Author: Dr. Neeraj Kumar Sharma

Target Audience: B.Sc. (H) Computer Sc. / B.Tech. Comp. Sc. / M.C.A. students of Artificial Intelligence / Algorithms

Topic: Best First Search & Minimax Principle

Best First Search & Minimax Principle

Best First Search

In the two basic search algorithms we have studied before i.e., depth first search and breadth first search we proceed in a systematic way by discovering/finding/exploring nodes in a predetermined order. In these algorithms at each step during the process of searching there is no assessment of which way to go because the method of moving is fixed at the outset.

The best first search belongs to a branch of search algorithms known as **heuristic search algorithms**. The basic idea of heuristic search is that, rather than trying all possible search paths at each step, we try to find which paths seem to be getting us nearer to our goal state. Of course, we can't be sure that we are really near to our goal state. It could be that we are really near to our goal state. It could be that we have to take some really complicated and circuitous sequence of steps to get there. But we might be able to make a good guess. Heuristics are used to help us make that guess.

To use any heuristic search we need an evaluation function that scores a node in the search tree according to how close to the goal or target node it seems to be. It will just be an estimate but it should still be useful. But the estimate should always be on the lower side to find the optimal or the lowest cost path. For example, to find the optimal path/route between Delhi and Jaipur, an estimate could be straight arial distance between the two cities.

There are a whole batch of heuristic search algorithms eg. Hill Climbing, best first search, A* ,AO* etc. But here we will be focussing on best first search.

Best First Search combines the benefits of both depth first and breadth first search by moving along a single path at a time but change paths whenever some other path looks more promising than the current path.

At each step in the depth first search, we first generate the successors of the current node and then apply a heuristic function to find the most promising child/successor. We then expand/visit (i.e., find it's successors) the chosen successor i.e., find its unknown successors. If one of the successors is a goal node we stop. If not then all these nodes are added to the list of nodes generated or discovered so fart. During this process of generating successors a bit of depth search is performed but ultimately if the solution i.e., goal node is not found then at some point the newly found/discovered/generated node will have a less promising heuristic value than one of the top level nodes which were ignored previously. If this is the case then

we backtrack to the previously ignored but currently the most promising node and we expand /visit that node. But when we back track, we do not forget the older branch from where we have come. Its last node remains in the list of nodes which have been discovered but not yet expanded/ visited . The search can always return to it if at some stage during the search process it again becomes the most promising node to move ahead.

Choosing the most appropriate heuristic function for a particular search problem is not easy and it also incurs some cost. One of the simplest heuristic functions is an estimate of the cost of getting to a solution from a given node this cost could be in terms of the number of expected edges or hops to be traversed to reach the goal node.

We should always remember that in best first search although one path might be selected at a time but others are not thrown so that they can be revisited in future if the selected path becomes less promising.

Although the example we have given below shows the best first search of a tree, it is sometimes important to search a graph instead of a tree so we have to take care that the duplicate paths are not pursued. To perform this job, an algorithm will work by searching a directed graph in which a node represents a point in the problem space. Each node, in addition to describing the problem space and the heuristic value associated with it, will also contain a link or pointer to its best parent and points to its successor node. Once the goal node is found, the parent link will allow us to trace the path from source node to the goal node. The list of successors will allow it to pass the improvement down to its successors if any of them are already existing.

In the algorithm given below, we assume two different list of nodes:

- **OPEN list →** is the list of nodes which have been found but yet not expanded i.e., the nodes which have been discovered /generated but whose children/successors are yet not discovered. Open list can be implemented in the form of a queue in which the nodes will be arranged in the order of decreasing prioity from the front i.e., the node with the most promising heuristic value (i.e., the highest priority node) will be at the first place in the list..
- **CLOSED list →** contains expanded/visited nodes i.e., th anodes whose successors are also genereated. We require to kep the nodes in memory I we want to search a graph rather than a tree, since whenever a new node is generated we need to check if it has been generated before.

The algorithm can be written as:

Best First Search

1. Place the start node on the OPEN list.
2. Create a list called CLOSED i.e., initially empty.
3. If the OPEN list is empty search ends unsuccessfully.
4. Remove the first node on OPEN list and put this node on CLOSED list.
5. If this is a goal node, search ends successfully.
6. Generate successors of this node:

For each successor :
(a). If it has bot been discovered / generated before i.e.e, it is not on OPEN, evaluate this node by applying the heuristic function, add it to the OPEN and record its parent.
(b). If it has been discovered / generated before, change the parent if the new path is better than the previous one. In that case update the cost of getting to this node and to any successors that this node may already have.

