State Space Representation and Search

1. Introduction

In this section we examine the concept of a **state space** and the different **searches** that can be used to explore the search space in order to find a solution. Before an AI problem can be solved it must be represented as a state space. The state space is then searched to find a solution to the problem. A state space essentially consists of a set of nodes representing each state of the problem, arcs between nodes representing the legal moves from one state to another, an initial state and a goal state. Each state space takes the form of a tree or a graph.

Factors that determine which search algorithm or technique will be used include the type of the problem and the how the problem can be represented. Search techniques that will be examined in the course include:

- Depth First Search
- Depth First Search with Iterative Deepening
- Breadth First Search
- Best First Search
- Hill Climbing
- Branch and Bound Techniques
- A* Algorithm

2. Classic AI Problems

Three of the classic AI problems which will be referred to in this section is the Traveling Salesman problem and the Towers of Hanoi problem and the 8 puzzle.

Traveling Salesman

A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route that the salesman should follow for the shortest trip that both starts and finishes at any one of the cities.

Towers of Hanoi

In a monastery in the deepest Tibet there are three crystal columns and 64 golden rings. The rings are different sizes and rest over the columns. At the beginning of time all the rings rested on the leftmost column, and since than the monks have toiled ceaselessly moving the rings one by one between the columns. In moving the rings a larger ring must not be placed on a smaller ring. Furthermore, only one ring at a time can be moved from one column to the next. A simplified version of this problem which will consider involves only 2 or 3 rings instead of 64.

8-Puzzle

1	2	3
8		4
7	6	5

The 8-Puzzle involves moving the tiles on the board above into a particular configuration. The black square on the board represents a space. The player can move a tile into the space, freeing that position for another tile to be moved into and so on.

For example, given the initial state above we may want the tiles to be moved so that the following goal state may be attained.

1	8	3
2	6	4
7		5

3. Problem Representation

A number of factors need to be taken into consideration when developing a state space representation. Factors that must be addressed are:

- What is the goal to be achieved?
- What are the legal moves or actions?
- What knowledge needs to be represented in the state description?
- Type of problem - There are basically three types of problems. Some problems only need a representation, e.g. crossword puzzles. Other problems require a yes or no response indicating whether a solution can be found or not. Finally, the last type problem are those that require a solution path as an output, e.g. mathematical theorems, Towers of Hanoi. In these cases we know the goal state and we need to know how to attain this state
- Best solution vs. Good enough solution - For some problems a good enough solution is sufficient. For example, theorem proving , eight squares. However, some problems require a best or optimal solution, e.g. the traveling salesman problem.

Towers Hanoi

A possible state space representation of the Towers Hanoi problem using a graph is indicated in **Figure 3.1**.

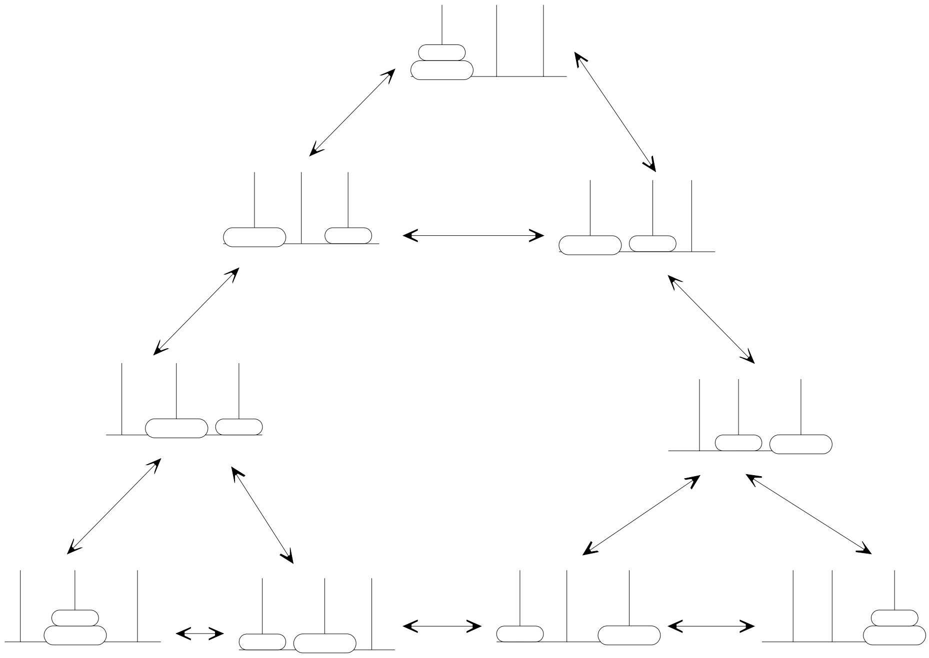


Figure 3.1: Towers of Hanoi state space representation

The legal moves in this state space involve moving one ring from one pole to another, moving one ring at a time, and ensuring that a larger ring is not placed on a smaller ring.

