[Chapter 3: Searching]

Artificial Intelligence (CSC 355)

Arjun Singh Saud

Central Department of Computer Science & Information Technology

Tribhuvan University

Problem Solving by Searching

Four general steps in problem solving:

- Goal formulation
  - What are the successful world states
- Problem formulation
  - What actions and states to consider given the goal
- Search
  - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
- Execute
  - Give the solution perform the actions.

Problem formulation

A problem is defined by:

- An initial state: State from which agent start
- Successor function: Description of possible actions available to the agent.
- Goal test: Determine whether the given state is goal state or not
- Path cost: Sum of cost of each path from initial state to the given state.

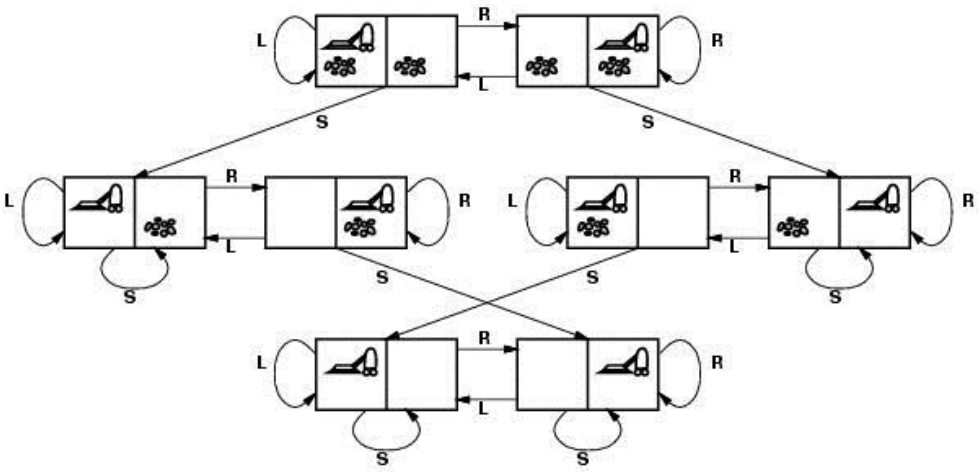
A solution is a sequence of actions from initial to goal state. Optimal solution has the lowest path cost.

State Space representation

The state space is commonly defined as a directed graph in which each node is a state and each arc represents the application of an operator transforming a state to a successor state.

A **solution** is a path from the initial state to a goal state.

State Space representation of Vacuum World Problem:



States?? two locations with or without dirt: 2 x 2^2=8 states.
Initial state?? Any state can be initial
Actions?? {*Left, Right, Suck*}
Goal test?? Check whether squares are clean.
Path cost?? Number of actions to reach goal.

A search problem

Figure below contains a representation of a map. The nodes represent cities, and the links represent direct road connections between cities. The number associated to a link represents the length of the corresponding road.

The search problem is to find a path from a city S to a city G

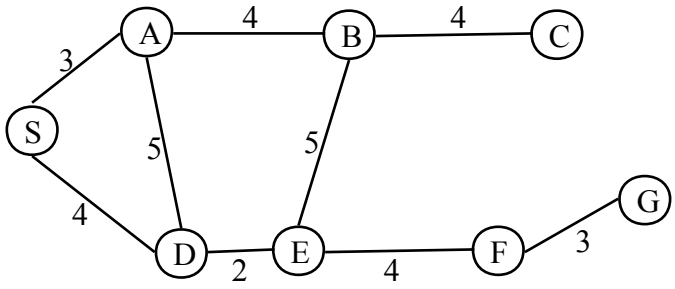


Figure : A graph representation of a map

This problem will be used to illustrate some search methods.

There are two broad classes of search methods:
- uninformed (or blind) search methods;
- heuristically informed search methods.

In the case of the uninformed search methods the order in which potential solution paths are considered is arbitrary, using no domain-specific information to judge where the solution is likely to lie.

In the case of the heuristically informed search methods one uses domain-dependent (heuristic) information in order to search the space more efficiently.

Measuring problem Solving Performance

We will evaluate the performance of a search algorithm in four ways

- **Completeness**
  - An algorithm is said to be complete if it definitely finds solution to the problem, if exist.
- **Time Complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**
- **Space Complexity**
  - How much space is used by the algorithm? Usually measured in terms of the **maximum number of nodes in memory at a time**
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

Uninformed search

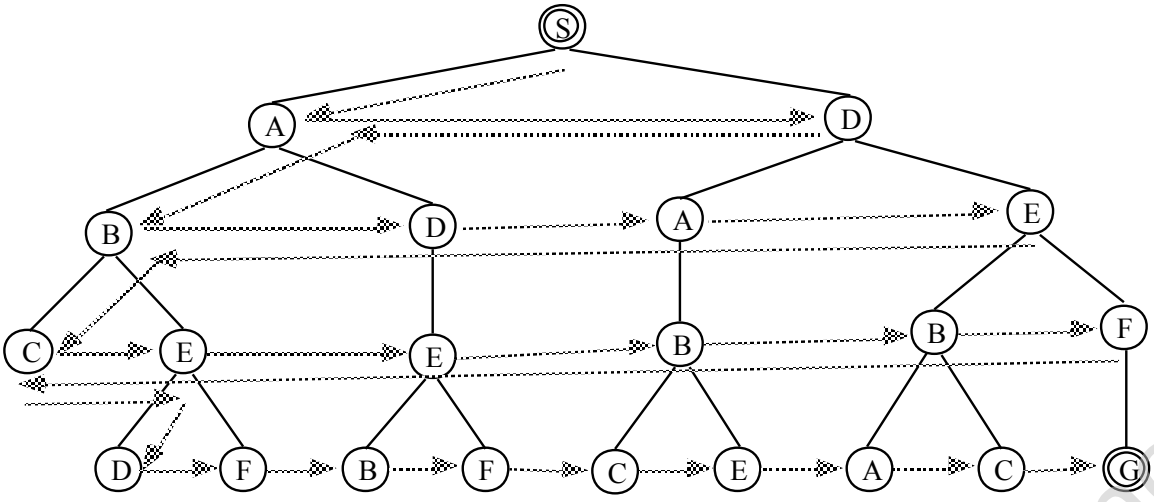
Breadth First Search(BFS)

Looks for the goal node among all the nodes at a given level before using the children of those nodes to

Expand *shallowest* unexpanded node

fringe is implemented as a FIFO queue

Note: Fringe is a generated but unexpanded node.



BFS evaluation

Completeness:

- *Does it always find a solution if one exists?*
- YES
  - If shallowest goal node is at some finite depth d and If *b* is finite

Time complexity:

- Assume a state space where every state has *b* successors.
  - root has *b* successors, each node at the next level has again b successors (total b^2), …
  - Assume solution is at depth *d*
  - Worst case; expand all except the last node at depth *d*
  - Total no. of nodes generated:

$$b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots\ldots b^d + (b^{d+1} - b) = \mathbf{O(b^{d+1})}$$

Space complexity:

- Each node that is generated must remain in memory
- Total no. of nodes in memory:

$$1 + b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots\ldots b^d + (b^{d+1} - b) = \mathbf{O(b^{d+1})}$$

Optimal (i.e., admissible):

- if all paths have the same cost. Otherwise, not optimal but finds solution with shortest path length (shallowest solution). If each path does not have same path cost shallowest solution may not be optimal

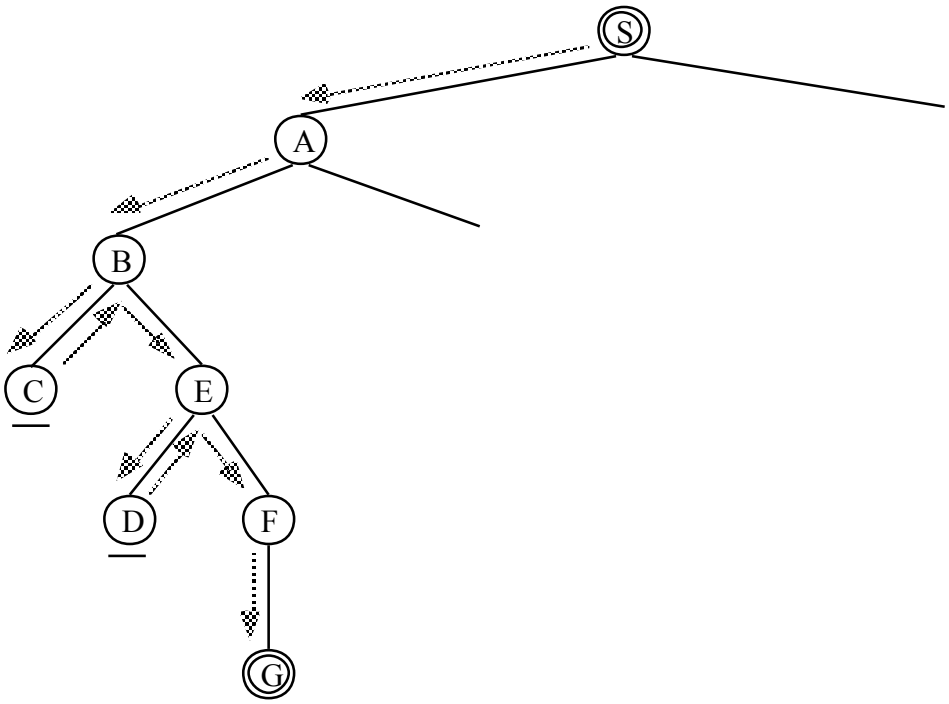
Two lessons:

- Memory requirements are a bigger problem than its execution time.
- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH2	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	107	19 minutes	10 gigabytes
8	109	31 hours	1 terabyte
10	1011	129 days	101 terabytes
12	1013	35 years	10 petabytes
14	1015	3523 years	1 exabyte

Depth First Search(DFS)

Looks for the goal node among all the children of the current node before using the sibling of this node
Expand *deepest* unexpanded node
Fringe is implemented as a LIFO queue (=stack)



DFS evaluation

Completeness;

- *Does it always find a solution if one exists?*
- NO
  - If search space is infinite and search space contains loops then DFS may not find solution.

