Chapter 3

Problem solving by Searching

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

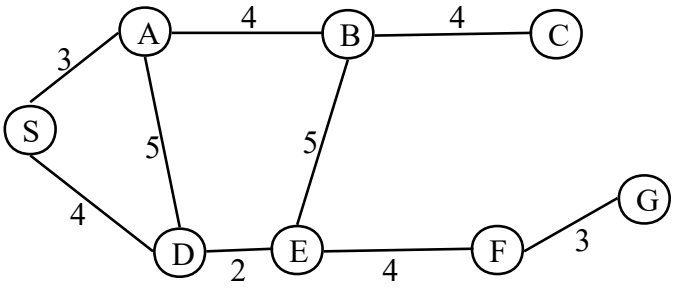


Figure : A graph representation of a map

This problem will be used to illustrate some search methods.

Search problems are part of a large number of real world applications:

- VLSI layout
- Path planning
- Robot navigation etc.

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?

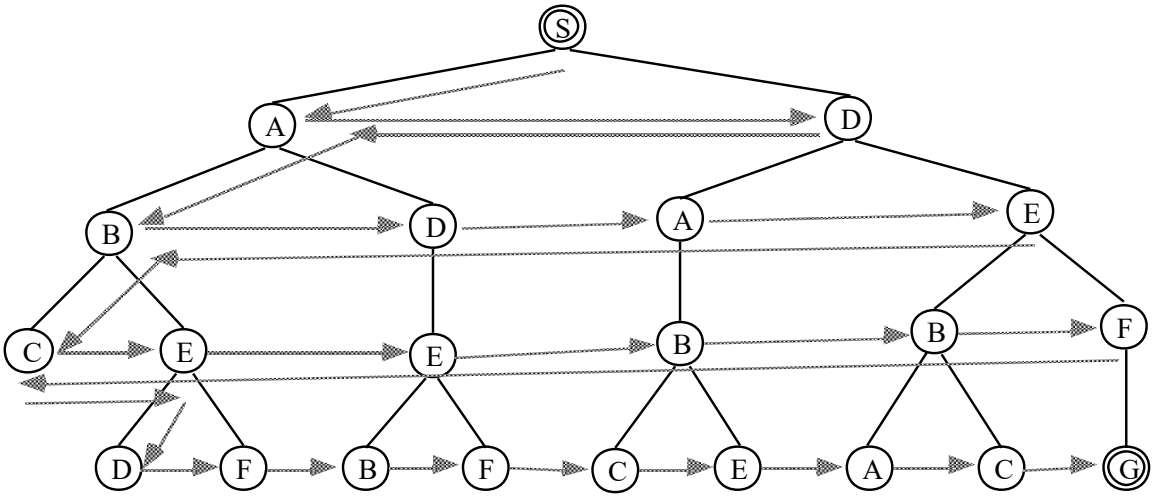
Time and space complexity are measured in terms of
b -- maximum branching factor (number of successor of any node) of the search tree
d -- depth of the least-cost solution
m -- maximum length of any path in the space

Breadth First Search

All nodes are expended at a given depth in the search tree before any nodes at the next level are expanded until the goal reached.

Expand *shallowest* unexpanded node. *fringe* is implemented as a FIFO queue

Constraint: Do not generate as child node if the node is already parent to avoid more loop.



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

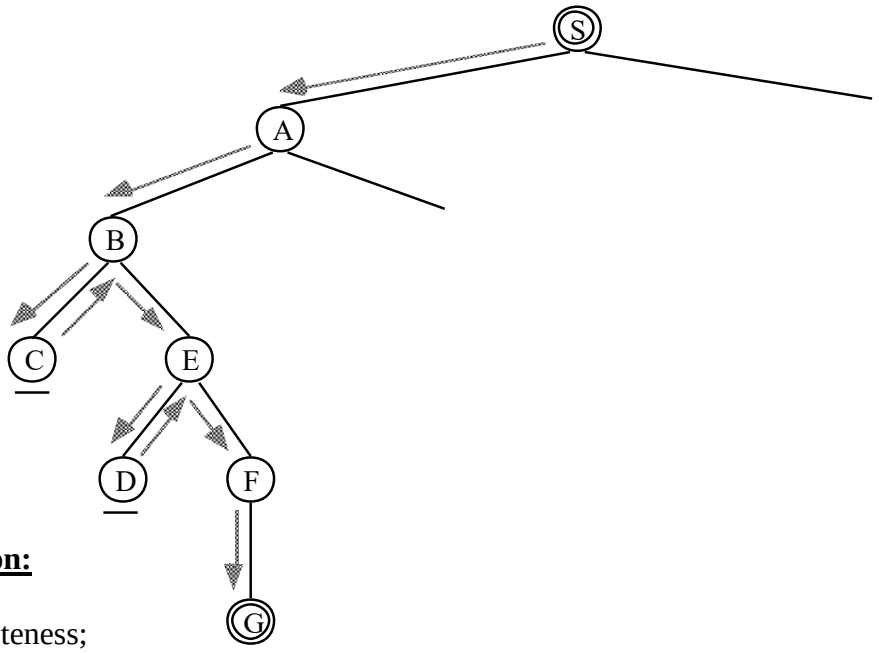
Two lessons:

- 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

Looks for the goal node among all the children of the current node before using the sibling of this node i.e. **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:

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

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.

Uniform Cost Search:

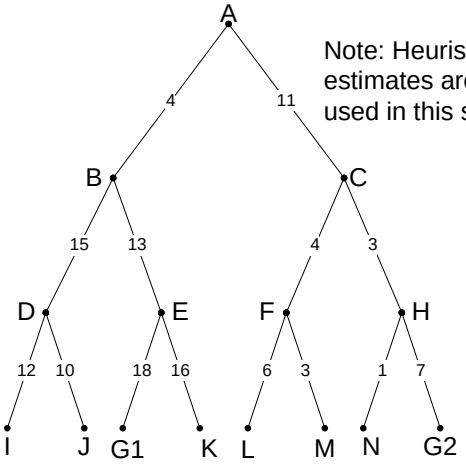
Uniform-cost search (**UCS**) is modified version of BFS to make optimal. It is basically a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph. The search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

Typically, the search algorithm involves expanding nodes by adding all unexpanded neighboring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its total path cost from the root, where the least-cost paths are given highest priority. The node at the head of the queue is subsequently expanded, adding the next set of connected nodes with the total path cost from the root to the respective node. The uniform-cost search is **complete** and **optimal** if the cost of each step exceeds some positive bound ε.

Does not care about the number of steps, only care about total cost.