8-Puzzle

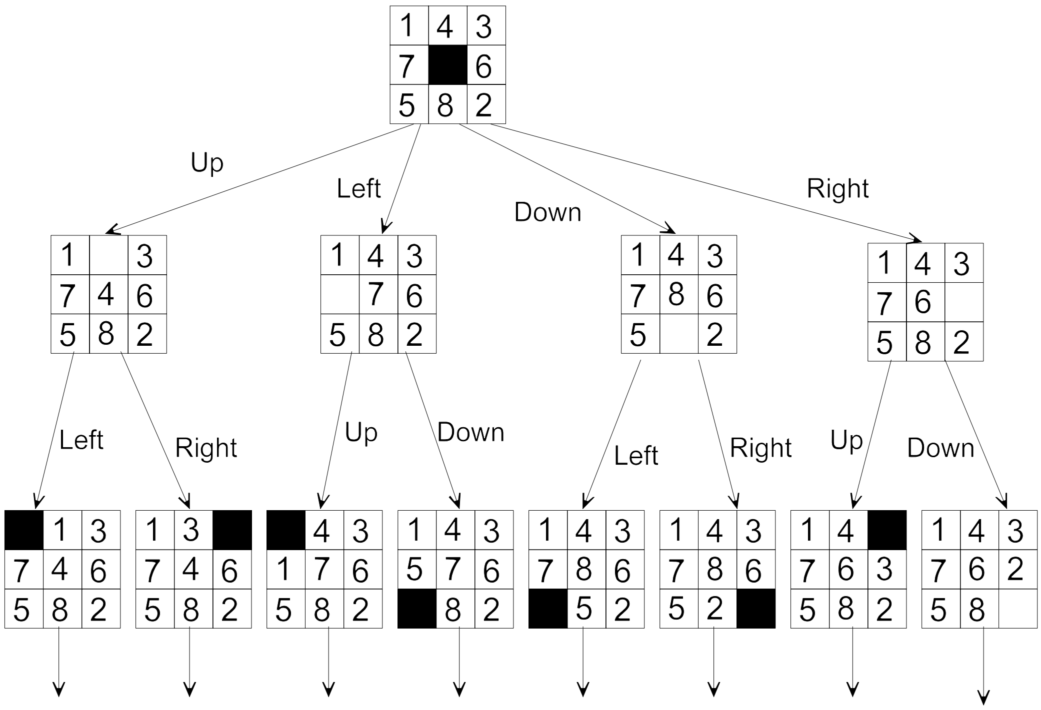


Figure 3.2: Eight-Puzzle Problem state space representation

Although a player moves the tiles around the board to change the configuration of tiles. However, we will define the legal moves in terms of moving the space. The space can be moved up, down, left and right.

Summary:

- A state space is a set of descriptions or states.
- Each search problem consists of:
  - One or more initial states.
  - A set of legal actions. Actions are represented by operators or moves applied to each state. For example, the operators in a state space representation of the 8-puzzle problem are left, right, up and down.
  - One or more goal states.
- The number of operators are problem dependant and specific to a particular state space representation. The more operators the larger the branching factor of the state space. Thus, the number of operators should kept to a minimum, e.g. 8-puzzle: operations are defined in terms of moving the space instead of the tiles.

- Why generate the state space at run-time, and not just have it built in advance?
- A search algorithm is applied to a state space representation to find a solution path. Each search algorithm applies a particular search strategy.

4. Graphs versus Trees

If states in the solution space can be revisited more than once a directed graph is used to represent the solution space. In a graph more than one move sequence can be used to get from one state to another. Moves in a graph can be undone. In a graph there is more than one path to a goal whereas in a tree a path to a goal is more clearly distinguishable. A goal state may need to appear more than once in a tree. Search algorithms for graphs have to cater for possible loops and cycles in the graph. Trees may be more “efficient” for representing such problems as loops and cycles do not have to be catered for. The entire tree or graph will not be generated.

Figure 4.1 illustrates the both a graph a representation of the same the states.

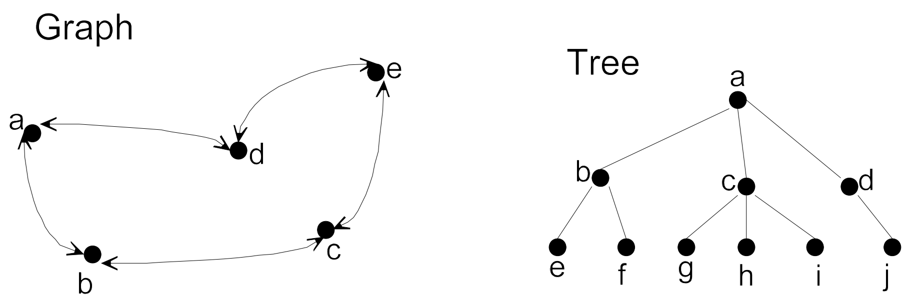


Figure 4.1: Graph vs. Tree

Notice that there is more than one path connecting two particular nodes in a graph whereas this is not so in a tree.

Example 1:

Prove x + (y +z) = y + (z+x) given

L+(M+N) = (L+M) + N ....(A)
M+N = N+M........................(C)

Figure 4.2 represents the corresponding state space as a graph. The same state space is represented as a tree in **Figure 4.3.** Notice that the goal state is represented more than once in the tree.

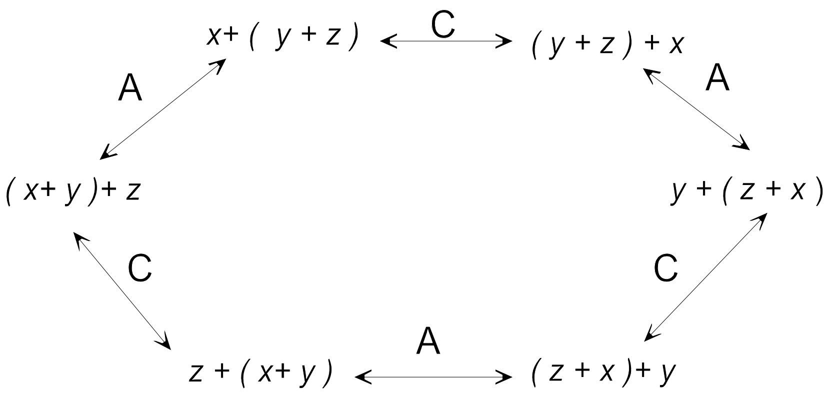


Figure 4.2: Graph state space representation

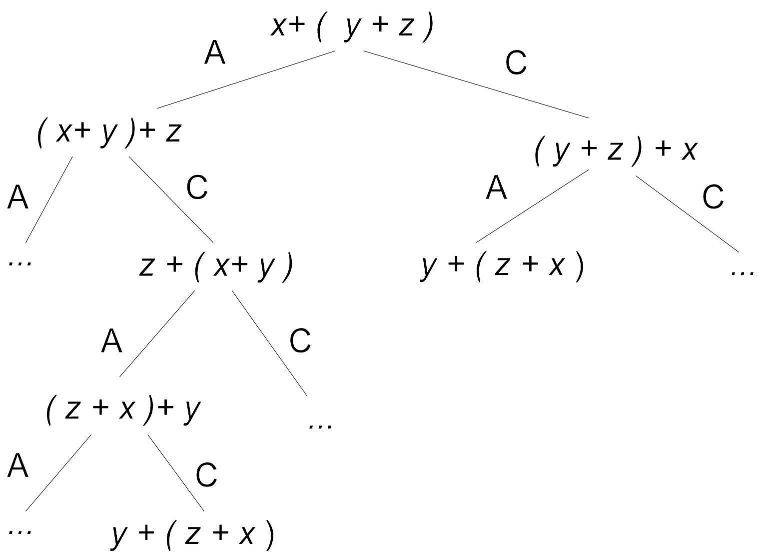


Figure 4.3: Tree state space representation

5. Depth First Search

One of the searches that are commonly used to search a state space is the depth first search. The depth first search searches to the lowest depth or ply of the tree. Consider the tree in **Figure 5.1**. In this case the tree has been generated prior to the search.