Time complexity;

- Let m is the maximum depth of the search tree. In the worst case Solution may exist at depth m.
- root has *b* successors, each node at the next level has again b successors (total b^2), …
- Worst case; expand all except the last node at depth *m*
- Total no. of nodes generated:

$$b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots b^m = \mathbf{O(b^m)}$$

Space complexity:

- It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
- Total no. of nodes in memory:

  1+ b + b + b + …….. b m times = **O(bm)**

Optimal (i.e., admissible):

- DFS expand deepest node first, if expands entire let sub-tree even if right sub-tree contains goal nodes at levels 2 or 3. Thus we can say DFS may not always give optimal solution.

Depth-limited search

It is DF-search with depth limit *l*.

- i.e. nodes at depth *l is assumed to* have no successors.
- Problem knowledge can be used to guess the value of *l*

Solves the infinite-path problem of DFS. Yet it introduces another source of problem if we are unable to find good guess of *l*. Let d is the depth of shallowest solution.

If $l < d$ then incompleteness results.

If $l > d$ then not optimal.

Time complexity: $O(b^l)$

Space complexity: $O(bl)$

Challenge!!

Derive Time complexity and Space Complexity

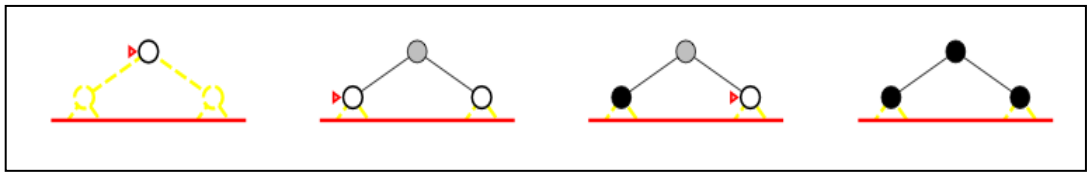
Iterative deepening search

- A general strategy to find best depth limit *l*.
  - Goals is found at depth *d*, the depth of the shallowest goal-node.

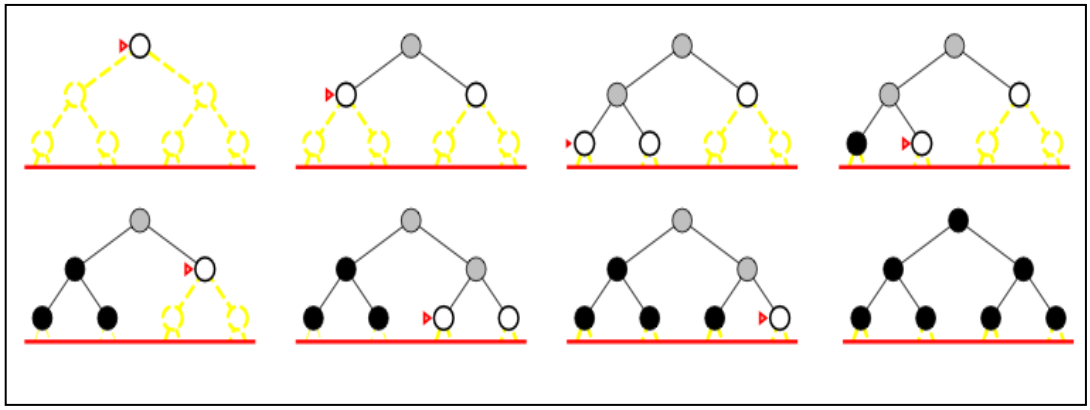
- Often used in combination with DF-search
- Combines benefits of DFS and BFS
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on until the goal is found. This will occur when depth d is reached: that is depth of shallowest goal node.

Limit = 0

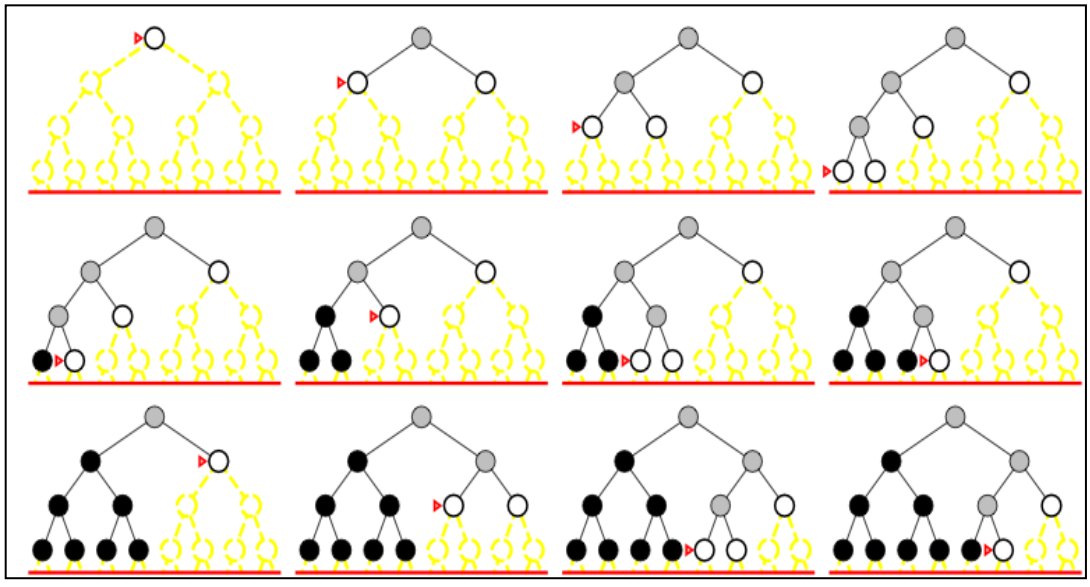
Limit = 1



Limit = 2



Limit = 3



ID search evaluation

Completeness:

- YES (no infinite paths)

Time complexity:

- Algorithm seems costly due to repeated generation of certain states.
- Node generation:

  level d: once
  level d-1: 2
  level d-2: 3
  …
  level 2: d-1
  level 1: d

- Total no. of nodes generated:

  $d.b + (d-1).b^2 + (d-2).b^3 + \ldots\ldots\ldots\ldots\ldots\ldots + 1.b^d = \mathbf{O(b^d)}$

Space complexity:

- It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
- Total no. of nodes in memory:

  $1 + b + b + b + \ldots\ldots\ldots\ldots\ldots\ldots$ b d times = **O(bd)**

Optimality:

- YES if path cost is non-decreasing function of the depth of the node.

Note::-
Notice that BFS generates some nodes at depth d+1, whereas IDS does not.The result is that IDS is actually faster than BFS , despite the repeated generation of node.
Example:
Num. of nodes generated for b=10 and d=5 solution at far right

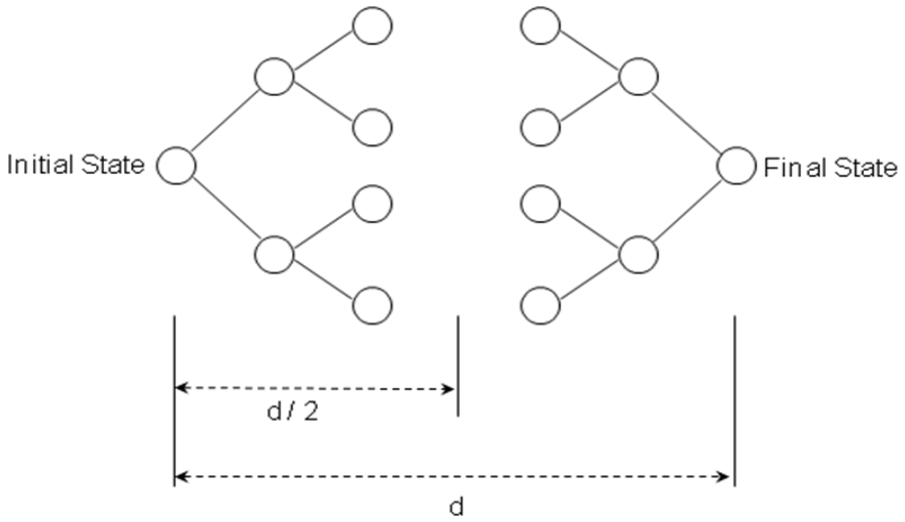
N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450

N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100

Recommended reading!!!

Practice the problems for calculating time and sapace requirement of search algorithm that are done in class and also other problems from book.

Bidirectional search



Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches. one forward from the initial state and one backward from the goal, stopping when the two meet in the middle. The reason for this approach is that in many cases it is faster: for instance, in a simplified model of search problem complexity in which both searches expand a tree with branching factor b, and the distance from start to goal is d, each of the two searches has complexity $O(b^{d/2})$ and the sum of these two search times is much less than the $O(b^{d})$, complexity that would result from a single search from the beginning to the goal. Even if theoretically bidirectional search seems effective than unidirectional search practically the case may be different because finding out where two search algorithms (forward and backward search) intersects needs additional data structure to be maintained and addition implementation steps to identify whether two search algorithms intersects or not.

Drawbacks of uniformed search :

- Criterion to choose next node to expand depends only on a global criterion: level.
- Does not exploit the structure of the problem.
- One may prefer to use a more flexible rule, that takes advantage of what is being discovered on the way, and hunches about what can be a good move.
- Very often, we can select which rule to apply by comparing the current state and the desired state

Heuristic Search:

Heuristic Search Uses domain-dependent (heuristic) information in order to search the space more efficiently.

Ways of using heuristic information:

- Deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order;
- In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time;
- Deciding that certain nodes should be discarded, or *pruned*, from the search space.

Best-First Search

Idea: use an *evaluation function f(n)* that gives an indication of which node to expand next for each node.

- usually gives an estimate to the goal.
- the node with the lowest value is expanded first.

A key component of *f(n*) is a heuristic function, *h(n),*which is a additional knowledge of the problem.