•Complete? Yes, if step cost $\geq \varepsilon$ (small positive number).
•Time? Maximum as of BFS
•Space? Maximum as of BFS.
•Optimal? Yes

Consider an example:



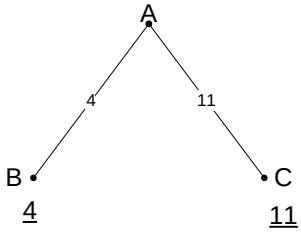
Note: Heurisitic estimates are not used in this search!

Start with root node A.

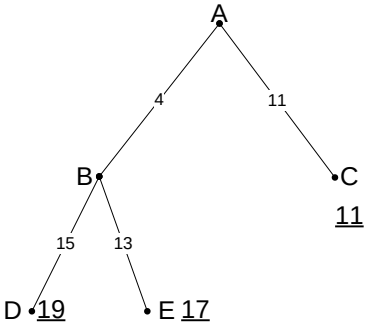
A

0

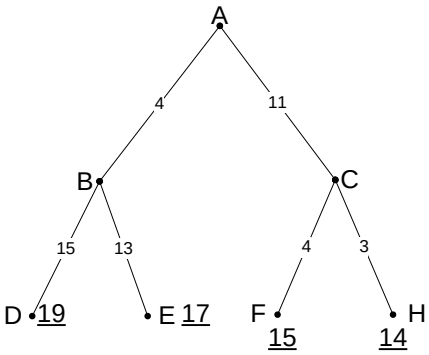
Paths from root are generated.



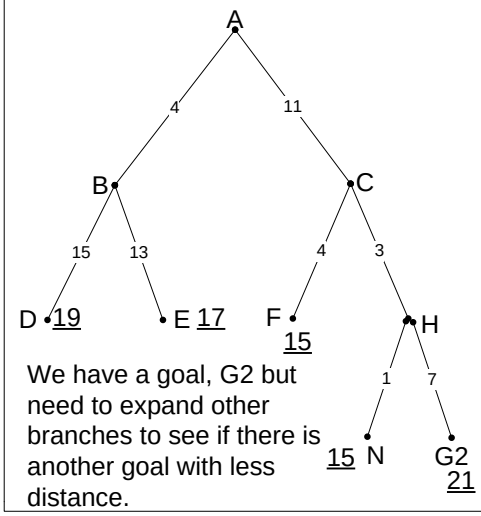
Since B has the least cost, we expand it.



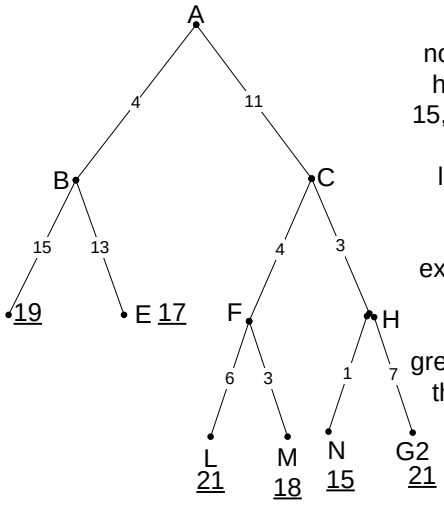
Of our 3 choices, C has the least cost so we'll expand it.



Node H has the least cost thus far, so we expand it.



We have a goal, G2 but need to expand other branches to see if there is another goal with less distance.



Note: Both nodes F and N have a cost of 15, we chose to expand the leftmost node first. We continue expanding until all remaining paths are greater than 21, the cost of G2

Depth Limited Search:

The problem of unbounded trees can be solve by supplying depth-first search with a determined depth limit (nodes at depth are treated as they have no successors) –**Depth limited search. Depth-limited search** is an algorithm to explore the vertices of a graph. It is a modification of depth-first search and is used for example in the iterative deepening depth-first search algorithm.

Like the normal depth-first search, depth-limited search is an uninformed search. It works exactly like depth-first search, but avoids its drawbacks regarding completeness by imposing a maximum limit on the depth of the search. Even if the search could still expand a vertex beyond that depth, it will not do so and thereby it will not follow infinitely deep paths or get stuck in cycles. Therefore depth-limited search will find a solution if it is within the depth limit, which guarantees at least completeness on all graphs.

It 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: $O (bl)$

Iterative Deepening Depth First Search:

In this strategy, depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches *d*, the depth of the shallowest goal state. On each iteration, IDDFS visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited, assuming no pruning, is effectively breadth-first.

IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness (when the branching factor is finite). It is optimal when the path cost is a non-decreasing function of the depth of the node.

The technique of *iterative deepening* is based on this idea. *Iterative deepening* is depth-first search to a fixed depth in the tree being searched. If no solution is found up to this depth then the depth to be searched is increased and the whole `bounded' depth-first search begun again.

t works by setting a depth of search -say, depth 1- and doing depth-first search to that depth. If a solution is found then the process stops -otherwise, increase the depth by, say, 1 and repeat until a solution is found. Note that every time we start up a new bounded depth search *we start from scratch* - i.e. we throw away any results from the previous search.

Now *iterative deepening* is a popular method of search. We explain why this is so.

Depth-first search can be implemented to be much cheaper than breadth-first search in terms of memory usage -but it is not guaranteed to find a solution even where one is guaranteed.

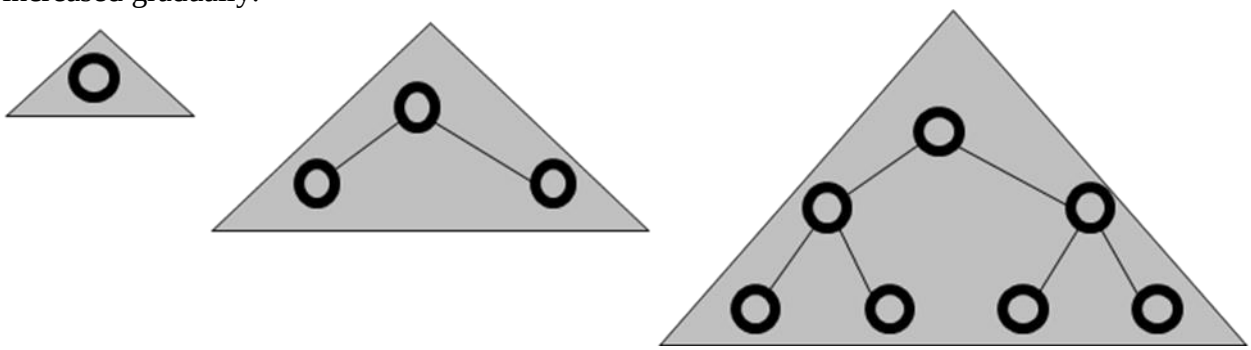
On the other hand, breadth-first search can be guaranteed to terminate if there is a winning state to be found and will always find the `quickest' solution (in terms of how many steps need to be taken from the root node). It is, however, a very expensive method in terms of memory usage.