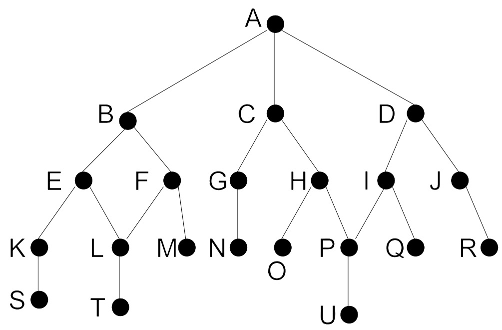


Figure 5.1

Each node is not visited more than once. A depth first search of the this tree produces: A, B, E, K, S, L ,T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R.

Although in this example the tree was generated first and then a search of the tree was conducted. However, often there is not enough space to generate the entire tree representing state space and then search it. The algorithm presented below conducts a depth first search and generates parts of the tree as it goes along.

Depth First Search Algorithm

```
def dfs (in Start, out State)
 open = [Start];
 closed = [];
 State = failure;
 while (open <> []) AND (State <> success)
   begin
    remove the leftmost state from open, call it X;
    if X is the goal, then
      State = success
    else begin
      generate children of X;
      put X on closed
      eliminate the children of X on open or closed
      put remaining children on left end of open
    end else
 endwhile
 return State;
enddef
```

Example 1: Suppose that the letters A,B, etc represent states in a problem. The following moves are legal:

A to B and C
B to D and E
C to F and G
D to I and J
I to K and L

Start state: A Goal state: E , J

Let us now conduct a depth first search of the space. Remember that tree will not be generated before hand but is generated as part of the search. The search and hence generation of the tree will stop as soon as success is encountered or the open list is empty.

	OPEN	**CLOSED**	**X**	**X's Children**	**State**
1.	A	-			failure
2.			A	B, C	failure
3.	B, C	A			failure
4.	C	A	B	D, E	failure
5.	D, E, C	A, B			failure
6.	E, C	A, B	D	I, J	failure
7.	I, J, E, C	A, B, D			failure
8.	J, E, C	A, B, D	I	K, L	failure
9.	K, L, J, E, C	A, B, D, I			failure
10.	L, J, E, C	A, B, D, I,	K	none	failure
11.	J, E, C	A, B, D, I	L	none	failure
12.	E, C	A, B, D, I	J		success

6. Depth First Search with Iterative Deepening

Iterative deepening involves performing the depth first search on a number of different iterations with different depth bounds or cutoffs. On each iteration the cutoff is increased by one. There iterative process terminates when a solution path is found. Iterative deepening is more advantageous in cases where the state space representation has an infinite depth rather than a constant depth. The state space is searched at a deeper level on each iteration.

DFID is guaranteed to find the shortest path. The algorithm does retain information between iterations and thus prior iterations are “wasted”.

Depth First Iterative Deepening Algorithm

```
procedure DFID (intial_state, goal_states)
begin
        search_depth=1
        while (solution path is not found)
        begin
                dfs(initial_state, goals_states) with a depth bound of search_depth
                increment search_depth by 1
        endwhile
end
```

Example 1: Suppose that the letters A,B, etc represent states in a problem. The following moves are legal:

A to B and C
B to D and E
C to F and G
D to I and J
I to K and L

Start state: A Goal state: E , J

Iteration 1: search_depth =1

	OPEN	CLOSED	X	X’s Children	State
1.	A	-			failure
2.			A	B, C	failure
3.	B, C	A			failure
4.	C	A	B		failure
5.	C	A, B			failure
6.		A, B	C		failure
7.		A, B, C			failure

Iteration 2: search_depth = 2

	OPEN	CLOSED	X	X's Children	State
1.	A	-			failure
2.			A	B, C	failure
3.	B, C	A			failure
4.	C	A	B	D, E	failure
5.	D, E, C	A, B			failure
6.	E, C	A, B	D		failure
7.	E, C	A, B, D			failure
8.	C	A, B, D	E		success

7. Breadth First Search

The breadth first search visits nodes in a left to right manner at each level of the tree. Each node is not visited more than once. A breadth first search of the tree in **Figure 5.1**. produces A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.

Although in this example the tree was generated first and then a search of the tree was conducted. However, often there is not enough space to generate the entire tree representing state space and then search it.

The algorithm presented below conducts a breadth first search and generates parts of the tree as it goes along.

Breadth First Search Algorithm

```
def bfs (in Start, out State)
 open = [Start];
 closed = [];
 State = failure;
 while (open <> []) AND (State <> success)
   begin
    remove the leftmost state from open, call it X;
    if X is the goal, then
      State = success
    else begin
      generate children of X;
      put X on closed
      eliminate the children of X on open or closed
      put remaining children on right end of open
    end else
 endwhile
 return State;
enddef
```

Let us now apply the breadth first search to the problem in **Example 1** in this section.

	OPEN	CLOSED	X	X's Children	State
1.	A	-			failure
2.			A	B, C	failure
3.	B, C	A			failure
4.	C	A	B	D, E	failure
5.	C, D, E	A, B			failure
6.	D, E	A, B	C	F, G	failure
7.	D, E, F, G	A, B, C			failure
8.	E, F, G	A, B, C	D	I, J	failure
9.	E, F, G, I, J	A, B, C, D			failure
10.	F, G, I, J	A, B, C, D	E		success

8. Differences Between Depth First and Breadth First

Breadth first is guaranteed to find the shortest path from the start to the goal. DFS may get “lost” in search and hence a depth-bound may have to be imposed on a depth first search. BFS has a bad branching factor which can lead to combinatorial explosion. The solution path found by the DFS may be long and not optimal. DFS more efficient for search spaces with many branches, i.e. the branching factor is high.

The following factors need to be taken into consideration when deciding which search to use:

- The importance of finding the shortest path to the goal.
- The branching of the state space.
- The available time and resources.
- The average length of paths to a goal node.
- All solutions or only the first solution.

9. Heuristic Search

DFS and BFS may require too much memory to generate an entire state space - in these cases heuristic search is used. Heuristics help us to reduce the size of the search space. An evaluation function is applied to each goal to assess how promising it is in leading to the goal. Examples of heuristic searches:

- Best first search
- A* algorithm
- Hill climbing

Heuristic searches incorporate the use of domain-specific knowledge in the process of choosing which node to visit next in the search process. Search methods that include the use of domain knowledge in the form of heuristics are described as “weak” search methods. The knowledge used is “weak” in that it usually helps but does not always help to find a solution.