7. Reorder the list OPEN, according to the heuristic merit.
8. Go to step 3.

Example

In this example, each node has a heuristic value showing the estimated cost of getting to a solution from this node. The example shows part of the search process using best first search.

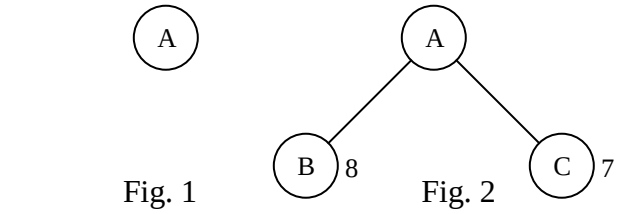


Fig. 1

Fig. 2

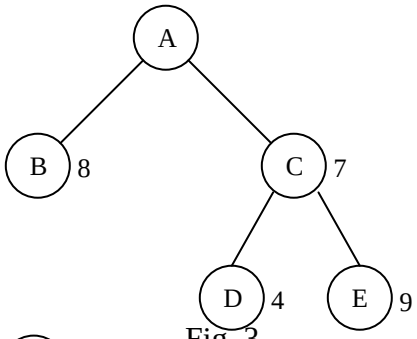


Fig. 3

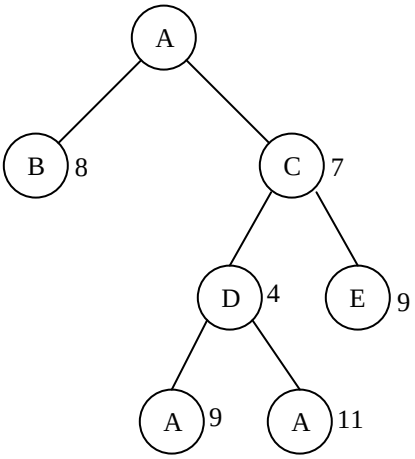


Fig. 4

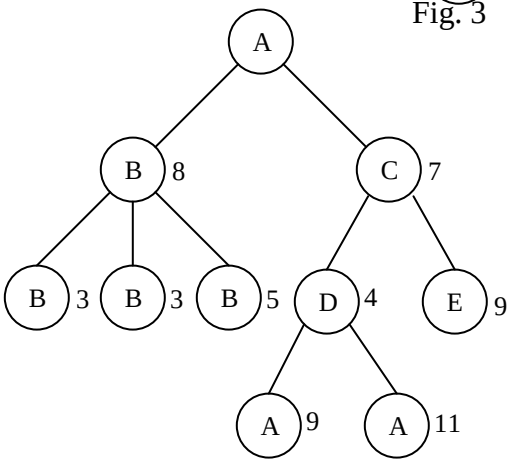


Fig. 5

Fig. 1 : A is the starting node

Fig. 2 : Generate its successors B and C

Fig. 3 : As the estimated goal distance of C is less so expand C to find its successors d and e.

Fig. 4 : Now D has lesser estimated goal distance i.e., 4 , so expand D to generate F and G with distance 9 and 11 respectively.

Fig. 5 : Now among all the nodes which have been discovered but yet not expanded B has the smallest estimated goal distance i.e., 8, so now backtrack and expand B and so on.

Best first searches will always find good paths to a goal node if there is any. But it requires a good heuristic function for better estimation of the distance to a goal node.

The Minimax Principle

Whichever search technique we may use, it can be seen that many graph problems including game problems, complete searching of the associated graph is not possible. The alternative is to perform a partial search or what we call a limited horizon search from the current position. This is the principle behind the minimax procedure.

Minimax is a method in decision theory for minimizing the expected maximum loss. It is applied in two players games such as tic-tac-toe, or chess where the two players take alternate moves. It has also been extended to more complex games which require general decision making in the presence of increased uncertainty. All these games have a common property that they are logic games. This means that these games can be described by a set of rules and premises. So it is possible to know at a given point of time, what are the next available moves. We can also call them full information games as each player has complete knowledge about possible moves of the adversary.