Iterative deepening is liked because it is an effective compromise between the two other methods of search. It is a form of depth-first search with a lower bound on how deep the search can go. Iterative deepening terminates if there is a solution. It can produce the same solution that breadth-first search would produce but does not require the same memory usage (as for breadth-first search).

Note that depth-first search achieves its efficiency by generating the next node to explore only when this needed. The breadth-first search algorithm has to grow all the search paths available until a solution is found -and this takes up memory. Iterative deepening achieves its memory saving in the same way that depth-first search does -at the expense of redoing some computations again and again (a time cost rather than a memory one). In the search illustrated, we had to visit node d three times in all!

- Complete (like BFS)
- Has linear memory requirements (like DFS)
- Classical time-space tradeoff.
- This is the preferred method for large state spaces, where the solution path length is unknown.

The overall idea goes as follows until the goal node is not found i.e. the depth limit is increased gradually.



Iterative Deepening 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.

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.

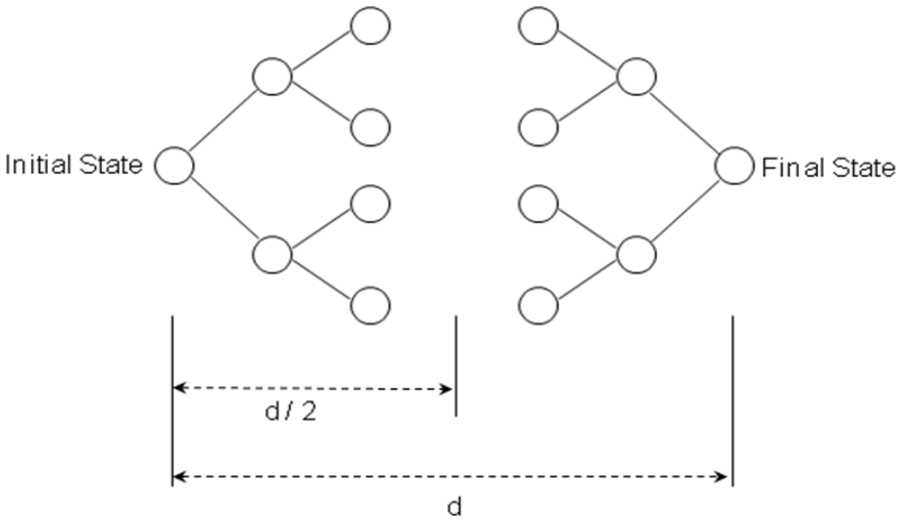
Example: Number of nodes generated for b=10 and d=5 solution at far right

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

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

Bidirectional Search:

This is a search algorithm which replaces a single search graph, which is likely to with two smaller graphs -- one starting from the initial state and one starting from the goal state. It then, expands nodes from the start and goal state simultaneously. Check at each stage if the nodes of one have been generated by the other, i.e, they meet in the middle. If so, the path concatenation is the solution.



- Completeness: yes
- Optimality: yes (If done with correct strategy- e.g. breadth first)
- Time complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$

Problems: generate predecessors; many goal states; efficient check for node already visited by other half of the search; and, what kind of search.

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.

Heuristic Searches - Why Use?

- It may be too resource intensive (both time and space) to use a blind search
- Even if a blind search will work we may want a more efficient search method

Informed Search uses domain specific information to improve the search pattern

- Define a heuristic function, h(n), that estimates the "goodness" of a node n.
- Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
- The heuristic function is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal.

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

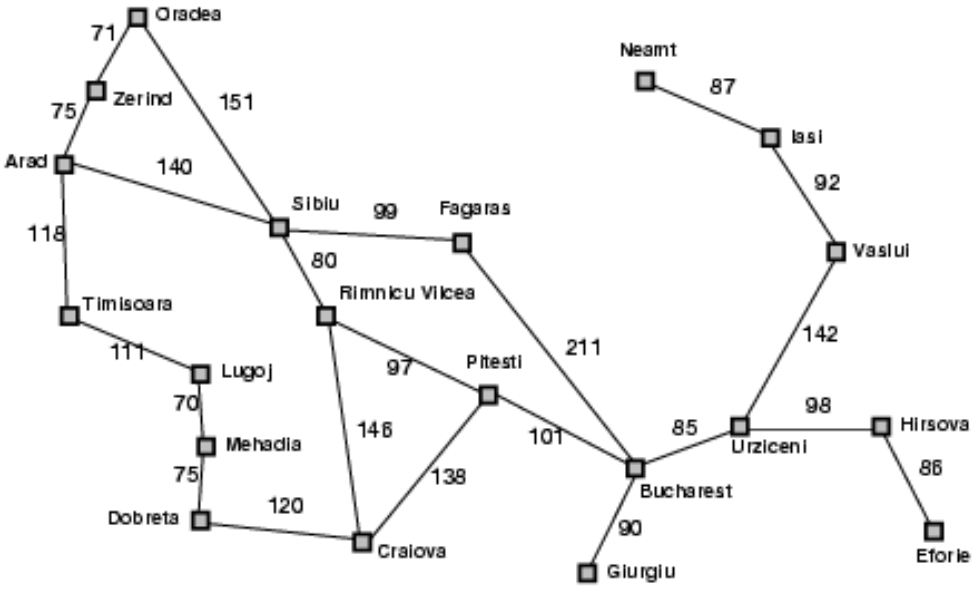
The best-first search part of the name means that it uses an evaluation function to select which node is to be expanded next. The node with the lowest evaluation is selected for expansion because that is the *best* node, since it supposedly has the closest path to the goal (if the heuristic is good). Unlike A* which uses both the link costs and a heuristic of the cost to the goal, greedy best-first search uses only the heuristic, and not any link costs. A disadvantage of this approach is that if the heuristic is not accurate, it can go down paths with high link cost since there might be a low heuristic for the connecting node.

Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from n to *goal*.

e.g., $h_{SLD}(n)$ = straight-line distance from n to goal

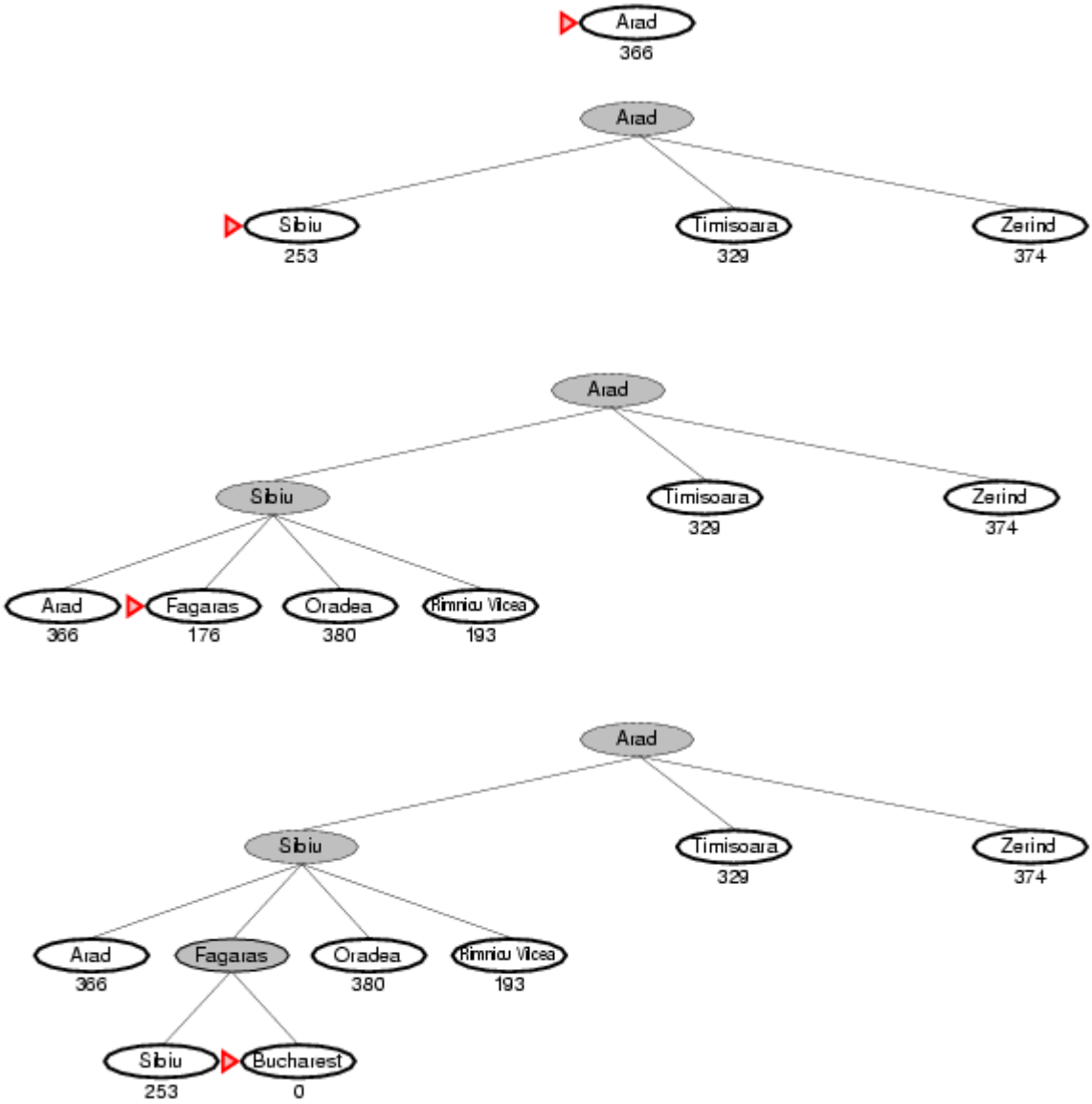
Greedy best-first search expands the node that appears to be closest to goal. The greedy best-first search algorithm is $O(b^m)$ in terms of space and time complexity. (Where b is the average branching factor (the average number of successors from a state), and m is the maximum depth of the search tree.)

Example: Given following graph of cities, starting at Arad city, problem is to reach to the Bucharest.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Solution using greedy best first can be as below:



Greedy Best-first search

- minimizes estimated cost h(n) from current node n to goal;
- is informed but (almost always) suboptimal and incomplete.

Admissible Heuristic:

A heuristic function is said to be **admissible** if it is no more than the lowest-cost path to the goal. In other words, a heuristic is admissible if it never overestimates the cost of reaching the goal. An admissible heuristic is also known as an **optimistic heuristic**.

An admissible heuristic is used to estimate the cost of reaching the goal state in an informed search algorithm. In order for a heuristic to be admissible to the search problem, the estimated cost must always be lower than the actual cost of reaching the goal state. The search algorithm uses the admissible heuristic to find an estimated optimal path to the goal

state from the current node. For example, in A* search the evaluation function (where *n* is the current node) is: $f(n) = g(n) + h(n)$

where;

$f(n)$ = the evaluation function.
$g(n)$ = the cost from the start node to the current node
$h(n)$ = estimated cost from current node to goal.

$h(n)$ is calculated using the heuristic function. *With a non-admissible heuristic, the A* algorithm would overlook the optimal solution to a search problem due to an overestimation in f(n).*

It is obvious that the SLD heuristic function is admissible as we can never find a shorter distance between any two towns.

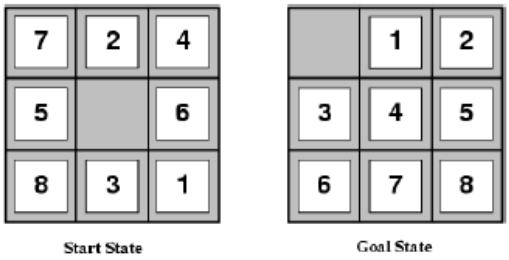
Formulating admissible heuristics:

- *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

<u>Consistency (Monotonicity)</u>

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 is less than the sum of step cost from N to N' and estimated cost from node N' to goal node.
i.e $h(n) \leq c(n, n') + h(n')$
Where;

h(n) = Estimated cost to reach to the goal node from node n
c(n, n') = actual cost from n to n'

A* Search:

A* is a best first, informed graph search algorithm. A* is different from other best first search algorithms in that it uses a heuristic function h(x) as well as the path cost to the node g(x), in computing the cost f(x) = h(x) + g(x) for the node. The *h*(*x*) part of the *f*(*x*) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, *h*(*x*) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

It finds a minimal cost-path joining the start node and a goal node for node n. Evaluation function: f(n) = g(n) + h(n)