9.1 Calculating heuristics

Heuristics are rules of thumb that may find a solution but are not guaranteed to. Heuristic functions have also been defined as evaluation functions that estimate the cost from a node to the goal node. The incorporation of domain knowledge into the search process by means of heuristics is meant to speed up the search process. Heuristic functions are not guaranteed to be completely accurate. If they were there would be no need for the search process. Heuristic values are greater than and equal to zero for all nodes. Heuristic values are seen as an approximate cost of finding a solution. A heuristic value of zero indicates that the state is a goal state.

A heuristic that never overestimates the cost to the goal is referred to as an **admissible** heuristic. Not all heuristics are necessarily admissible. A heuristic value of infinity indicates that the state is a “deadend” and is not going to lead anywhere. A good heuristic must not take long to compute. Heuristics are often defined on a simplified or relaxed version of the problem, e.g. the number of tiles that are out of place. A heuristic function h_1 is better than some heuristic function h_2 if fewer nodes are expanded during the search when h_1 is used than when h_2 is used.

Example 1: The 8-puzzle problem

Initial State

2	8	3
1	6	4
7		5

Goal State

1	2	3
8		4
7	6	5

Each of the following could serve as a heuristic function for the 8-Puzzle problem:

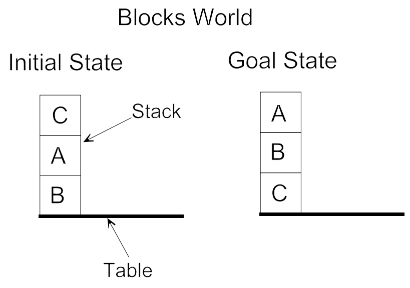
- Number of tiles out of place - count the number of tiles out of place in each state compared to the goal .
- Sum all the distance that the tiles are out of place.
- Tile reversals - multiple the number of tile reversals by 2.

State	Tiles out of place	Sum of distances out of place	2 x the number of direct tile reversals
2 8 3 1 6 4 7 5	5	6	0
2 8 3 1 4 7 6 5	3	4	0
2 8 3 1 6 4 7 5	5	6	0

Figure 8.2: Heuristic functions for the 8-puzzle problem

Exercise: Derive a heuristic

function for the following problem



Experience has shown that it is difficult to devise heuristic functions. Furthermore, heuristics are falliable and are by no means perfect.

10. Best First Search

Figure 10.1 illustrates the application of the best first search on a tree that had been generated before the algorithm was applied

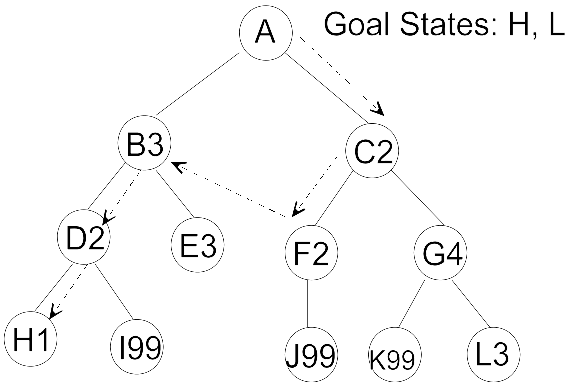


Figure 10.1: Best First Search

The best-first search is described as a general search where the minimum cost nodes are expanded first. The best- first search is not guaranteed to find the shortest solution path. It is more efficient to not revisit nodes (unless you want to find the shortest path). This can be achieved by restricting the legal moves so as not to include nodes that have already been visited. The best-first search attempts to minimize the cost of finding a solution. Is a combination of the depth first-search and breadth-first search with heuristics. Two lists are maintained in the implementation of the best-first algorithm: OPEN and CLOSED. OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet. CLOSED contains those nodes that have already been visited. Application of the best first search: Internet spiders.

The algorithm for the best first search is described below. The algorithm generates the tree/graph representing the state space.

The Best First Search

```
def bestfs(in Start, out State)
 open = [Start],
 close = [],
 State = failure;
 while (open <> []) AND (State <> success)
   begin
    remove the leftmost state from open, call it X;
    if X is the goal, then
      State = success
    else begin
      generate children of X;
      for each child of X do
      case
        the child is not on open or closed
         begin
          assign the child a heuristic value,
          add the child to open,
         end;
        the child is already on open
          if the child was reached by a shorter path then
           give the state on open the shorter path
        the child is already on closed:
         if the child is reached by a shorter path then
          begin
           remove the state from closed,
           add the child to open
         end;
       endcase
     endfor
     put X on closed;
     re-order states on open by heuristic merit (best leftmost);
   endwhile;
     return State;
end.
```

We will now apply the best first search to a problem.

Example 1:

Suppose that each letter represents a state in a solution space. The following moves are legal :

A5 to B4 and C4
B4 to D6 and E5
C4 to F4 and G5
D6 to I7 and J8
I7 to K7 and L8

Start state : A
Goal state : E

The numbers next to the letters represent the heuristic value associated with that particular state.

	OPEN	**CLOSED**	**X**	**X's Children**	**State**
1	A5	-			failure
2			A5	B4, C4	failure
3	B4, C4	A5			failure
4	C4	A5	B5	D6, E5	failure
5	C4, E5, D6	A5, B4			failure
6	E5, D6	A5, B4	C4	F4, G5	failure
7	F4, E5, G5, D6	A5, B4, C4			failure
8	E5, G5, D6	A,5 B4, C4	F4	none	failure
9	E5, G5, D6	A5, B4, C4, F4			failure
10	G5, D6	A5, B4, C4, F4	E5		success

11. Hill-Climbing

Hill-climbing is similar to the best first search. While the best first search considers states globally, hill-climbing considers only local states. The application of the hill-climbing algorithm to a tree that has been generated prior to the search is illustrated in **Figure 11.1**.

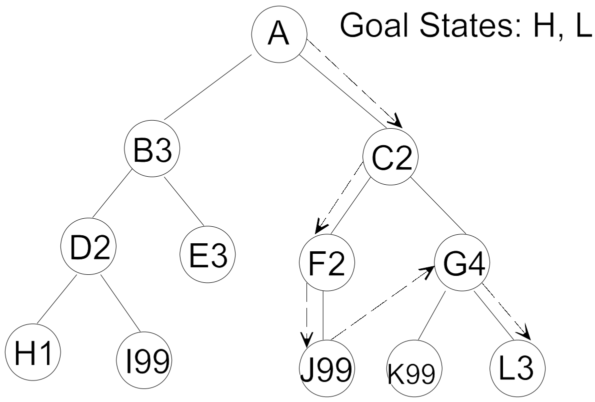


Figure 11.1

The hill-climbing algorithm is described below. The hill-climbing algorithm generates a partial tree/graph.

