 *DFID*—the algorithm in detail.

Thus, the algorithm *DFID* finds the shortest solution using only linear space. Is there a catch somewhere? In a way there is, but only a small one. The *DFID* algorithm does a series of searches. In each search, it explores a new level of nodes. But for inspecting these new nodes, it has to generate the tree all over again. That is, for exploring the new level, it has to pay the additional cost of regenerating the internal nodes of the search tree all over again.

The question one should ask is how significant is the above cost? The new nodes are the leaves of the search tree at the end of that cycle. What then, is the ratio of the number of internal nodes I (the extra cost) to the number of leaves $\mathcal{L}$ (the new nodes) in a tree?

The ratio is the highest for binary trees. The number of internal nodes is just one less than the number of leaves. So, the number of nodes inspected is at most twice as in *BFS.* In general, for inspecting b^d new nodes at level d, one has to inspect $(b^d - 1) / (b - 1)$ extra nodes. The ratio of internal nodes to leaves tends to $1/(b - 1)$ for large d.

Every time *DFID* inspects the leaves at the depth d, it does extra work of inspecting all the internal nodes as well. That is, every time *BFS* scans $\mathcal{L}$ nodes at depth d, *DFID* inspects $L + I$ nodes. Thus, it does $(\mathcal{L} + I)/\mathcal{L}$ times extra work as compared to *BFS.* This ratio tends to $b/(b - 1)$ for large d, and as the branching factor becomes larger, the number of extra nodes to be inspected becomes less and less significant. A look at Table 2.1 above shows that when $b = 10$, the number of internal nodes is about 11% of the leaves.

Box 2.5: Leaves Vastly Outnumber Internal Nodes

Given a tree with branching factor b, a simple way to compute the ratio of leaves to internal nodes is by thinking of the tree as a tournament in which b players compete, and one winner emerges at each event (internal node). That is, at each (event) internal node, b – 1 players are eliminated. Thus, if there are $\mathcal{L}$ players, and only one emerges at the root the number of events (internal nodes), I must satisfy

$$\mathcal{L} = (b - 1)\, I + 1$$

Thus

$$I = (\mathcal{L} - 1)/(b - 1), \text{ and } I/\mathcal{L} = 1/(b - 1) \text{ for large } \mathcal{L}.$$

Also

$$(\mathcal{L} + I)/\mathcal{L} = b/(b - 1) \text{ for large } \mathcal{L}.$$

The work done by DFID can also be computed by summing up the work done by all the Depth First Searches that it is made up of,

to arrive at the same result.

$$\textit{That is } N_{DFID} = \Sigma_{j=0 \text{ to } d} (b^{j+1} - 1)/(b-1)$$

Thus, the cost of repeating work done at shallow levels is not prohibitive

Thus, the extra work that *DFID* has to do becomes insignificant as the branching factor becomes larger. And the advantage gained over *BFS* is a major one, the space requirement drops from exponential to linear.

The above discussion shows that as search algorithms have to search deeper and deeper then at every level they have to do much more work than at all the previous levels combined. This is the ferocious nature of the monster *CombEx* that problem solving finds itself up against.

Nevertheless, the fact remains that all the three search algorithms explore an exponentially growing number of states, as they go deeper into the search space. This would mean that these approaches will not be suitable as the problem size becomes larger, and the corresponding search space grows. The *DFS* and *DFID* algorithms need only a linear amount of space to store *OPEN.* They still need exponential space to store *CLOSED* whose size is correlated with time complexity. We will need to do better than that.

There are basically two approaches to improve our problem solving algorithms. One is to try and reduce the space in which the algorithm has to search for solutions. This is what is done in *Constraint Propagation.* The other is to guide the search problem solving method to the solution. Knowledge based methods tend to exploit knowledge acquired earlier to solve a current problem. These methods range from fishing out complete solutions from the problem solver's memory, to providing domain insights to a search-based first principles approach.

Observe that the search methods seen so far are completely oblivious of the goal, except for testing for termination. Irrespective of what the goal is or where the goal is in the solution space, each search method described in this chapter explores the space in one single order, which they blindly follow. They can be called *blind* or *uninformed* search methods.

In later chapters, we will find the weapon to fight *CombEx*, our adversary that confounds our algorithms with unimaginably large numbers. Our weapon will be *knowledge*. Knowledge will enable our algorithms to cut through the otherwise intractable search spaces to arrive at solutions in time to be of use to us. In the next chapter, we look at search methods that use some domain knowledge to impart a sense of direction to search, so that it can decide where to look in the solution space in a more informed manner.