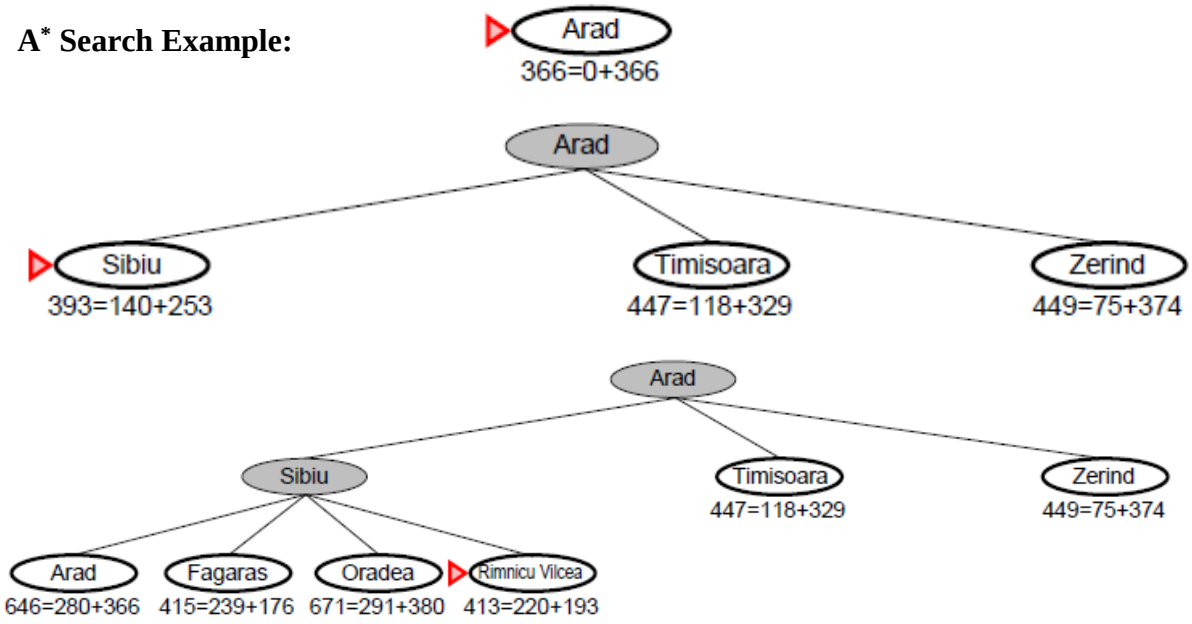
Where,

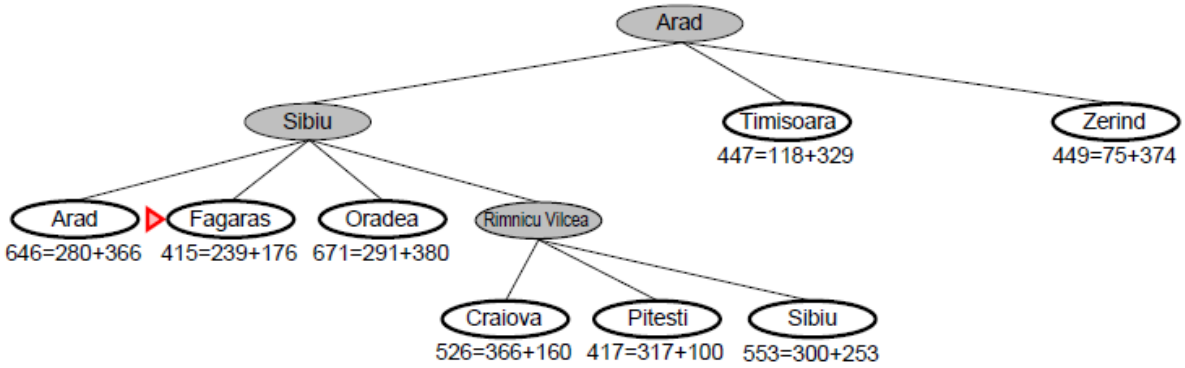
g(n) = cost so far to reach n from root
h(n) = estimated cost to goal from n
f(n) = estimated total cost of path through n to goal

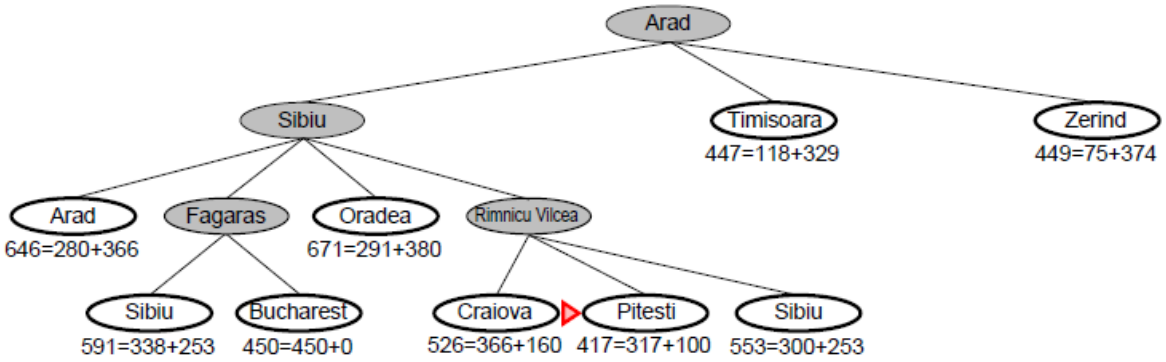
- combines the two by minimizing f(n) = g(n) + h(n);
- is informed and, *under reasonable assumptions*, optimal and complete.

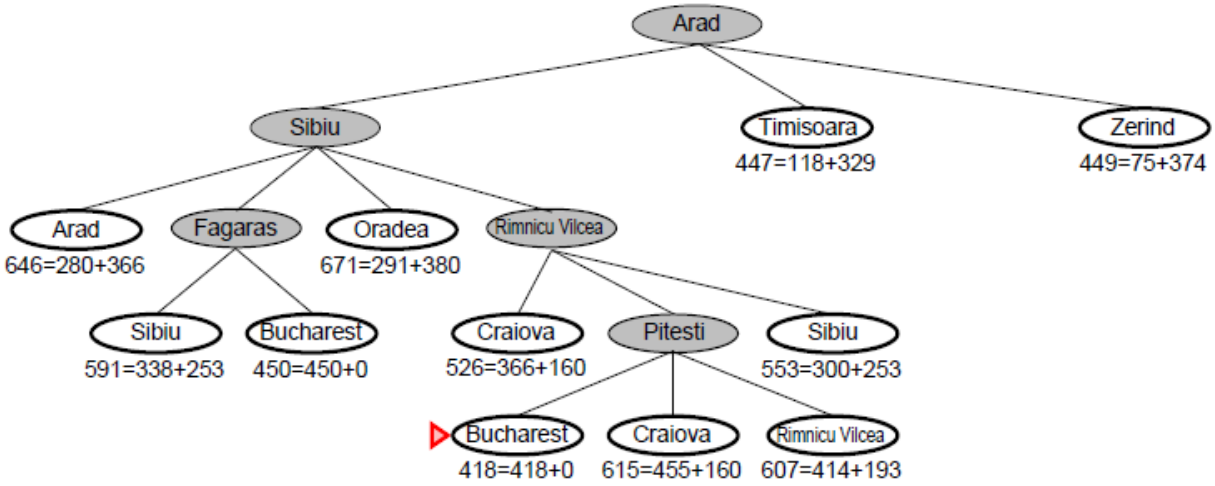
As A* traverses the graph, it follows a path of the lowest *known* path, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached.