Hill Climbing Algorithm

```
def hill_climbing(in Start, out State)
 open = [Start], close = [],
 State = failure;
 while (open <> []) AND (State <> success)
   begin
    remove the leftmost state from open, call it X;
    if X is the goal, then
      State = success
    else begin
      generate children of X;
      for each child of X do
      case
        the child is not on open or closed
         begin
          assign the child a heuristic value,
         end;
        the child is already on open
          if the child was reached by a shorter path then
           give the state on open the shorter path
        the child is already on closed:
         if the child is reached by a shorter path then
          begin
           remove the state from closed,
         end;
       endcase
     endfor
     put X on closed;
     re-order the children states by heuristic merit (best leftmost);
     place the reordered list on the leftmost side of open;
   endwhile;
   return State;
end.
```

State Space Representation and Search

Example 1: Suppose that each of the letters represent a state in a state space representation. Legals moves:

A to B3 and C2
B3 to D2 and E3
C2 to F2 and G4
D2 to H1 and I99
F2 to J99
G4 to K99 and L3

Start state: A
Goal state: H, L

The result of applying the hill-climbing algorithm to this problem is illustrated in the talbe below.

	OPEN	**CLOSED**	**X**	**X's Children**	**State**
1	A	-			failure
2			A	B3, C2	failure
3	C2, B3	A			failure
4	B3	A	C2	F2, G4	failure
5	F2, G4, B3	A, C2			failure
6	G4, B3	A, C2	F2	J99	failure
7	J99, G4, B3	A, C2, F2			failure
8	G4, B3	A, C2, F2	J99	none	failure
9	G4, B3	A, C2, F2, J99			failure
10	B3	A, C2, F2, J99	G4	K99, L3	failure
11	L3,K99, B3	A, C2, F2,J99, G4			failure
12	K99, B3	A, C2, F2, J99, G4	L3		success

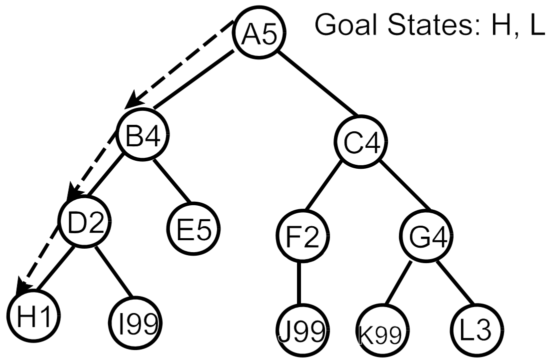
11.1 Greedy Hill Climbing

This is a form of hill climbing which chooses the node with the least estimated cost at each state. It does not guarantee an optimal solution path. Greedy hill climbing is susceptible to the problem of getting stuck at a local optimum.

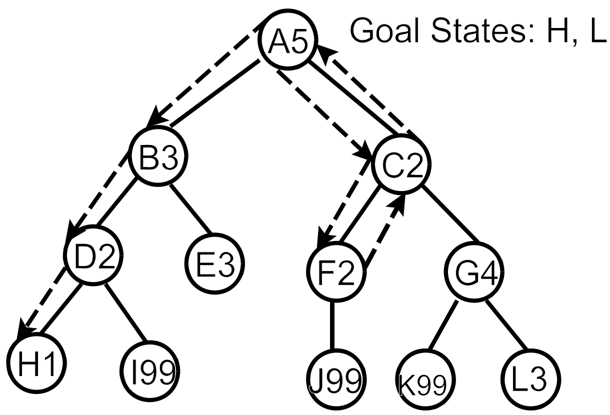
The Greedy Hill Climbing Algorithm

1) Evaluate the initial state.
2) Select a new operator.
3) Evaluate the new state
4) If it is closer to the goal state than the current state make it the current state.
5) If it is no better ignore
6) If the current state is the goal state or no new operators are available, quit. Otherwise repeat steps 2 to 4.

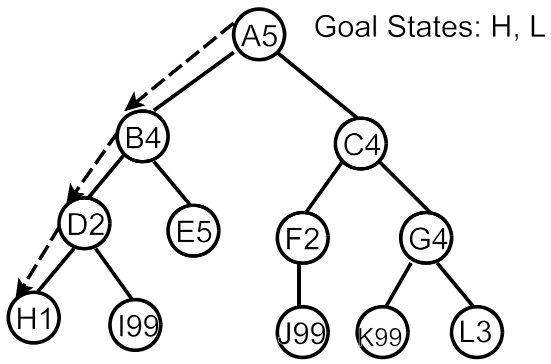
Example 1: Greedy Hill Climbing with Backtracking



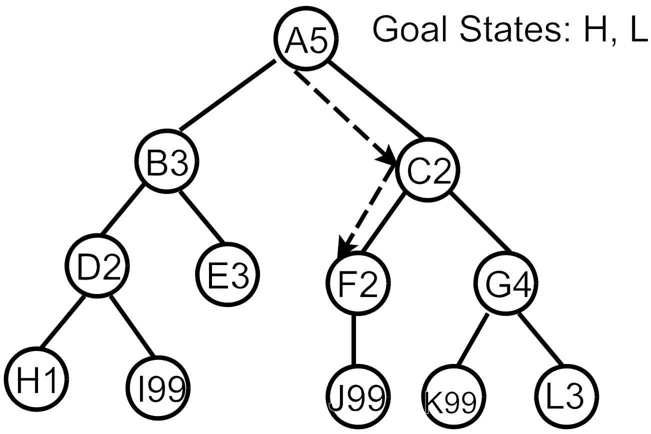
Example 2: Greedy Hill Climbing with Backtracking



Example 1: Greedy Hill Climbing without Backtracking



Example 2: Greedy Hill Climbing without Backtracking



12. The A Algorithm

The A algorithm is essentially the best first search implemented with the following function:

$$f(n) = g(n) + h(n)$$

where g(n) - measures the length of the path from any state n to the start state
h(n) - is the heuristic measure from the state n to the goal state

The use of the evaluation function f(n) applied to the 8-Puzzle problem is illustrated in for following the initial and goal state **Figure 12.1**.

Initial State		
2	8	3
1	6	4
7		5

Goal State		
1	2	3
8		4
7	6	5

The heuristic used for h(n) is the number of tiles out of place. The numbers next to the letters represent the heuristic value associated with that particular state.