There is a whole family of best-first search strategies, each with a different evaluation function.

Typically, strategies use estimates of the cost of reaching the goal and try to minimize it.

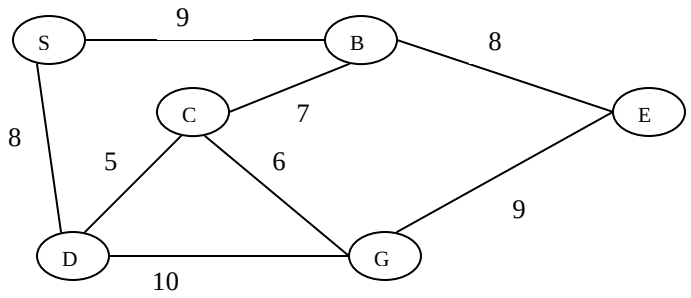
Special cases: based on the evaluation function.

- Greedy best-first search
- A*search

Greedy Best First Search

Greedy best first search expands the node that seems to be closest to the goal node. Heuristic function is used to estimate which node is closest to the goal node. It doest not always gives optimal solution. Optimality of the solution depends upon how good our heuristic is. Once the greedy best first search takes a node, it never looks backward.
Heuristic used by Greedy Best First Search is:
$f(n) = h(n)$
Where h(n) is the estimated cost of the path from node n to the goal node.

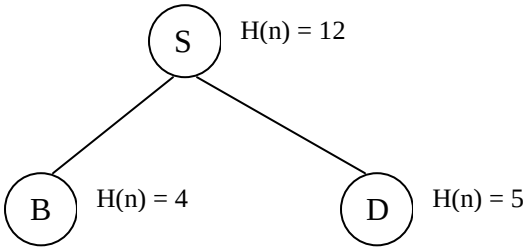
For example consider the following graph



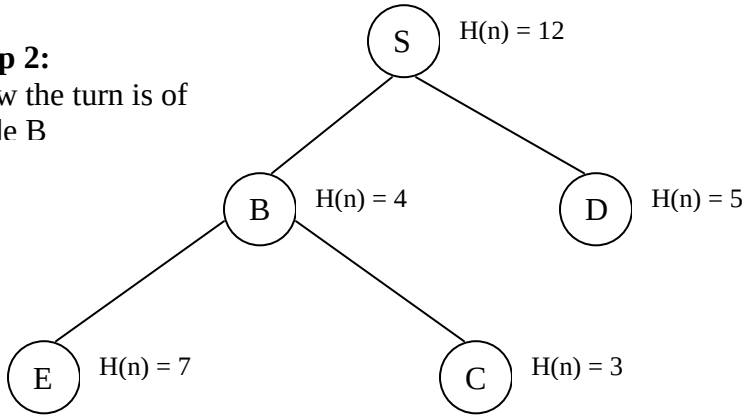
Straight Line distances to node G (goal node) from other nodes is given below:
S→ G = 12
B→ G = 4
E→ G = 7
D→ G = 5
C→ G = 3

Let H(n)= Straight Line distance
Now Greedy Search operation is done as below:

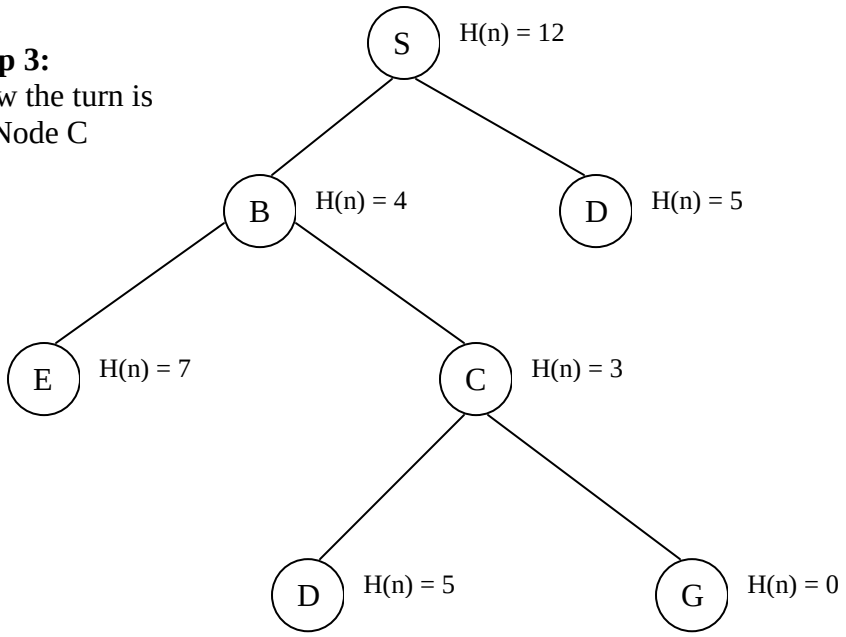
Step 1



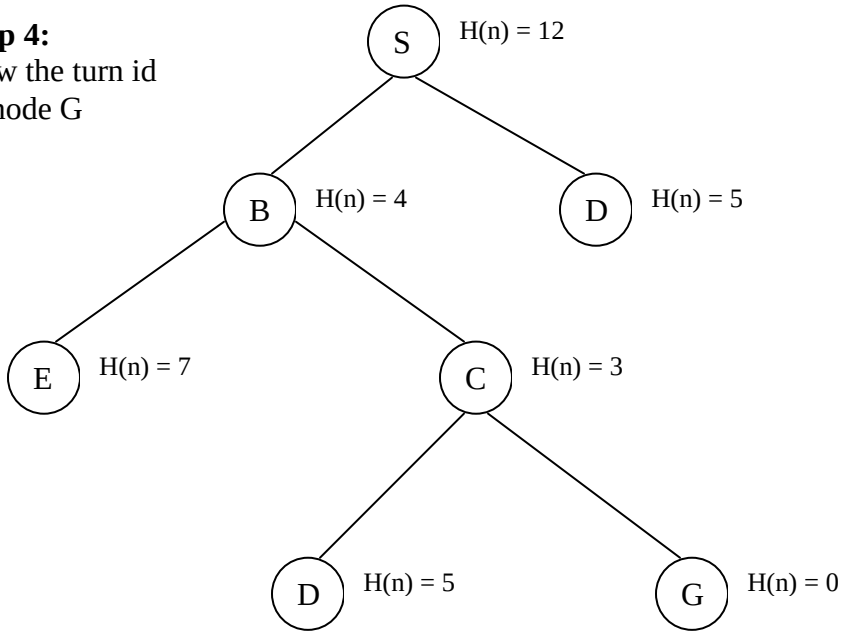
Step 2:
Now the turn is of node B



Step 3:
Now the turn is of Node C



Step 4:
Now the turn id of node G



Evaluation Greedy Search

Completeness:

- While minimizing the value of heuristic greedy may oscilate between two nodes. Thus it is not complete.

Optimality:

- Greedy search is not optimal. Same as DFS.

Time complexity:

– In worst case Greedy search is same as DFS therefore it's time complexity is $O(b^m)$.

Space Complexity:

– Space complexity of DFS is $O(b^m)$. No nodes can be deleted from memory.

A * Search

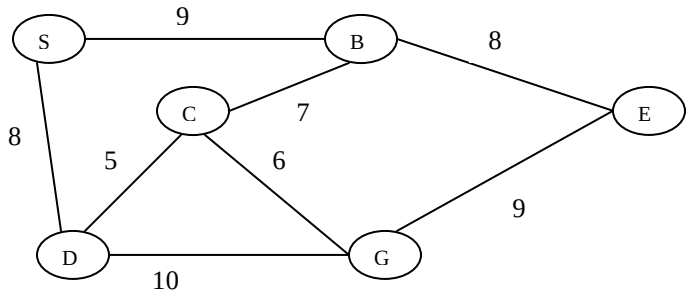
Like all informed search algorithms, it first searches the routes that appear to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account (the g(n) part of the heuristic is the cost from the start, and not simply the local cost from the previously expanded node).The algorithm traverses various paths from start to goal. Evaluation function used by A* search algorithm is

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

Where:

- g(n): the actual shortest distance traveled from initial node to current node
- h(n): the estimated (or "heuristic") distance from current node to goal
- f(n): the sum of g(n) and h(n)

For example consider the following graph



Straight Line distances to node G (goal node) from other nodes is given below:

S→ G = 12

B→ G = 4

E→ G = 7

D→ G = 6

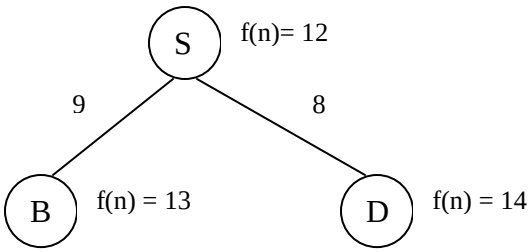
C→ G = 3

Labels in the graph shows actual distance.