A* Search Example:









Admissibility and Optimality:

A* is admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A* uses an "optimistic" estimate of the cost of a path through every node that it considers—optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

Here is the main idea of the proof:

When A* terminates its search, it has found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.

Suppose, now that some other search algorithm B terminates its search with a path whose actual cost is *not* less than the estimated cost of a path through some open node. Based on the heuristic information it has, Algorithm B cannot rule out the possibility that a path through that node has a lower cost. So while B might consider fewer nodes than A*, it cannot be admissible. Accordingly, A* considers the fewest nodes of any admissible search algorithm.

This is only true if both:

- A* uses an admissible heuristic. Otherwise, A* is not guaranteed to expand fewer nodes than another search algorithm with the same heuristic.

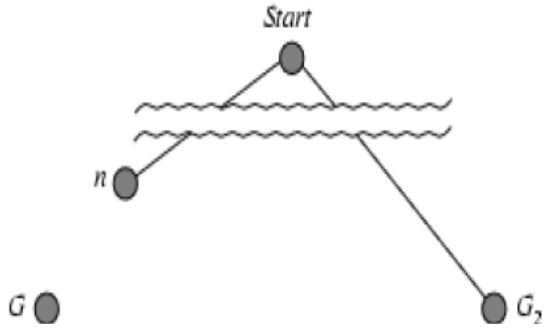
- A* solves only one search problem rather than a series of similar search problems. Otherwise, A* is not guaranteed to expand fewer nodes than incremental heuristic search algorithms

Thus, if estimated distance h(n) never exceed the true distance h*(n) between the current node to goal node, the A* algorithm will always find a shortest path -This is known as the *admissibility* of A* algorithm and h(n) is a admissible heuristic.

IF 0 =< h (n) =< h*(n), and costs of all arcs are positive
THEN A* is guaranteed to find a solution path of minimal cost if any solution path exists.

Theorem: *A* is optimal if h(n) is admissible*.

Suppose suboptimal goal *G2* in the queue.
Let *n* be an unexpanded node on a shortest path 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 admissible, It does not overestimates the cost of completing the solution path.

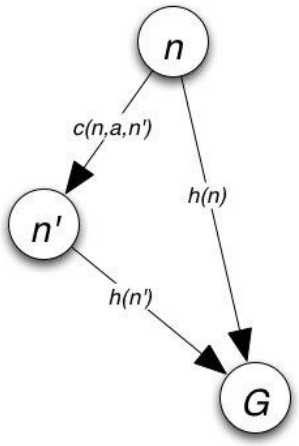
$f(n) = g(n) + h(n) \leq C^*$ ……(2)

Now from (1) and (2)

$f(n) \leq C^* < f(G2)$

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

Theorem: ***If h(n) is consistent , then the values of f(n) along the path are non-decreasing.***



Suppose n' is successor of n, then

$$g(n') = g(n) + C(n, a, n')$$

We know that,

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

$$f(n') = g(n) + C(n, a, n') + h(n') \quad \ldots\ldots\ldots\ldots\ldots\ldots (1)$$

A heuristic is consistent if

$$h(n) \leq C(n, a, n') + h(n') \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Now from (1) and (2)

$$f(n') = g(n) + C(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

$$f(n') \geq f(n)$$

f(n) is non-decreasing along any path.

Hill Climbing Search:

Hill climbing can be used to solve problems that have many solutions, some of which are better than others. **It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates.** Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

For example, hill climbing can be applied to the traveling salesman problem. It is easy to find a solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much better route is obtained. In hill climbing the basic idea is to always head towards a state which is better than the current one. So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

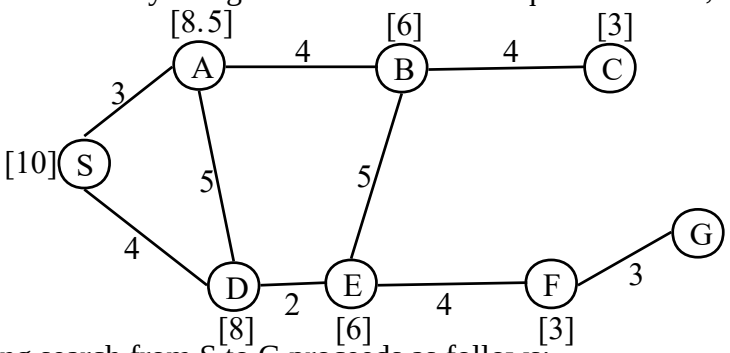
The hill climbing can be described as follows:

1. Start with *current-state* = initial-state.
2. Until *current-state* = goal-state OR there is no change in *current-state* do:
    - Get the successors of the current state and use the evaluation function to assign a score to each successor.
    - If one of the successors has a better score than the current-state then set the new current-state to be the successor with the best score.

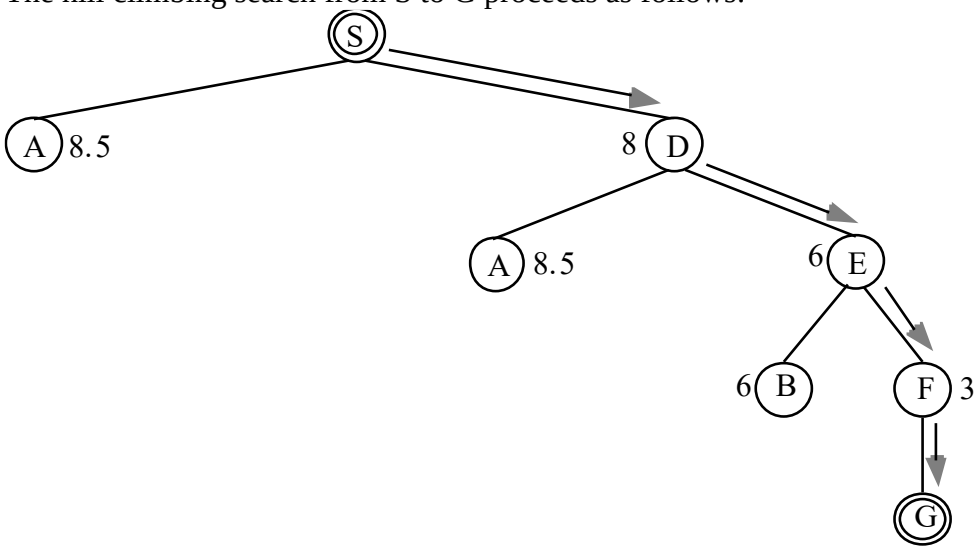
Hill climbing terminates when there are no successors of the current state which are better than the current state itself.