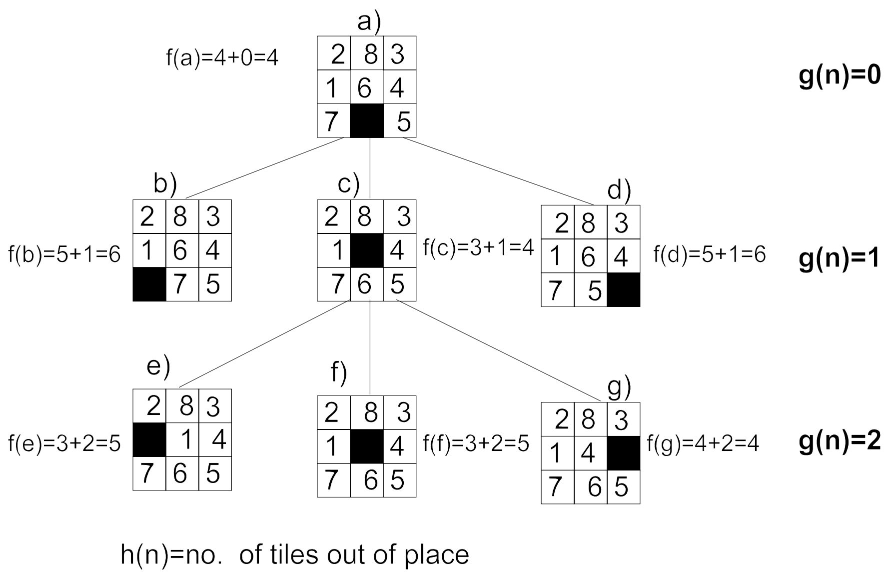


Figure 12.1

13. The A* Algorithm

Search algorithms that are guaranteed to find the shortest path are called admissible algorithms. The breadth first search is an example of an admissible algorithm. The evaluation function we have considered with the best first algorithm is

$$f(n) = g(n) + h(n),$$

where

g(n) - is an estimate of the cost of the distance from the start node to some node n, e.g. the depth at which the state is found in the graph

h(n) - is the heuristic estimate of the distance from n to a goal

An evaluation function used by admissible algorithms is:

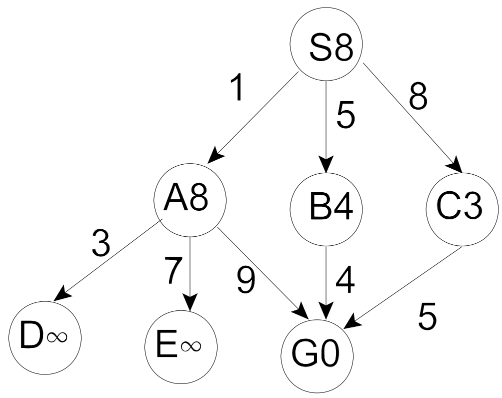
$$f^*(n) = g^*(n) + h^*(n)$$

where
$g^*(n)$ - estimates the shortest path from the start node to the node n.
$h^*(n)$ - estimates the actual cost from the start node to the goal node that passes through n.

f* does not exist in practice and we try to estimate f as closely as possible to f*. g(n) is a reasonable estimate of g*(n). Usually g(n) >= g*(n). It is not usually possible to compute h*(n). Instead we try to find a heuristic h(n) which is bounded from above by the actual cost of the shortest path to the goal, i.e. h(n) <= h*(n). The best first search used with f(n) is called the A algorithm. If h(n) <= h*(n) in the A algorithm then the A algorithm is called the A* algorithm.

The heuristic that we have developed for the 8- puzzle problem are bounded above by the number of moves required to move to the goal position. The number of tiles out of place and the sum of the distance from each correct tile position is less than the number of required moves to move to the goal state. Thus, the best first search applied to the 8-puzzle using these heuristics is in fact an A* algorithm.

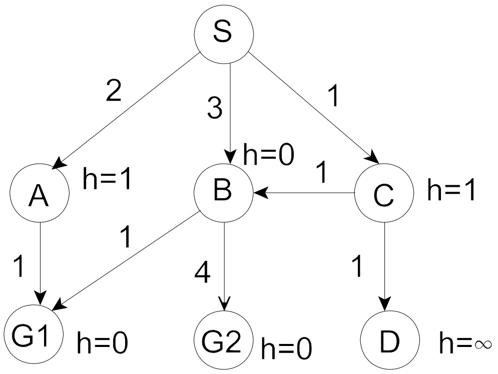
Exercise 1: Consider the following state space. **G** is the goal state. Note that the values of g(n) are given on the arcs:



(a) Is h(n) admissible?
(b) What is the optimal solution path?
(c) What is the cost of this path?

Note: Remember we cannot see the entire state space when we apply the search algorithm.

Exercise 2: Consider the following state space. **G1** and **G2** are goal states:



(a) Is h(n) admissible?
(b) What is the optimal solution path?
(c) What is the cost of this path?

Note: Remember we cannot see the entire state space when we apply the search algorithm.

14. Branch and Bound Techniques

Branch and bound techniques are used to find the most optimal solution. Each solution path has a cost associated with it. Branch and bound techniques keep track of the solution and its cost and continue searching for a better solution further on. The entire search space is usually searched. Each path and hence each arc (rather than the node) has a cost associated with it. The assumption made is that the cost increases with path length. Paths that are estimated to have a higher cost than paths already found are not pursued. These techniques are used in conjunction with depth first and breadth first searches as well as iterative deepening.

Consider the tree in **Figure 14.1**.

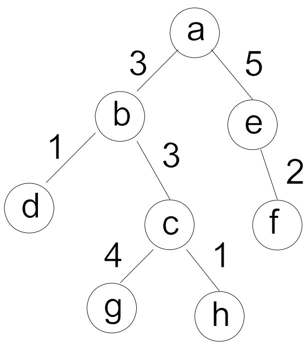


Figure 14.1

In this example d, g, h and f represent the goal state.

A branch and bound technique applied to this tree will firstly find the path from a to d which has a cost of 4. The technique will then compare the cost of each sub-path to 4 and will not pursue the path if the sub-path has a cost exceeding 4. Both the sub-paths a to e and a to c have a cost exceeding 4 these sub-paths are not pursued any further.

15. Game Playing

One of the earliest areas in artificial intelligence is game playing. In this section we examine search algorithms that are able to look ahead in order to decide which move should be made in a two-person zero-sum game, i.e. a game in which there can be at most one winner and two players. Firstly, we will look at games for which the state space is small enough for the entire space to be generated. We will then look at examples for which the entire state space cannot be generated. In the latter case heuristics will be used to guide the search.