Let H(n)= Straight Line distance

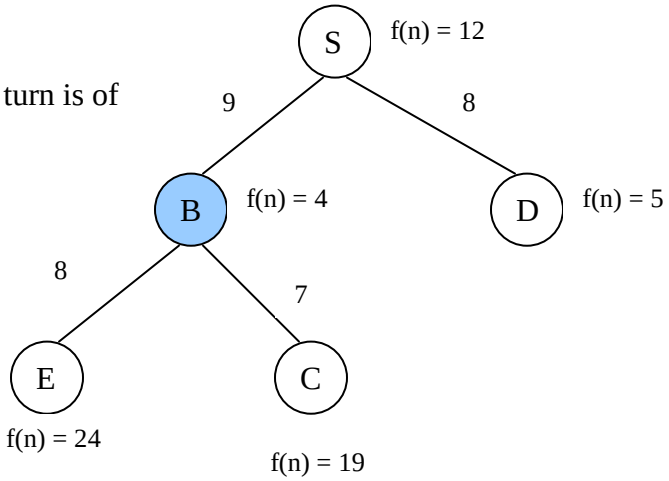
Now A* Search operation is done as below

Step 1



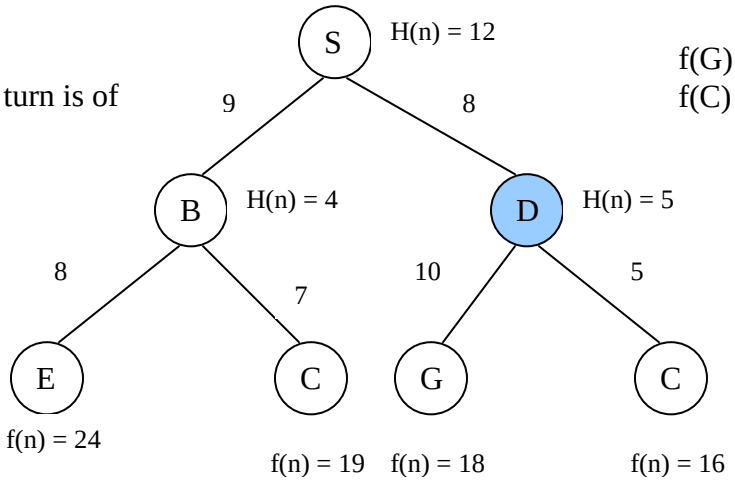
f(B) = 9 + 4 =13
f(D) = 8 + 6 =14

Step 2:
Now the turn is of node B



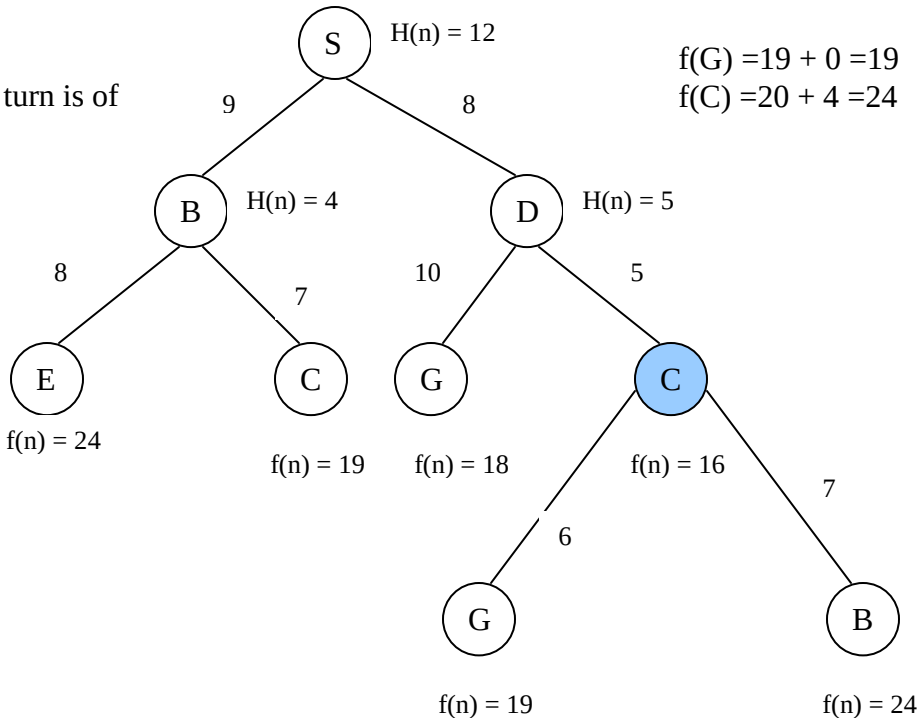
f(E) =17 + 7 =24
f(C) =16 + 3 =19

Step 3:
Now the turn is of node D



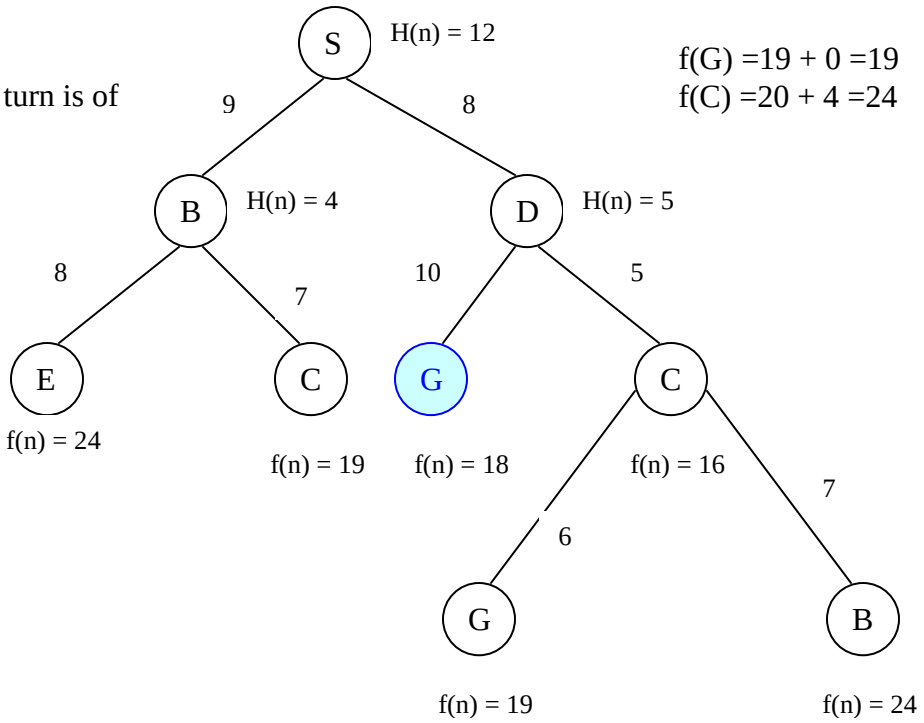
f(G) =18 + 0 =18
f(C) =13 + 3 =16

Step 4:
Now the turn is of node C



f(G) =19 + 0 =19
f(C) =20 + 4 =24

Step 4:
Now the turn is of node G



f(G) =19 + 0 =19
f(C) =20 + 4 =24

G is goal node, Hence we are done

Evaluating A* Search:

Completeness:

- Yes A* search always gives us solution

Optimality:

- A* search gives optimal solution when the heuristic function is admissible heuristic.

Time complexity:

- Exponential with path length i.e. $O(b^d)$ where d is length of the goal node from start node.

Space complexity:

- It keeps all generated nodes in memory. Hence space is the major problem not time

Admissible heuristic

A heuristic function is said to be admissible heuristic if iot never overestimates the cost to reach to the goal.
i.e
$h(n) \leq h^*(n)$
Where
h(n) = Estimated cost to reach to the goal node from node n
h*(n) = Actual cost to reach to the goal node from node n

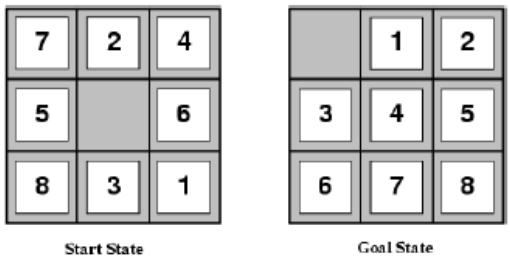
Formulating admissible heuristics:

- *n* is a node
- *h* is a heuristic
- *h*(*n*) is cost indicated by *h* to reach a goal from *n*
- *C*(*n*) is the actual cost to reach a goal from n
- *h* is admissible if

$$\forall n, h(n) \leq C(n)$$

For Example: 8-puzzle

Figure shows 8-puzzle start state and goal state. The solution is 26 steps long.



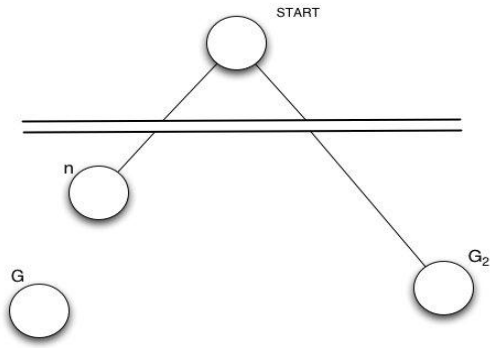
$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = sum of the distance of the tiles from their goal position (notdiagonal).
$h_1(S)$ = ? 8
$h_2(S)$ = ? 3+1+2+2+2+3+3+2 = 18
$h_n(S)$ = max{h1(S), h2(S)}= 18

Consistency (Monotonicity)

A heuristic is said to be consistent if for any node N and any successor N' of N , estimated cost to reach to the goal from node N to goal node is less than the sum of step cost from N to N' and estimated cost from node N' to goal node.
i.e
$h(n) \leq d(n, n') + h(n')$
Where
h(n) = Estimated cost to reach to the goal node from node n
d(n, n') = actual cost from n to n'

#Prove that A* search gives optimal solution when the heuristic is admissible.

Suppose suboptimal goal *G2* in the queue.
Let *n* be an unexpanded node on a shortest to optimal goal *G* and C* be the cost of optimal goal node.



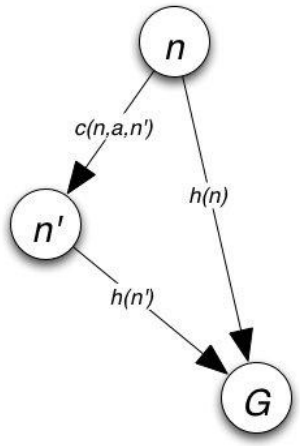
f(G2) = h (G2) + g(G2)
f(G2) = g(G2) since h(G2)=0
f(G2) > C* ……..(1)

Again
Since h(n) is admissibl, It doest not overstimatews the cost of completing the solution path.
=> f(n) = g(n) + h(n) ≤C* ……(2)

Now from (1) and (2)
f(n) ≤ C* < f(G2)

Since f(G2) > f(n), A* will never select G2 for expansion. Thus A* gives us optimal solution when heuristic function is admissible.