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 goal G is indicated in square brackets, i.e. the heuristic.

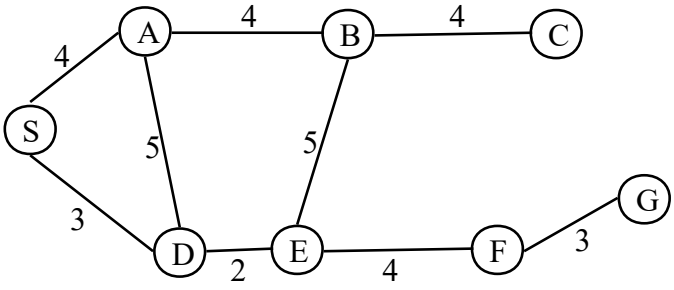


The hill climbing search from S to G proceeds as follows:



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

$$P = \text{exponential}(-c/t) > r$$

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

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter *T* (called the *temperature*), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when *T* is large, but increasingly

"downhill" as *T* goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local optima—which are the bane of greedier methods.

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.

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.

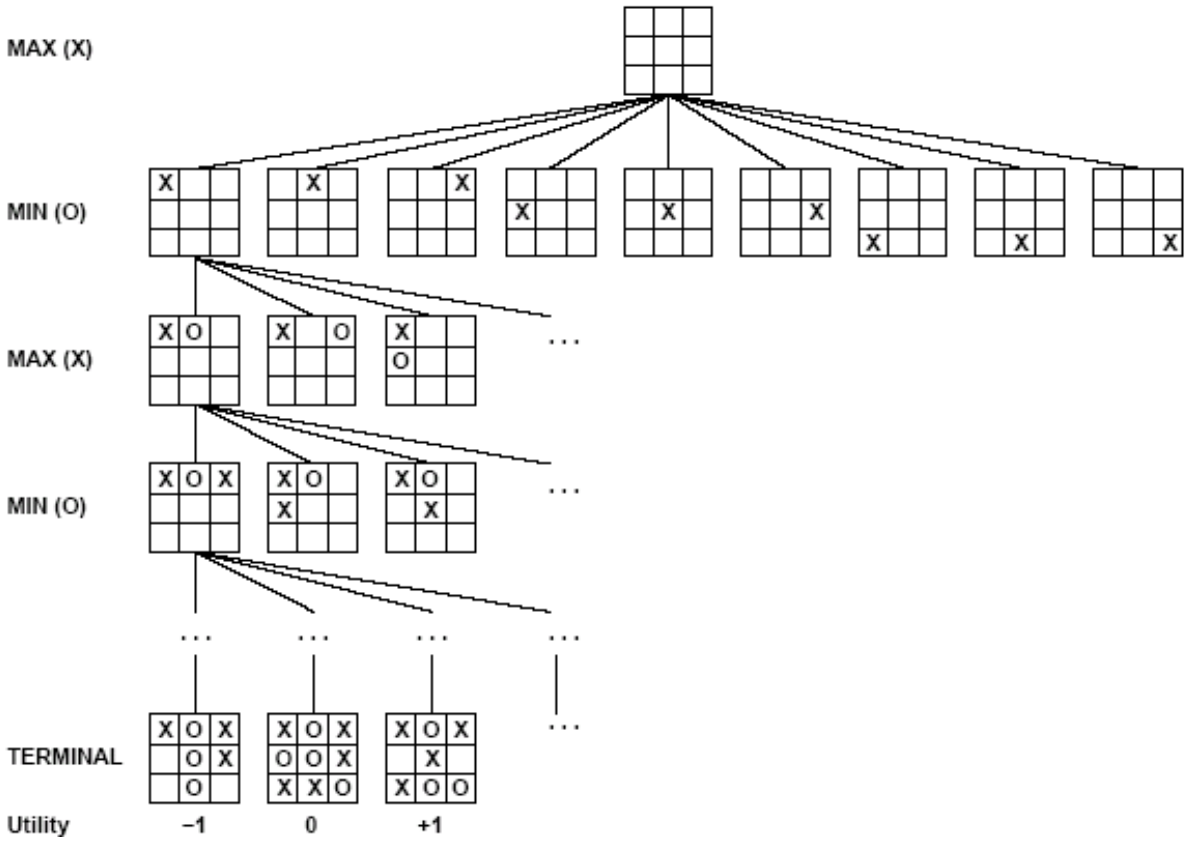


Fig: Partial game tree for Tic-Tac-Toe

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: This 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.

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.

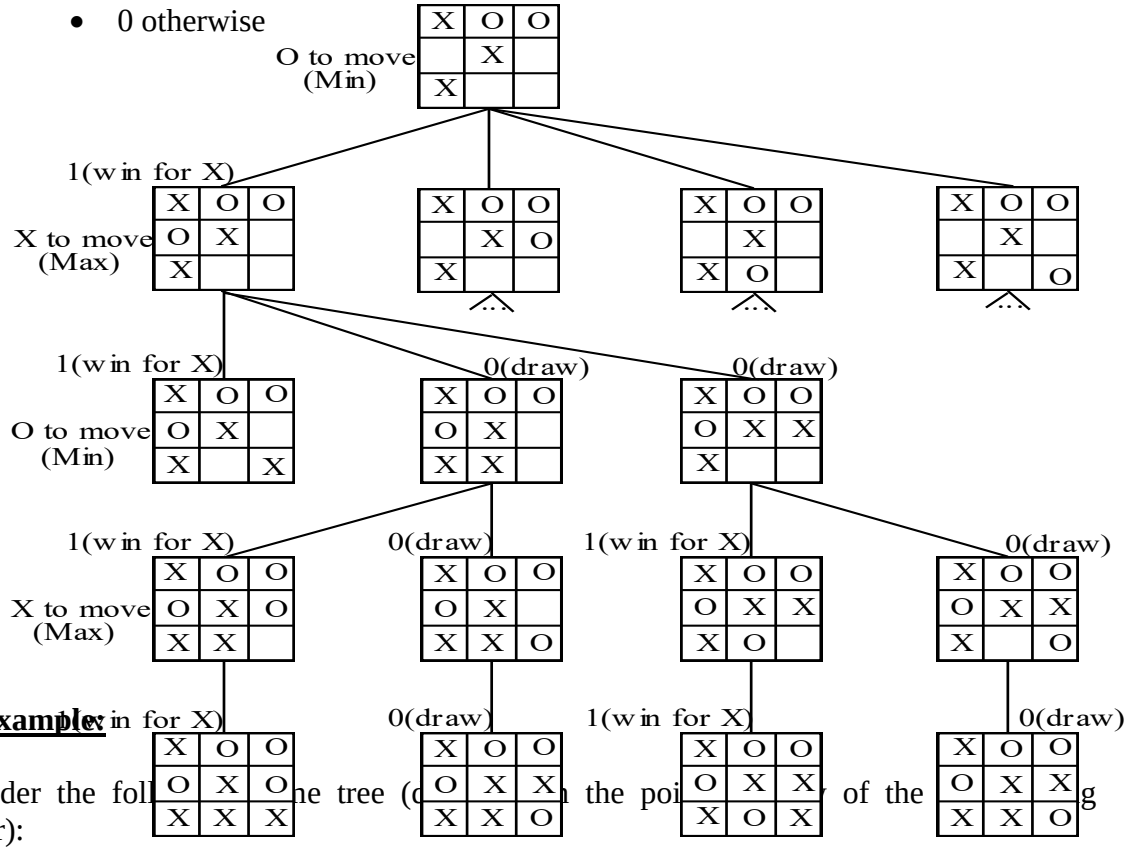
Given the values of the terminal nodes (win for X (1), loss for X (-1), or draw (0)), the values of the non-terminal nodes are computed as follows:

- 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).

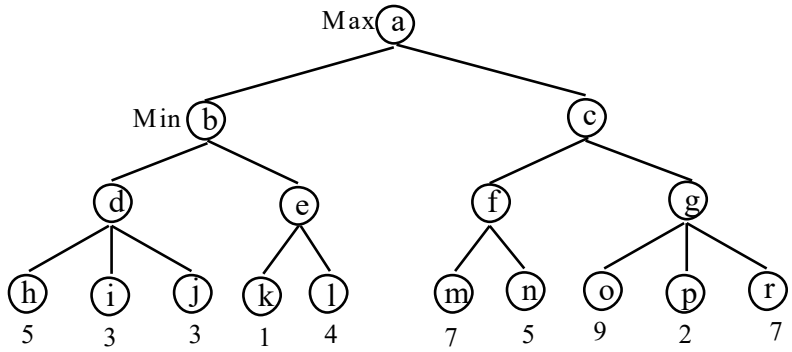
Figure below shows how the values of the nodes of the search tree are computed from the values of the leaves of the tree. The values of the leaves of the tree are given by the rules of the game:

- 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



An Example:

Consider the following tree (corresponding to 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:

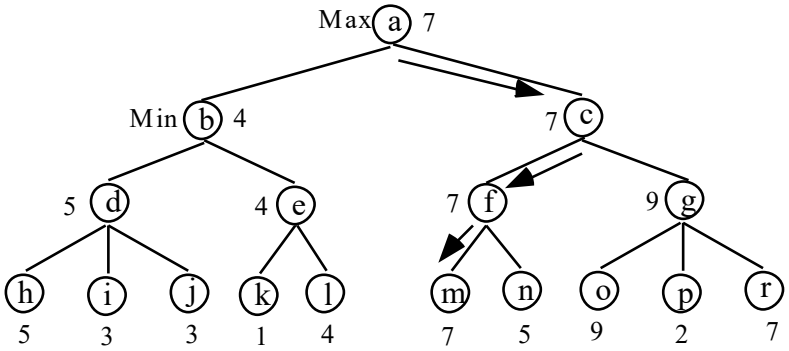


Figure 3.16: The mini-max path for the game tree

Alpha-Beta Pruning:

The problem with minimax search is that the number if game states it has examine is exponential in the number of moves. Unfortunately, we can't eliminate the exponent, but we can effectively cut it in half. The idea is to compute the correct minimax decision without looking at every node in the game tree, which is the concept behind pruning. Here idea is to eliminate large parts of the tree from consideration. The particular technique for pruning that we will discuss here is "**Alpha-Beta Pruning**". When this approach is applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision. Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub-trees rather than just leaves.

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

Alpha: is the value of the best (i.e. highest value) choice we have found so far at any choice point along the path for MAX.

Beta: is the value of the best (i.e. lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha-beta search updates the values of alpha and beta as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current alpha or beta for MAX or MIN respectively.

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.

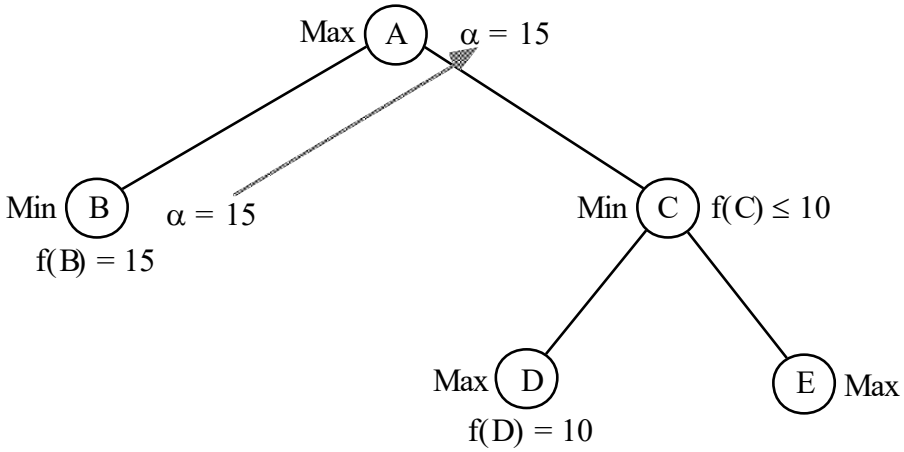


Figure 3.17: Illustration of the alpha cut-off.

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:

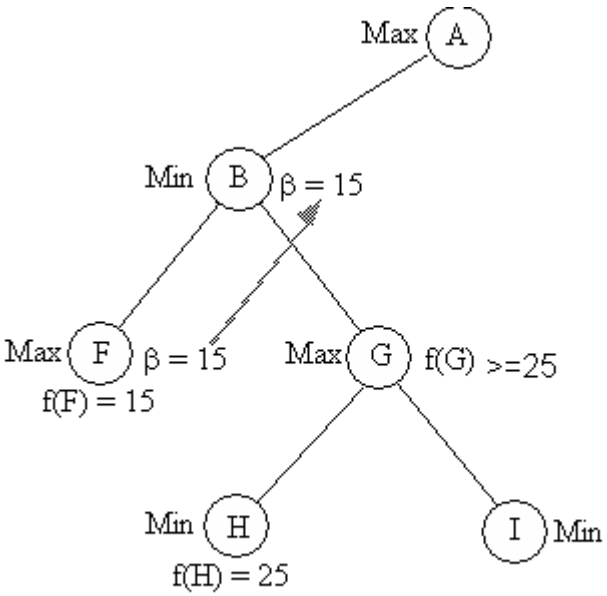


Figure 3.18: Illustration of the beta cut-off.

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.