15.1 The Game Nim

The game begins with a pile of seven tokens. Two players can play the game at a time, and there is one winner and one loser. Each player has a turn to divide the stack of tokens. A stack can only be divided into two stacks of different size. The game tree for the game is illustrated below.

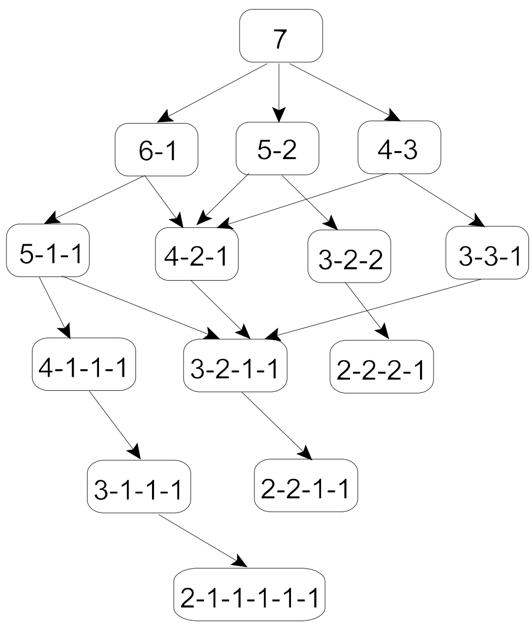


Figure 15.1: NIM state space

If the game tree is small enough we can generate the entire tree and look ahead to decide which move to make. In a game where there are two players and only one can win, we want to maximize the score of one player MAX and minimize the score of the other MIN. The game tree with MAX playing first is depicted in Figure 15.2. The MAX and MIN labels indicate which player's turn it is to divide the stack of tokens. The 0 at a leaf node indicates a loss for MAX and a win for MIN and the 1 represents a win for MAX and a loss for MIN. Figure 15.3 depicts the game space if MIN plays first.

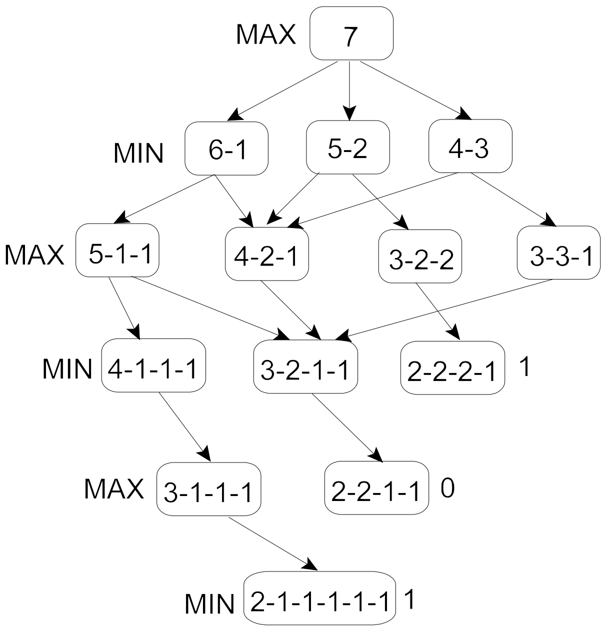


Figure 15.2: MAX plays first

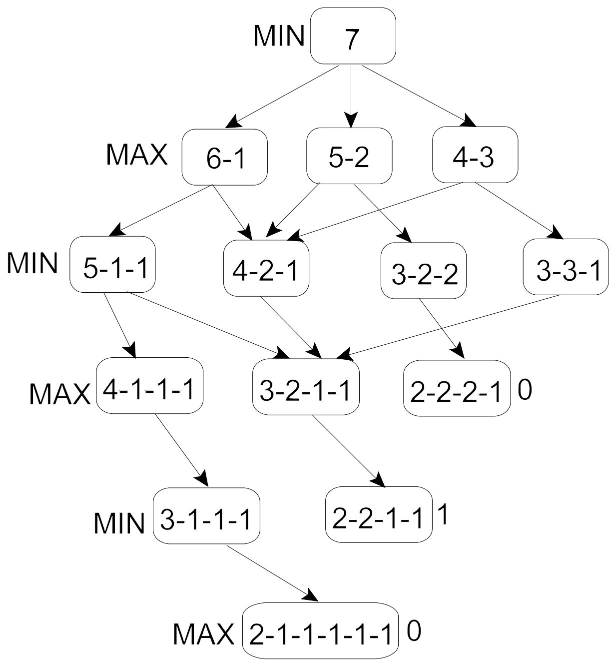


Figure 15.3: MIN plays first

15.3 The Minimax Search Algorithm

The players in the game are referred to as MAX and MIN. MAX represents the person who is trying to win the game and hence maximize his or her score. MIN represents the opponent who is trying to minimize MAX's score. The technique assumes that the opponent has and uses the same knowledge of the state space as MAX. Each level in the search space contains a label MAX or MIN indicating whose move it is at that particular point in the game.

This algorithm is used to look ahead and decide which move to make first. If the game space is small enough the entire space can be generated and leaf nodes can be allocated a win (1) or loss (0) value. These values can then be propagated back up the tree to decide which node to use. In propagating the values back up the tree a MAX node is given the maximum value of all its children and MIN node is given the minimum values of all its children.

***Algorithm 1**: Minimax Algorithm*

Repeat

1. If the limit of search has been reached, compute the static value of the current position relative to the appropriate player. Report the result.
2. Otherwise, if the level is a minimizing level, use the minimax on the children of the current position. Report the minimum value of the results.
3. Otherwise, if the level is a maximizing level, use the minimax on the children of the current position. Report the maximum of the results.

Until the entire tree is traversed

Figure 15.4 illustrates the minimax algorithm applied to the NIM game tree.

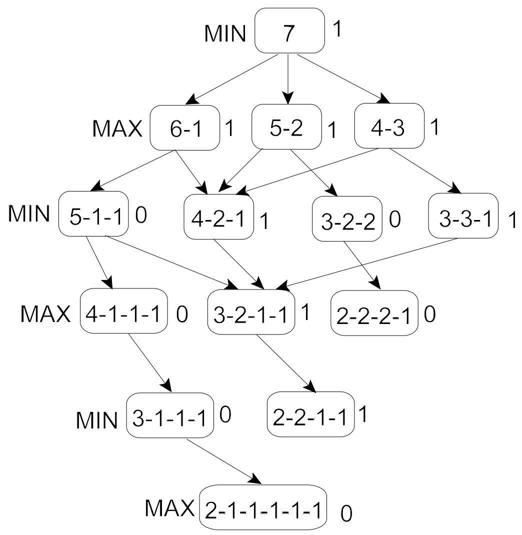


Figure 15.4: Minimax applied to NIM

The value propagated to the top of the tree is a 1. Thus, no matter which move MIN makes, MAX still has an option that can lead to a win.