Exercises

1. The *n-queens* problem is to place *n queens* on an n-by-n chessboard, such that no queen attacks another, as per chess rules. Pose the problem as a search problem.
2. Another interesting problem posed on the chessboard is the knight's tour. Starting at any arbitrary starting point on the board, the task is to move the knight such that it visits every square on the chessboard exactly once. Write an algorithm to accept a starting location on a chessboard and generate a knight's tour.

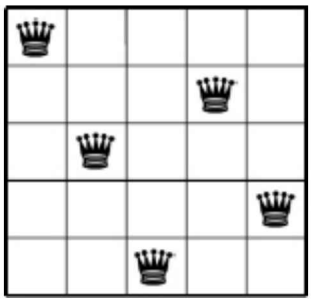


FIGURE 2.30 A 5-queens solution.

Box 2.6: Self Similarity

An interesting observation has been made about chessboards that have sides that are powers of 5. The adjoining figure displays one solution for 6. the 5-queens problem.

If we want to solve the 25 queens problem, then we can think of the 25 × 25 board as twenty five 5 × 5 boards. Then, in the five 5 × 5 boards that correspond to the queens in the above solution, we simply place the 5 × 5 solution board as shown in Figure 2.31. Thus, the 25 × 25 board solution is *similar* to the 5 × 5 board solution. Such similarity to a part of itself is known as *self similarity* and is a key property of fractals (see for example (Barnsley, 1988)). Clouds, coastlines, plants and many artifacts in nature have this property. Observe that with this *knowledge,* arbitrarily large *n*-queen problems that are powers of five can be solved quite easily, whereas the search would have taken very long.

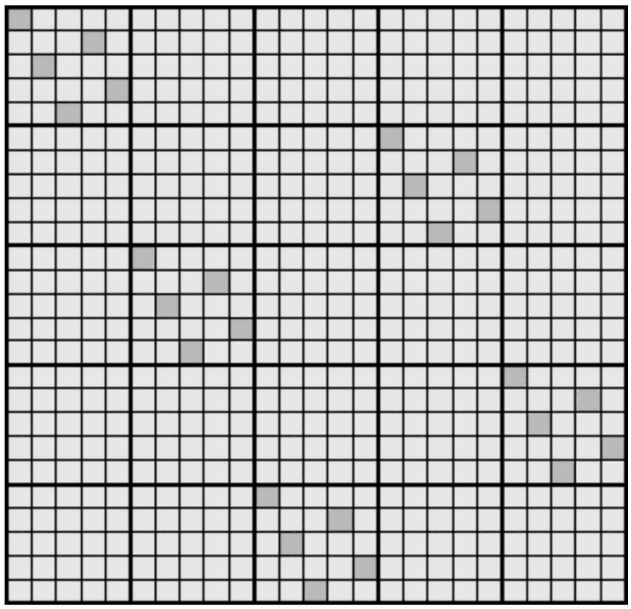


FIGURE 2.31 A 25-queens solution constructed from a 5-queens solution.

3. In the *rabbit leap problem,* three east-bound rabbits stand in a line blocked by three west-bound rabbits. They are crossing a stream with stones placed in the east west direction in a line. There is one empty stone between them.

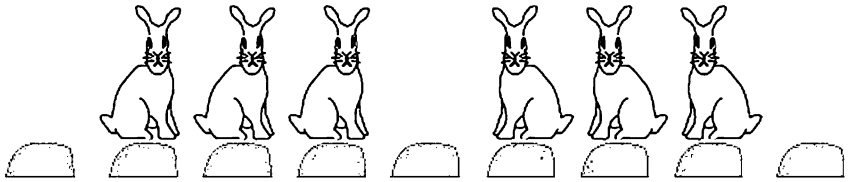


FIGURE 2.32 Rabbits waiting to cross. Each rabbit can jump over one, but not more than that. How can they avoid getting into a deadlock?

The rabbits can only move forward one step or two steps. They can jump over one rabbit if the need arises, but not more than that. Are they smart enough to cross each other without having to step into the water? Draw the state space for solving the problem, and find the solution path in the state space graph.

4. Given that you have a set of shopping tasks to be done at N locations, find a feasible order of the shopping tasks. The constraints are the different closing times of the N shops, given a closing time table. An $N \times N$ array D gives the time it takes to go between shops, where d_{ij} being the time taken from the i^{th} shop to the j^{th} shop.

Assume that the time taken for the actual shopping is negligible. ***Hint:*** Augment the state representation to have time. If the time of reaching a shop is later than its closing time, try another solution.