 # Prove that:If h(n) is consistent , then the values of f(n) along the path are nondecreasing.



Suppose n' is successor of n, then
g(n') = g(n) + C(n,a,n')
we know that,
f(n') = g(n') + h(n')
f(n') = g(n) + C(n,a,n') + h(n') …….(1)
A heuristic is consistent if
h(n) ≤ C(n,a,n') + h(n')……(2)
Now from (1) and (2)
f(n') = g(n) + C(n,a,n') + h(n') ≥ g(n) + h(n) = f(n)
⇨ f(n') ≥ f(n)
⇨ f(n) is nondecreasing along any path.

Hill climbing

Hill climbing is depth-first search with a heuristic measurement that orders choices as nodes are expanded. It always selects the most promising successor of the node last expanded.

For instance, consider that the most promising successor of a node is the one that has the shortest straight-line distance to the goal node G. In figure below, the straight line distances between each city and G is indicated in square brackets.

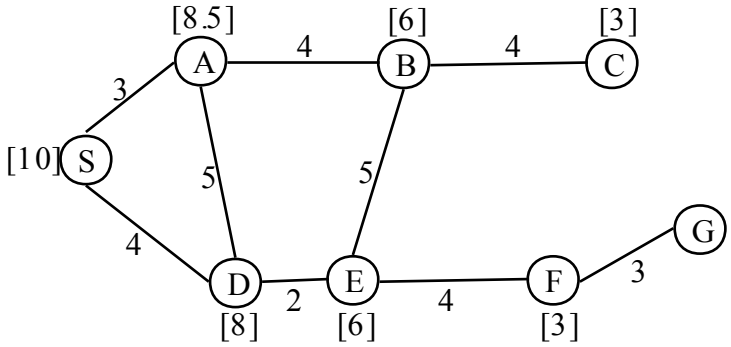


Figure: The map indicating also the straight line distances between each city and G.

The hill climbing search from S to G proceeds as follows:

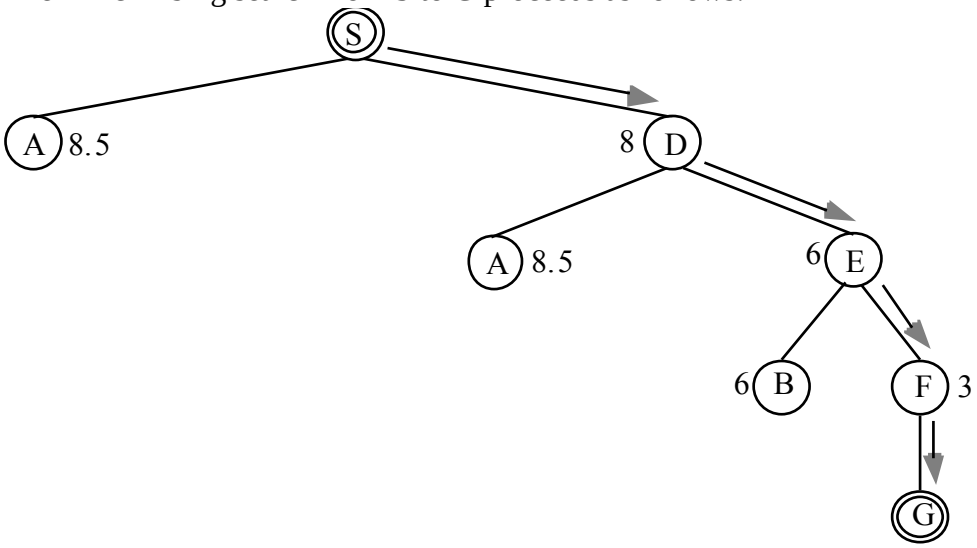
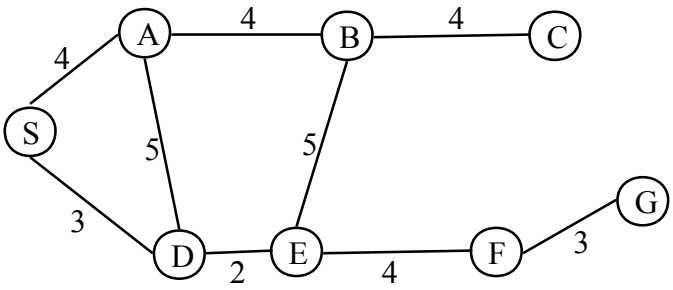


Figure : Hill climbing search of a path from node S to node G.

Exercise

Apply the hill climbing algorithm to find a path from S to G, considering that the most promising successor of a node is its closest neighbor.



Note:
The difference between the hill climbing search method and the best first search method is the following one:
- the best first search method selects for expansion the most promising leaf node of the current search tree;
- the hill climbing search method selects for expansion the most promising successor of the node last expanded.

Problems with Hill Climbing

:

- Gets stuck at **local minima** when we reach a position where there are no better neighbors, it is not a guarantee that we have found the best solution. **Ridge** is a sequence of local maxima.
- Another type of problem we may find with hill climbing searches is finding a ***plateau.*** This is an area where the search space is flat so that all neighbors return the same evaluation

Simulated Annealing

It is motivated by the physical annealing process in which material is heated and slowly cooled into a uniform structure. Compared to hill climbing the main difference is that SA allows downwards steps. Simulated annealing also differs from hill climbing in that a move is selected at random and then decides whether to accept it. If the move is better than its current position then simulated annealing will always take it. If the move is worse (i.e. lesser quality) then it will be accepted based on some probability. The probability of accepting a worse state is given by the equation

$$P = \exp(-c/t) > r$$

Where

c	=	the change in the evaluation function
t	=	the current value
r	=	a random number between 0 and 1

The probability of accepting a worse state is a function of both the current value and the change in the cost function. The most common way of implementing an SA algorithm is to implement hill climbing with an accept function and modify it for SA

Game Search

Games are a form of *multi-agent environment*

- What do other agents do and how do they affect our success?
- Cooperative vs. competitive multi-agent environments.
- Competitive multi-agent environments give rise to adversarial search often known as *games*

Games – adversary

- Solution is strategy (strategy specifies move for every possible opponent reply).
- Time limits force an *approximate* solution
- Evaluation function: evaluate "goodness" of game position
- Examples: chess, checkers, Othello, backgammon

Difference between the search space of a game and the search space of a problem: In the first case it represents the moves of two (or more) players, whereas in the latter case it represents the "moves" of a single problem-solving agent.

7.1 An exemplary game: Tic-tac-toe

There are two players denoted by X and O. They are alternatively writing their letter in one of the 9 cells of a 3 by 3 board. The winner is the one who succeeds in writing three letters in line.

The game begins with an empty board. It ends in a win for one player and a loss for the other, or possibly in a draw.

A complete tree is a representation of all the possible plays of the game. The root node is the initial state, in which it is the first player's turn to move (the player X). The successors of the initial state are the states the player can reach in one move, their successors are the states resulting from the other player's possible replies, and so on.

Terminal states are those representing a win for X, loss for X, or a draw.

Each path from the root node to a terminal node gives a different complete play of the game. Figure given below shows the initial search space of Tic-Tac-Toe.

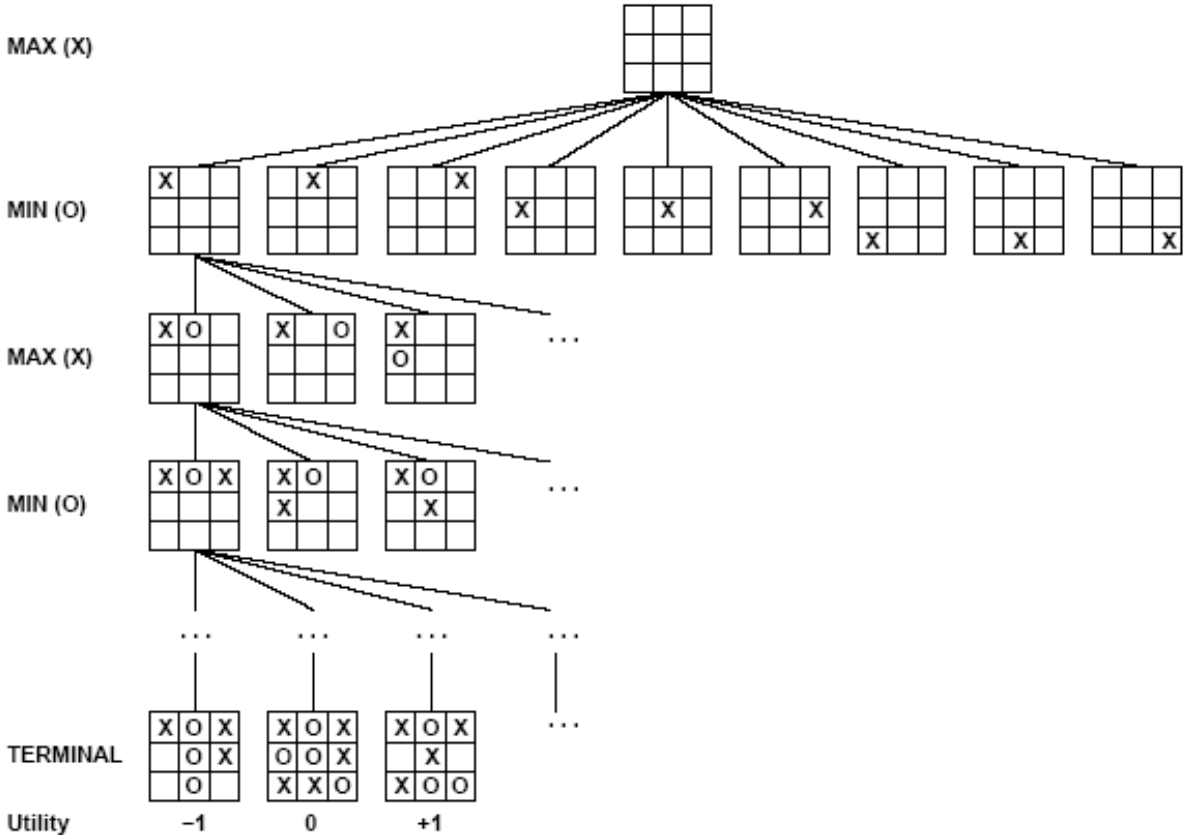


Figure 3.14: Partial game tree for Tic-Tac-Toe

A game can be formally defined as a kind of search problem as below:

- Initial state: It includes the board position and identifies the playesr to move.
- Successor function: It gives a list of (move, state) pairs each indicating a legal move and resulting state.
- Terminal test: Which determines when the game is over. States where the game is ended are called terminal states.
- Utility function: It gives numerical value of terminal states. E.g. win (+1), loose (-1) and draw (0). Some games have a wider variety of possible outcomes eg. ranging from +92 to -192.