If the game space is large the game tree can be generated to a particular depth or ply. However, win and loss values cannot be assigned to leaf nodes at the cut-off depth as these are not final nodes. A heuristic value is used instead. The heuristic values are an estimate of how promising the node is in leading to a win for MAX. The depth to which a tree is searched is dependant on the time and computer resource limitations. One of the problems associated with generating a state space to a certain depth is the horizon effect. This refers to the problem of a look-ahead to a certain depth not detecting a promising path and instead leading you to a worse situation. Figure 14.5 illustrates the minimax algorithm applied to a game tree that has been generated to a particular depth. The integer values at the cutoff depth have been calculated using a heuristic function.

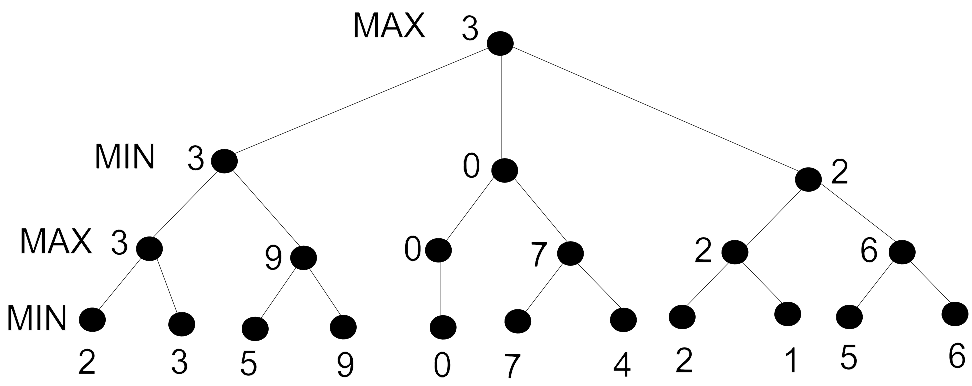
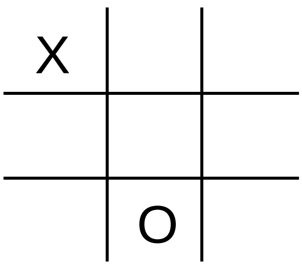


Figure 15.5: Minimax applied to a partially generated game tree

The function used to calculate the heuristic values at the cutoff depth is problem dependant and differs from one game to the next. For example, a heuristic function that can be used for the game tic-tac-toe : h(n) = x(n) - o(n) where x(n) is the total of MAX's possible winning lines (we assume MAX is playing x); o(n) is the total of the opponent's, i.e. MIN's winning lines and h(n) is the total evaluation for a state n. The following examples illustrate the calculation of h(n):

Example 1:



X has 6 possible winning paths
O has 5 possible winning paths
h(n) = 6 - 5 = 1

Example 2:

X		
	O	

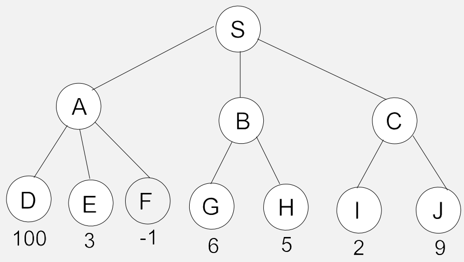
X has 4 possible winning paths
O has 6 possible winning paths
h(n) = 4 - 6 = -2

Example 3:

		O
	X	

X has 5 possible winning paths
O has 4 possible winning paths
h(n) = 5 - 4 = 1

Exercise: Write a program that takes a file as input. The file will contain nodes for a game tree together with the heuristic values for the leaf nodes. For example, for the following tree:



the file should contain:

```
S: ABC
A: DEF
B: GH
C: IJ
D: 100
E: 3
F: -1
G: 6
H: 5
I: 2
J: 9
```

The program must implement the minimax algorithm. The program must output the minimax value and move the that MAX must make. You may assume that MAX makes the first move.

15.4 Alpha-Beta Pruning

The main disadvantage of the minimax algorithm is that all the nodes in the game tree cutoff to a certain depth are examined. Alpha-beta pruning helps reduce the number of nodes explored. Consider the game tree in **Figure 15.6**.

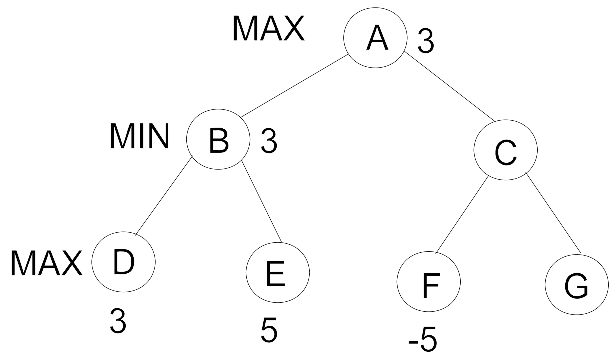


Figure 1.6: Game Tree

The children of B have static values of 3 and 5 and B is at a MIN level, thus the value return to B is 3. If we look at F it has a value of -5. Thus, the value returned to C will be at most -5. However, A is at a MAX level thus the value at A will be at least three which is greater than -5. Thus, no matter what the value of G is, A will have a value of 3. Therefore, we do not need to explore and evaluate it G a s the its value will have not effect on the value returned to A. This is called an alpha cut. Similarly, we can get a beta-cut if the value of a subtree will always be bigger than the value returned by its sibling to a node at a minimizing level.

***Algorithm 2**: Minimax Algorithm with Alpha-Beta Pruning*

Set alpha to -infinity and set beta to infinity
If the node is a leaf node return the value
If the node is a min node then
 For each of the children apply the minimax algorithm with alpha-beta pruning.
  If the value returned by a child is less then beta set beta to this value
  If at any stage beta is less than or equal to alpha do not examine any more children
 Return the value of beta
If the node is a max node
  For each of the children apply the minimax algorithm with alpha-beta pruning.
  If the value returned by a child is greater then alpha set alpha to this value
  If at any stage alpha is greater than or equal to beta do not examine any more children
 Return the value of alpha

Table of Contents