In the subsequent discussion of games, the two players are named as MAX and MIN. We are using an assumption that MAX moves first and after that the two players will move alternatively. The extent of search before each move will depend o the play depth – the amount of lookahead measured in terms of pairs of alternating moves for MAX and MIN.

As I have already specified, complete search of most game graphs is computationally infeasible. It can be seen that for a game like chess it might take centuries to generate the complete search graph even in an environment where a successor could be generated in a few nanoseconds. Therefore for many complex games, we must accept the fact that search to termination is impossible instead we must use partial searching techniques.

For searching we can use either breadth first, depth first or heuristic methods except that the termination conditions must now be specified. Several artificial termination conditions can be specified based on factors such as time limit, storage space and the depth of the deepest node in the search tree.

In a two player game, the first step is to define a **static evaluation function efun()**, which attaches a value to each position or state of the game. This value indicates how good it would be for a player to reach that position. So after the search terminates, we must extract from the search tree an estimate of the best first move by applying a static evaluation function efun() to the leaf nodes of the search tree. The evaluation function

measures the worth of the leaf node position. For example, in chess a simple static evaluation function might attach one point for each pawn, four points for each rook and eight points for queen and so on. But this static evaluation is too easy to be of any real use. Sometimes we might have to sacrifice queen to prevent the opponent from a winning move and to gain advantage in future so the key lies in the amount of lookahead. The more number of moves we are able to lookahead before evaluating a move, the better will be the choice.

In analyzing game trees, we follow a convention that the value of the evaluation function will increase as the position becomes favourable to player MAX , so **the positive values will indicate position that favours MAX** whereas for the **positions favourable to player MIN are represented by the static evaluation function having negative values** and values near zero correspond to game positions not favourable to either MAX or MIN. In a terminal position, the static evaluation function returns either positive infinity or negative infinity where as positive infinity represents a win for player MAX and negative infinity represents a win for the player MIN and a value zero represents a draw.

In the algorithm, we give ahead, the search tree is generated starting with the current game position upto the end game position or lookahead limit is reached. Increasing the lookahead limit increases search time but results in better choice. The final game position is evaluated from the MAX's point of view. The nodes that belong to the player MAX receive the maximum value of its children. The nodes for the player MIN will select the minimum value of its children.

In the algorithm, lookahead limit represents the lookahead factor in terms of number of steps, u and v represent game states or nodes, maxmove() and minmove() are functions to describe the steps taken by player MAX or player MIN to choose a move, efun() is the static evaluation function which attaches a positive or negative integer value to a node (i.e., a game state), value is a simple variable.

Now to move number of steps equal to the lookahead limit from a given game state u, MAX should move to the game state v given by the following code :

```
maxval = - ∞
for each game state w that is a successor of u
    val = minmove(w,lookaheadlimit)
    if (val >= maxval)
        maxval = val
        v = w              // move to the state v
```

The minmove() function is as follows :

```
minmove(w, lookaheadlimit)
{
  if(lookaheadlimit = = 0 or w has no successor)
      return efun(w)
  else
```

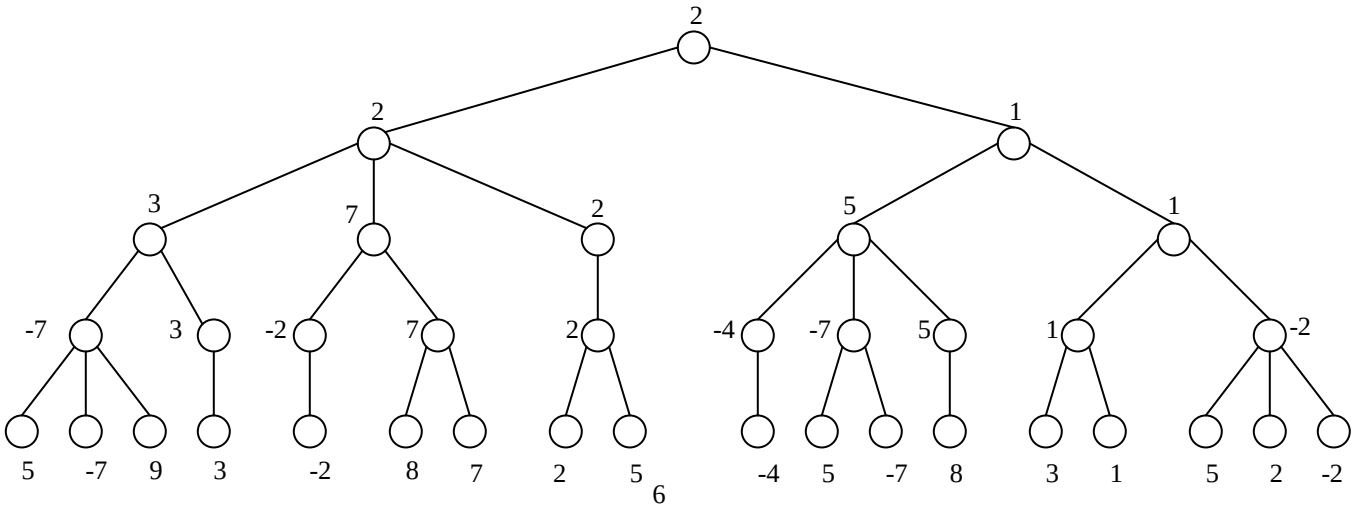
```
        minval = + ∞
        for each successor x of w
                val = maxmove(x,lookaheadlimit – 1)
                if (minval > val)
                        minval = val
        return(minval)
}
```