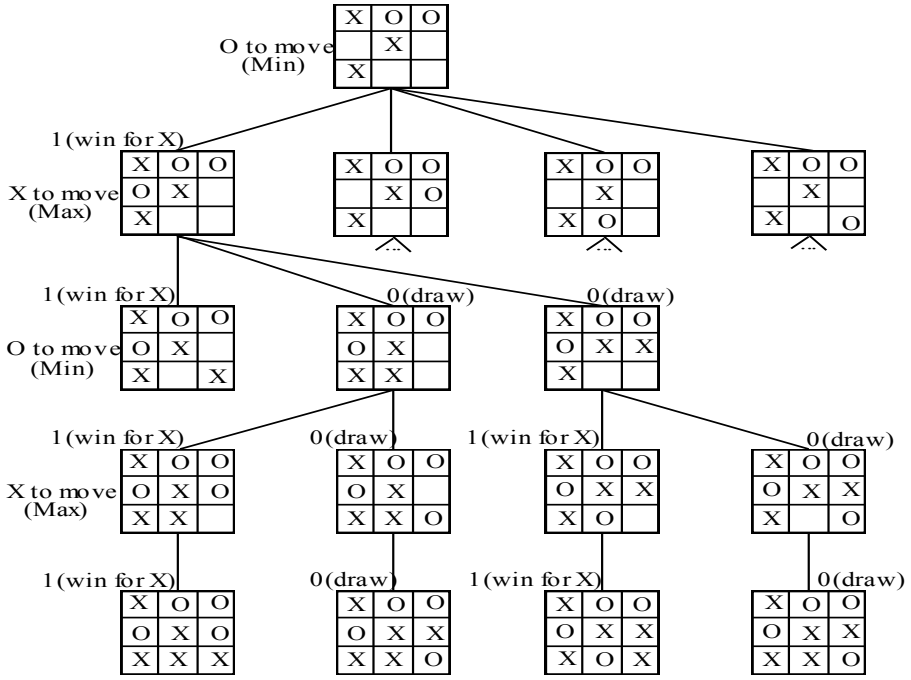
7.2 The minimax Algorithm

Let us assign the following values for the game: 1 for win by X, 0 for draw, -1 for loss by X.
Given the values of the terminal nodes (win for X (1), loss for X (-1), or draw (0)), the values of the non-terminal nodes are computed as follows:
- the value of a node where it is the turn of player X to move is the maximum of the values of its successors (because X tries to maximize its outcome);

- the value of a node where it is the turn of player O to move is the minimum of the values of its successors (because O tries to minimize the outcome of X).

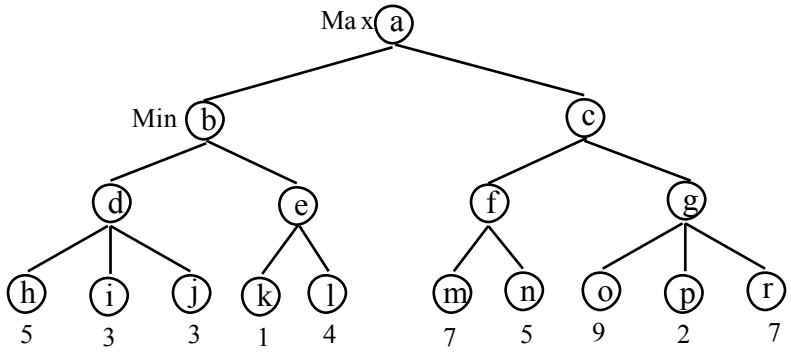
Figure below shows how the values of the nodes of the search tree are computed from the values of the leaves of the tree.
The values of the leaves of the tree are given by the rules of the game:

- 1 if there are three X in a row, column or diagonal;
- -1 if there are three O in a row, column or diagonal;
- 0 otherwise



Example

Consider the following game tree (drawn from the point of view of the Maximizing player):



Show what moves should be chosen by the two players, assuming that both are using the mini-max procedure.

Solution:

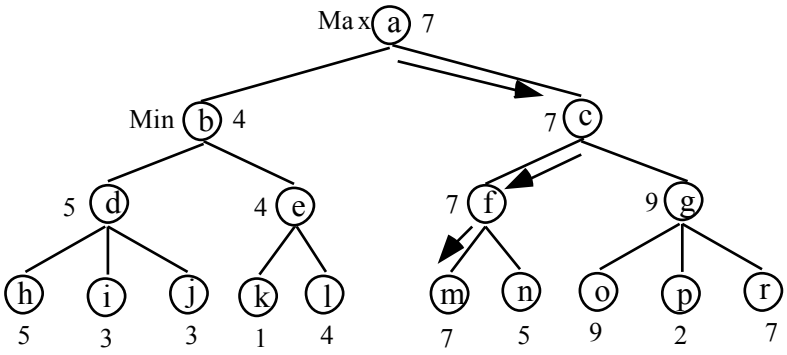


Figure 3.16: The mini-max path for the game tree

Alpha-beta pruning

Alpha-beta pruning is a technique for evaluating nodes of a game tree that eliminates unnecessary evaluations. It uses two parameters, alpha and beta.

An alpha cutoff

To apply this technique, one uses a parameter called alpha that represents a lower bound for the achievement of the Max player at a given node.

Let us consider that the current board situation corresponds to the node A in the following figure.

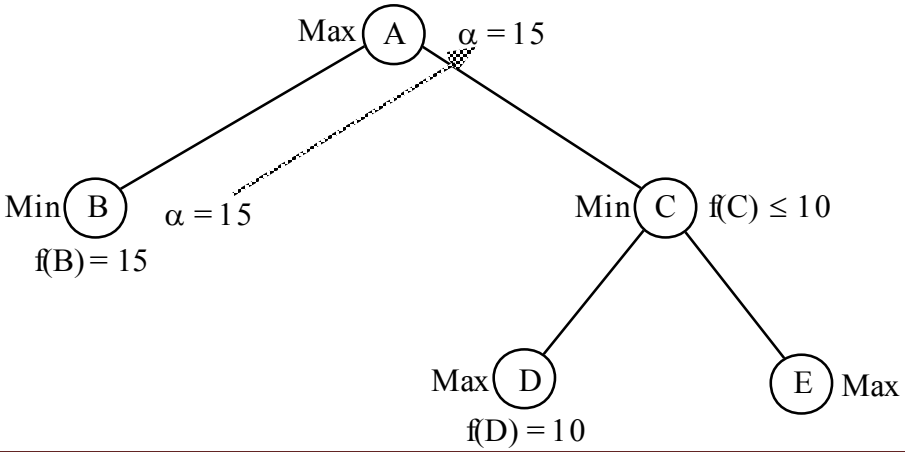


Figure 3.17: Illustration of the alpha cut-off.

The minimax method uses a depth-first search strategy in evaluating the descendants of a node. It will therefore estimate first the value of the node B. Let us suppose that this value has been evaluated to 15, either by using a static evaluation function, or by backing up from descendants omitted in the figure. If Max will move to B then it is guaranteed to achieve 15. Therefore 15 is a lower bound for the achievement of the Max player (it may still be possible to achieve more, depending on the values of the other descendants of A). Therefore, the value of α at node B is 15. This value is transmitted upward to the node A and will be used for evaluating the other possible moves from A.

To evaluate the node C, its left-most child D has to be evaluated first. Let us assume that the value of D is 10 (this value has been obtained either by applying a static evaluation function directly to D, or by backing up values from descendants omitted in the figure). Because this value is less than the value of α, the best move for Max is to node B, independent of the value of node E that need not be evaluated. Indeed, if the value of E is greater than 10, Min will move to D which has the value 10 for Max. Otherwise, if the value of E is less than 10, Min will move to E which has a value less than 10. So, if Max moves to C, the best it can get is 10, which is less than the value $\alpha = 15$ that would be gotten if Max would move to B. Therefore, the best move for Max is to B, independent of the value of E. The elimination of the node E is an alpha cutoff.

One should notice that E may itself have a huge subtree. Therefore, the elimination of E means, in fact, the elimination of this subtree.

A beta cutoff

To apply this technique, one uses a parameter called beta that represents an upper bound for the achievement of the Max player at a given node.

In the above tree, the Max player moved to the node B. Now it is the turn of the Min player to decide where to move:

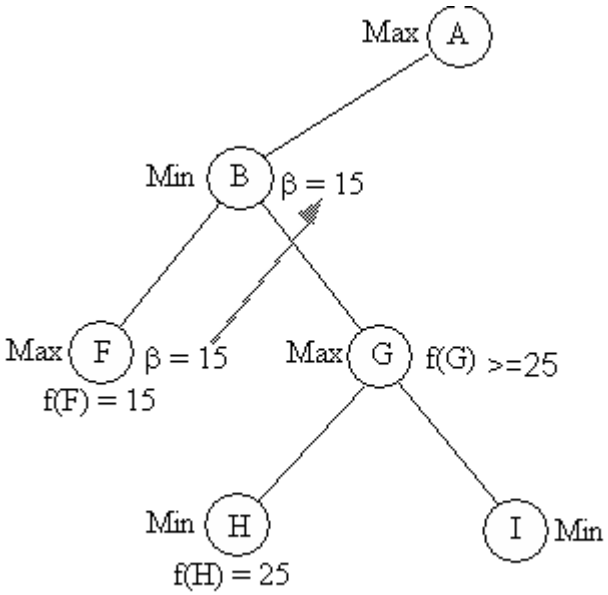


Figure 3.18: Illustration of the beta cut-off.