5. Amogh, Ameya and their grandparents have to cross a bridge over the river within one hour to catch a train. It is raining and they have only one umbrella which can be shared by two people. Assuming that no one wants to get wet, how can they get across in an hour or less? Amogh can cross the bridge in 5 minutes, Ameya in 10, their grandmother in 20, and their grandfather in 25. Design a search algorithm to answer the question.
6. The *AllOut* game is played on a 5 by 5 board. Each square can be in two positions, *ON* or *OFF.* The initial state is some state, where at least one square is *ON.* The moves constitute of clicking on a particular square. The effect of the click is to toggle the positions of its four neighbouring squares. The task is to bring all squares to *OFF* position. Pose the above problem as a state space search problem.
7. Ramesh claims that a given map can be coloured with three colours, such that no adjacent countries have the same colour. The map is represented as a planar graph with nodes as countries and arcs between countries that are adjacent to each other. Design a search program to test his claim for any given problem.
8. In the algorithms given in this chapter, the list *CLOSED* is searched in a linear fashion to find out whether a node has been visited earlier or not. Devise a faster approach to accomplish this task.
9. In the following graph, the node *A* is the start node and nodes *J, G* and *R* are goal nodes. The tree is being searched by the *DFID* algorithm, searching left to right. Write the sequence of nodes inspected by *DFID* till termination.

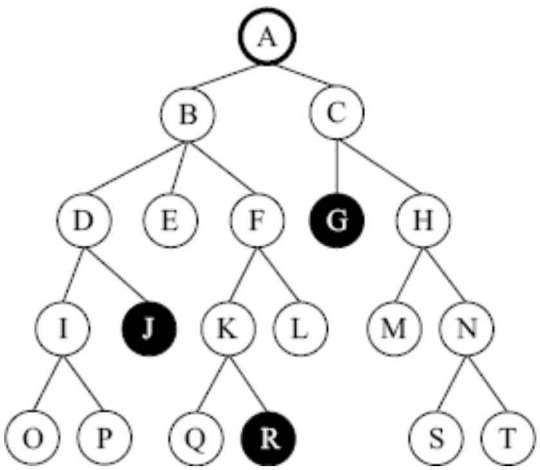


FIGURE 2.33 A small search tree. Assume that the algorithm searches it from left to right.

10. Given the *moveGen* function in the table below, and the

corresponding state space graph, the task is to find a path from the start node *S* to the goal node *J*.

```
   moveGen
S → (D C B A)
A → (S B J E)
B → (S F A)
C → (S D H G)
D → (S I C)
E → (A J K)
F → (B K J)
G → (C L)
H → (C I M L)
I → (D H)
J → (A F E)
K → (E F)
L → (G H M)
M → (L H)
```

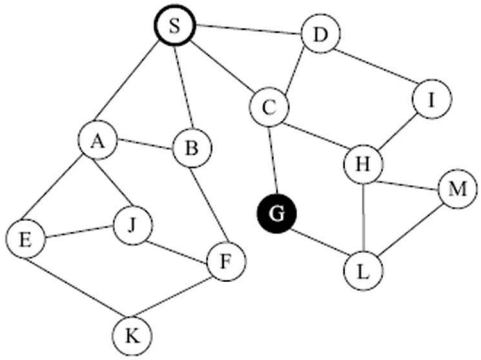


FIGURE 2.34 A small search problem. The task is to find a path from the start node *S* to the goal node *J*.

Show the *OPEN* and *CLOSED* lists for the *DFS* and the *BFS* algorithms. What is the path found by each of them?

11. The search algorithms depicted in this chapter maintain an *OPEN* list of candidate states. In some domains, the description of a move may need much less space than the description of a state. Show how the *DFS* algorithm can be modified by storing a list of move pairs in the list *OPEN*. Each move pair consists of a forward move and a backward move, to and fro to a successor. What is the space complexity of the resulting algorithm?
12. How can the same technique be applied to *BFS?* What is the resulting complexity?
13. Show the order in which *DFID* applies goalTest to the nodes of the above graph. What is the path it finds?
14. What is the effect of not maintaining the *CLOSED* list in *DFID* search? Discuss with an example. Will the algorithm terminate? Will it find the shortest solution?
15. Show that the effort expended by *DFID* is the same as its constituent depth first searches.
16. Which of the following is more amenable to parallelization? *DFS, BFS,* or *DFID?* Justify your answer.
17. Explain with reasons which search algorithms described in this chapter will be used by you for the following problems:
    (a) A robot finding its way in a maze.
    (b) Finding a winning move in a chessboard.
    (c) Finding all winning moves in a chessboard.
    (d) A sensor trying to route a packet to another sensor (Assume the network topology is known.
    Hint: Sensors are limited memory devices.) Justify your answers

with a description of your search primitives and state definitions for each problem.

---

[1] In an alternate version of the problem, if the missionaries outnumber the cannibals, they will convert them.

[2] Incidentally, while we have assumed in both *DFS* and *BFS* that the search picks up the leftmost of siblings, this is really determined by the order in which *moveGen* generates them. It is merely easier and more convenient for us to visualize the two algorithms searching from the left to the right.