The maxmove() function is as follows :

```
maxmove(w, lookaheadlimit)
{
    if (lookaheadlimit = =  0 or w has no successor)
            return efun(w)
    else
            maxval = - ∞
            for each successor x of w
                    val = minmove(x,lookaheadlimit – 1)
                    if (maxval < val)
                            maxval = val
            return(maxval)
}
```

We can see that in the minimax technique, player MIN tries to minimize te advantage he allows to player MAX, and on the other hand player MAX tries to maximize the advantage he obtains after each move.

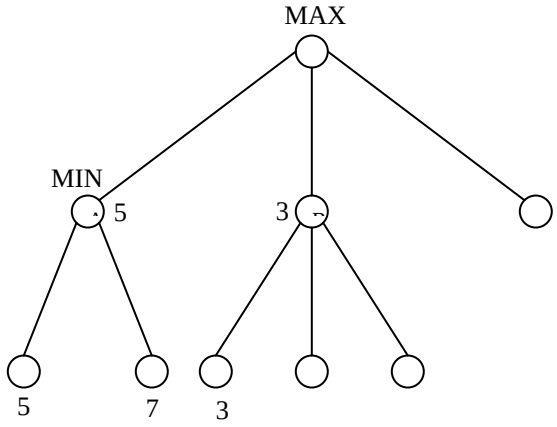
Let us suppose the graph given below shows part of the game. The values of leaf nodes are given using efun() procedure for a particular game then the value of nodes above can be calculated using minimax principle.Suppose the lookahead limit is 4 and it is MAX's turn.



Speeding up the minimax algorithm using Alpha-Beta Pruning

We can take few steps to reduce the search time in minimax procedure. In the figure given below, the value for node A is 5 and the first found value for the subtree starting at node B is 3. So since the B node is at the MIN player level, we know that the selected value for the B node must be less than or equal to 3. But we also know that the A node has the value 5 and both A and B nodes share the same parent at the MAX level immediately above. This means that the game path starting at the B node can never be selected because 5 is better than 3 for the MAX node. So it is not worth spending time to search for children of the B node and so we can safely ignore all the remaining children of B.

This shows that **the search on same paths can sometimes be aborted** (i.e., it is not required to explore all paths) because we find out that the search subtree will not take us to any viable answer.



This optimization is known as alpha beta pruning/procedure and the values, below which search need not be carried out are known as alpha beta cutoffs.

A general algorithm for alpha beta procedure is as follows :

1. Have two values passed around the tree nodes :
   The alpha value - which holds best MAX value found at the MAX level
   The beta value - which holds best Min value found at the MIN level

2. At MAX player level, before evaluating each child path, compare the returned value of the previous path with the beta value. If the returned value is greater then abort the search for the current node.
3. At Min player level,before evaluating eachh child path, compare the returned value of the previouss path with the alpha value. If the value is lesser then abort the searh for the current node.

We should note that :

- The alpha values of MAX nodes (including the start value) can never decrease.
- The beta value of MIN nodes can never increase.

So we can see that remarkable reductions in the amount of search needed to evaluate a good move are possible by using alpha beta pruning / procedure.