The Min player also evaluates its descendants in a depth-first order.

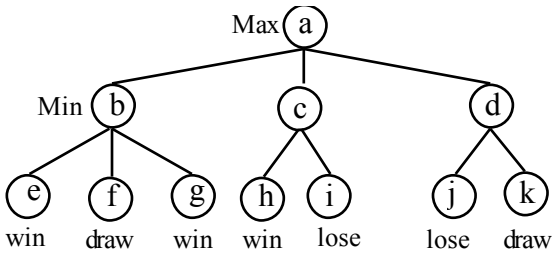
Let us assume that the value of F has been evaluated to 15. From the point of view of Min, this is an upper bound for the achievement of Min (it may still be possible to make Min achieve less, depending of the values of the other descendants of B). Therefore the value of β at the node F is 15. This value is transmitted upward to the node B and will be used for evaluating the other possible moves from B.

To evaluate the node G, its left-most child H is evaluated first. Let us assume that the value of H is 25 (this value has been obtained either by applying a static evaluation function directly to H, or by backing up values from descendants omitted in the figure). Because this value is greater than the value of β, the best move for Min is to node F, independent of the value of node I that need not be evaluated. Indeed, if the value of I is $v \geq 25$, then Max (in G) will move to I. Otherwise, if the value of I is less than 25, Max will move to H. So in both cases, the value obtained by Max is at least 25 which is greater than β (the best value obtained by Max if Min moves to F).
Therefore, the best move for Min is at F, independent of the value of I. The elimination of the node I is a beta cutoff.

One should notice that by applying alpha and beta cut-off, one obtains the same results as in the case of mini-max, but (in general) with less effort. This means that, in a given amount of time, one could search deeper in the game tree than in the case of mini-max.

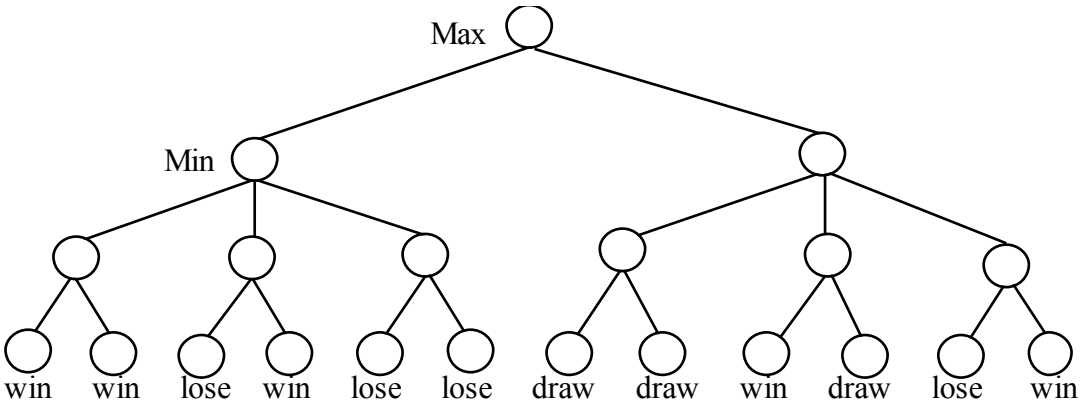
Exercises:

1. Consider the following game tree (drawn from the point of view of the Maximizing player):



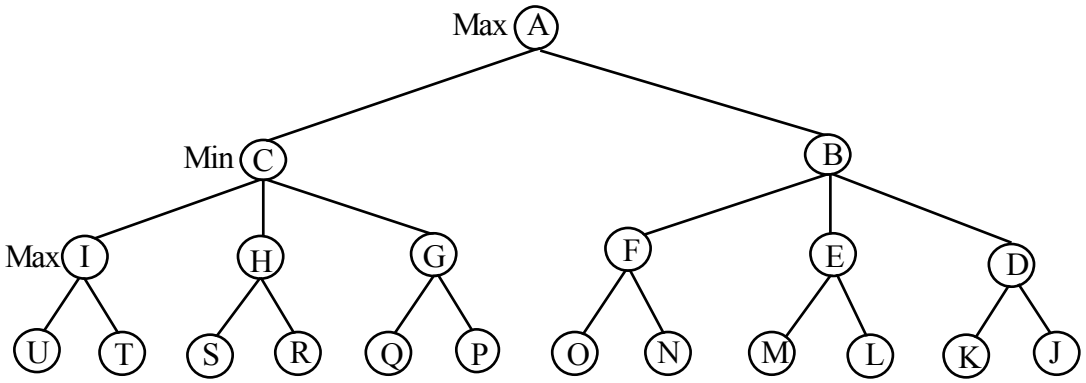
What move should be chosen by the Max player, and what should be the response of the Min player, assuming that both are using the mini-max procedure?

2. Consider the following game tree (drown from the point of view of the Maximizing player):



What move should be chosen by the Max player, and what should be the response of the Min player, assuming that both are using the mini-max procedure?

3. Consider the following two-person game tree, drown from the point of view of the Maximizing player:

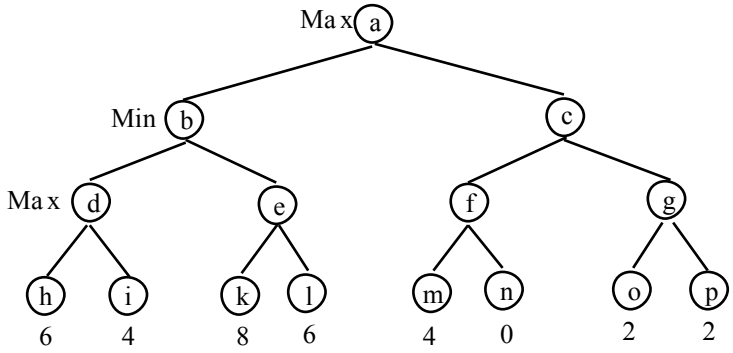


Assume a static evaluation function which produces the following values:

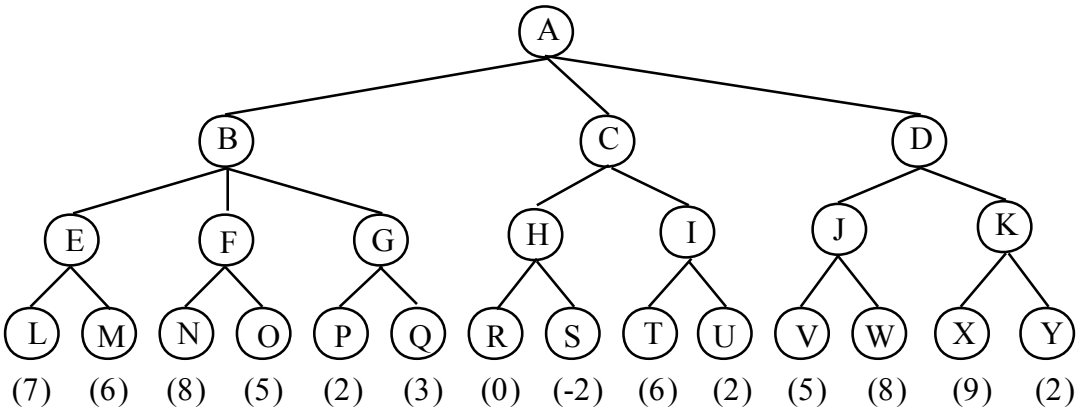
f(J)=win	f(K)=win	f(L)=lose	f(M)=lose	f(N)=win
f(O)=lose	f(P)=lose	f(Q)=win	f(R)=draw	f(S)=draw
f(T)=draw	f(U)=win			

Use the alpha-beta pruning procedure to determine the moves which should be chosen by the two players, assuming a depth-first search (generation) of the tree. Which nodes need not to be examined (generated)?

4. Consider the following game tree (drown from the point of view of the Maximizing player):



a) Use the mini-max procedure and show what moves should be chosen by the two players.
b) Use the alpha-beta pruning procedure and show what nodes would not need to be examined.
5. Consider the following game tree in which static scores are all from the first player's point of view:

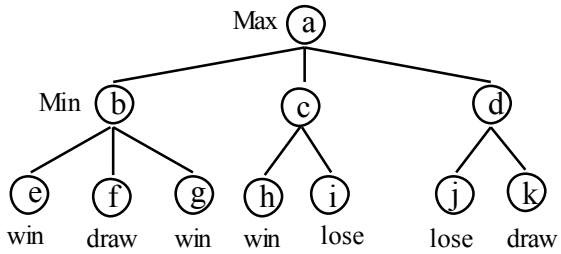


Suppose the first player is the maximizing player. What move should be chosen?

what nodes would not need to be examined using the alpha-beta pruning procedure?

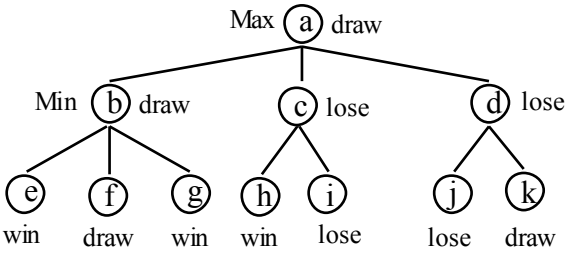
Solutions to exercises

1. Consider the following game tree (drawn from the point of view of the Maximizing player):



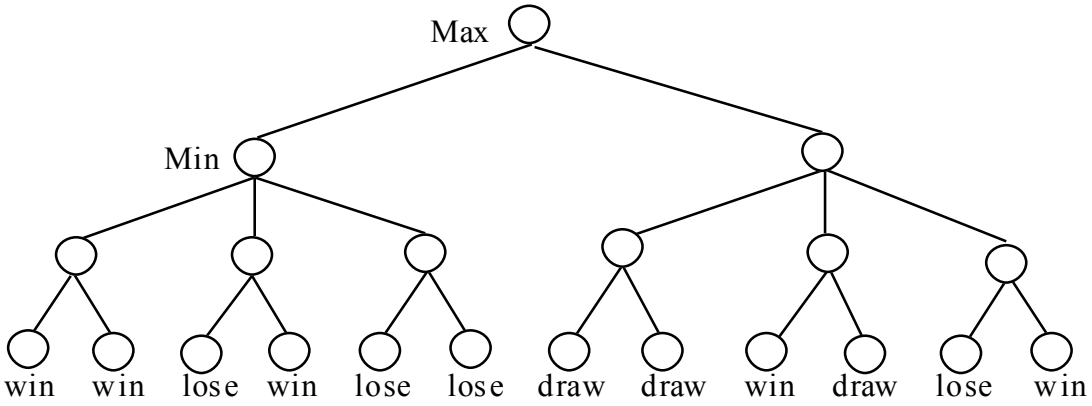
What move should be chosen by the Max player, and what should be the response of the Min player, assuming that both are using the mini-max procedure ?

Solution:



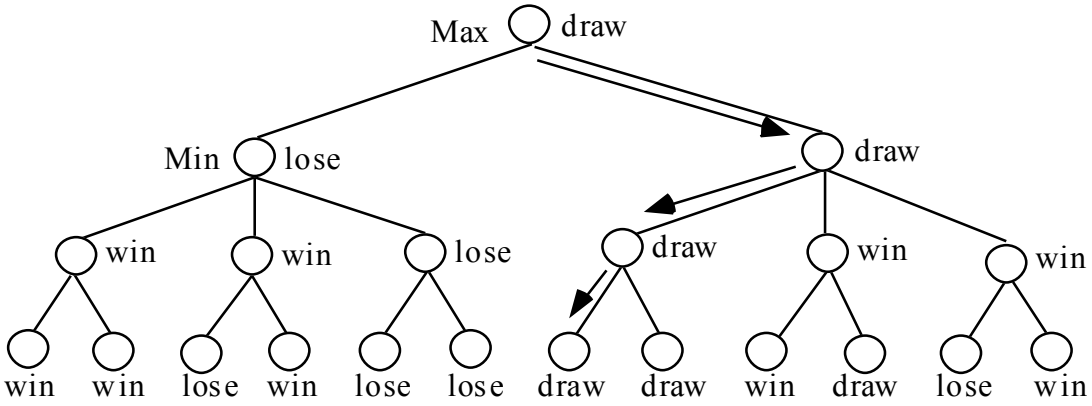
Max will move to b and min will respond by moving to f.

2. Consider the following game tree (drown from the point of view of the Maximizing player):

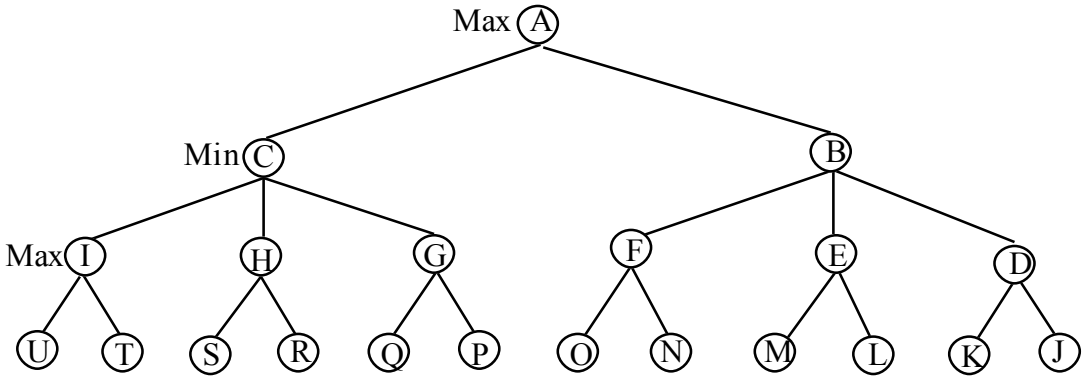


What move should be chosen by the Max player, and what should be the response of the Min player, assuming that both are using the mini-max procedure?

Solution:



3. Consider the following two-person game tree, drown from the point of view of the Maximizing player:

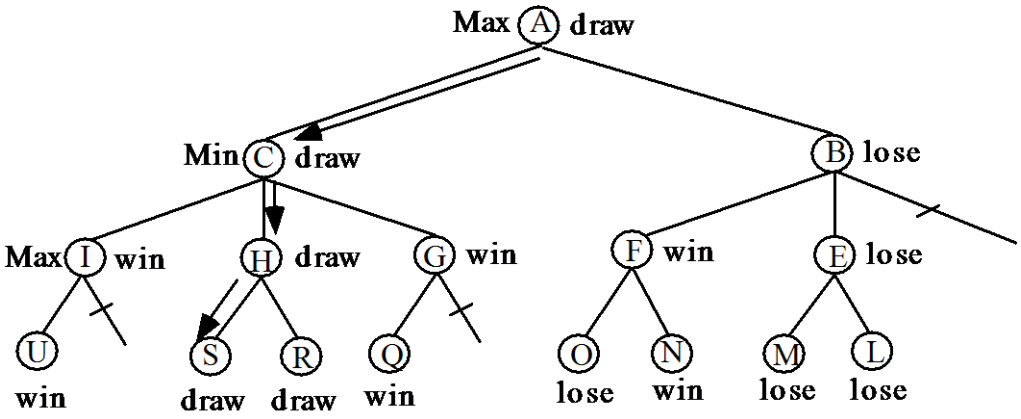


Assume a static evaluation function which produces the following values:

f(J)=win	f(K)=win	f(L)=lose	f(M)=lose	f(N)=win
f(O)=lose	f(P)=lose	f(Q)=win	f(R)=draw	f(S)=draw
f(T)=draw	f(U)=win			

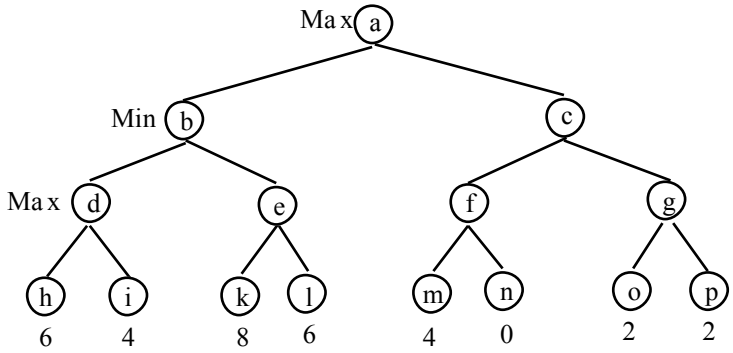
Use the alpha-beta pruning procedure to determine the moves which should be chosen by the two players, assuming a depth-first search (generation) of the tree. Which nodes need not to be examined (generated)?

a) The following figure shows the application of the alpha-beta pruning procedure on:



Max moves to C and Min moves to H and Max moves to either S or R.
The game ends in a draw.

4. Consider the following game tree (drown from the point of view of the Maximizing player):

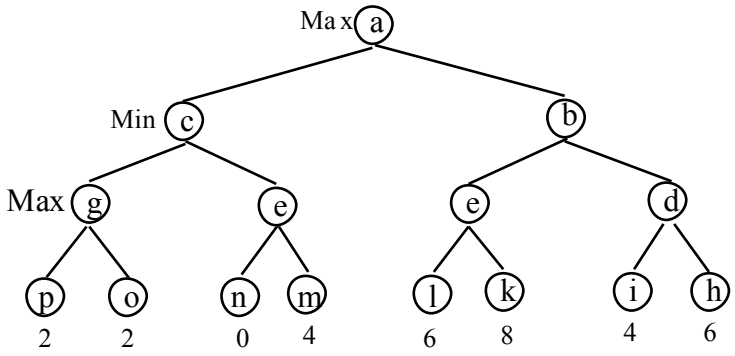


a) Use the mini-max procedure and show what moves should be chosen by the two players.

b) Use the alpha-beta pruning procedure and show what nodes would not need to be examined.

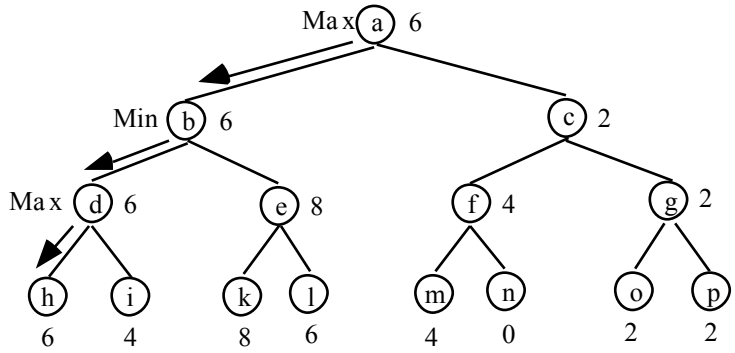
c) Consider the mirror image of the above tree and apply again the alpha-beta pruning procedure. What do you notice?

The mirror image is the following tree:

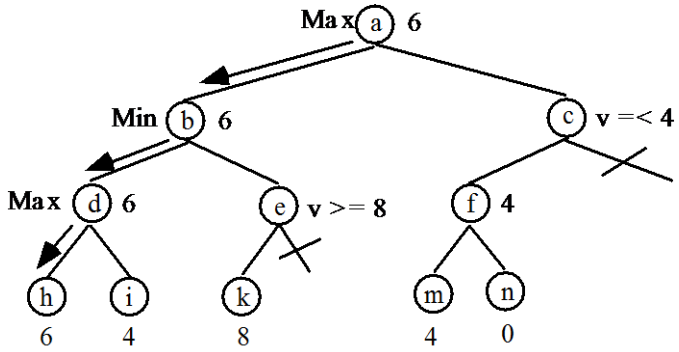


Solution:

a)



b) The cut branches need not be examined:



c) Using alpha-beta pruning on the mirror image of the tree does not produce any savings: