

[Unit 3: Problem Solving by Searching]
Artificial Intelligence (MDS 556)
Master in Data Science

Problem Solving:

Problem solving, particularly in artificial intelligence, may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution. Problem-solving methods divide into special purpose and general purpose. A special-purpose method is tailor-made for a particular problem and often exploits very specific features of the situation in which the problem is embedded. In contrast, a general-purpose method is applicable to a wide variety of problems. One general-purpose technique used in AI is means-end analysis—a step-by-step, or incremental, reduction of the difference between the current state and the final goal.

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.

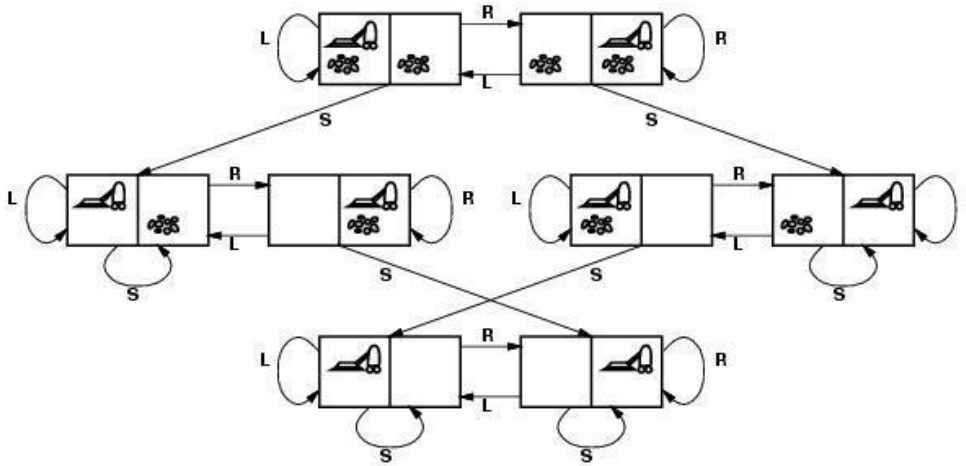
A solution is a sequence of actions from initial to goal state. Optimal solution has the lowest path cost.

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:



There are two jugs, a 4-gallon one and a 3-gallon one. Neither jug has any measuring markers on it. There is a pump that can be used to fill the jugs with water.

How can you get exactly n (0, 1, 2, 3, 4) gallons of water into one of the two jugs ?

Solution Paradigm:

- build a simple production system for solving this problem.
- represent the problem by using the state space paradigm.

State = (x, y) ; where: x represents the number of gallons in the 4-gallon jug; y represents the number of gallons in the 3-gallon jug. $x \in \{0, 1, 2, 3, 4\}$ and $y \in \{0, 1, 2, 3\}$.

The initial state represents the initial content of the two jugs.

For instance, it may be $(2, 3)$, meaning that the 4-gallon jug contains 2 gallons of water and the 3-gallon jug contains three gallons of water.

The goal state is the desired content of the two jugs.

The left hand side of a production rule indicates the state in which the rule is applicable and the right hand side indicates the state resulting after the application of the rule.

For instance;

(x, y) such that $x < 4 \rightarrow (4, y)$ represents the production

If the 4-gallon jug is not full then fill it from the pump.

The rule base contains the following production rules:

1. (x, y) such that $x < 4 \rightarrow (4, y)$; Fill the 4-gallon jug from pump
2. (x, y) such that $y < 3 \rightarrow (x, 3)$; Fill the 3-gallon jug from pump
3. (x, y) such that $x > 0 \rightarrow (0, y)$; Empty the 4-gallon jug on the ground
4. (x, y) such that $y > 0 \rightarrow (x, 0)$; Empty the 3-gallon jug on the ground
5. (x, y) such that $x + y \geq 4, x < 4, y > 0 \rightarrow (4, y - (4 - x))$
; Completely fill the 4-gallon jug from the 3-gallon jug
6. (x, y) such that $x + y \geq 3, x > 0, y < 3 \rightarrow (x - (3 - y), 3)$
; Completely fill the 3-gallon jug from the 4-gallon jug
7. (x, y) such that $x + y \leq 4, y > 0 \rightarrow (x+y, 0)$
; Empty the 3-gallon jug into the 4-gallon jug
8. (x, y) such that $x + y \leq 3, x > 0 \rightarrow (0, x + y)$
; Empty the 4-gallon jug into the 3-gallon jug

The short term memory contains the current state (x, y) .

Let us consider the initial situation $(0, 0)$ and the goal situation $(n, 2)$

short term memory : $(0, 0)$

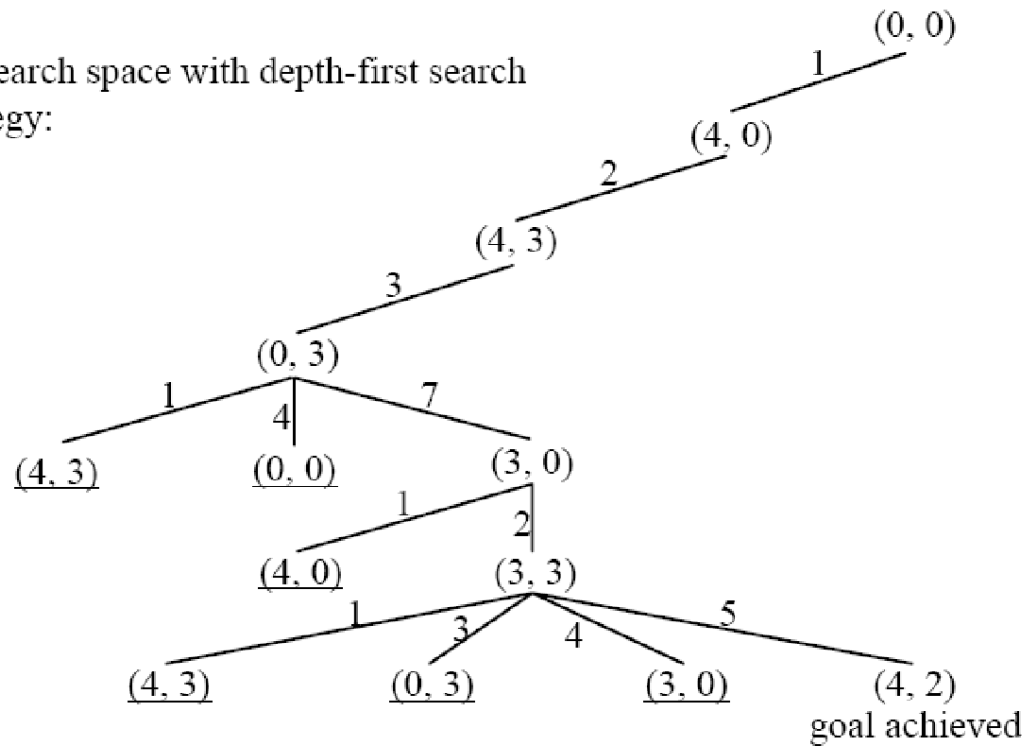
- | | | |
|------------------------------------|----------------------------------------|-------------------|
| 1. Match: 1, 2. | 2. Conflict resolution: select rule 2. | 3. Apply the rule |
| short term memory becomes $(0, 3)$ | | |
| 1. Match: 1, 4, 7 | 2. Conflict resolution: select rule 7 | 3. Apply the rule |
| short term memory becomes $(3, 0)$ | | |
| 1. Match: 1, 2, 3, 6, 8 | 2. Conflict resolution: select rule 2 | 3. Apply the rule |
| short term memory becomes $(3, 3)$ | | |
| 1. Match: 1, 3, 4, 5 | 2. Conflict resolution: select rule 5 | 3. Apply the rule |
| short term memory becomes $(4, 2)$ | | Goal achieved |

The sequence of the applied rules:

- Fill the 3-gallon jug from pump
- Empty the 3-gallon jug into the 4-gallon jug
- Fill the 3-gallon jug from pump
- Fill the 4-gallon jug from the 3-gallon jug

The Water Jug Problem: State Space Representation

the search space with depth-first search strategy:

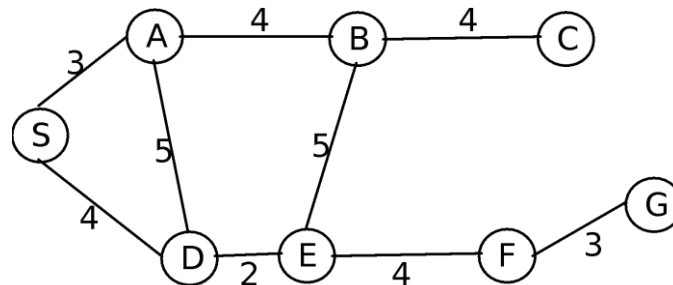


Problem Solving by Searching

A search problem

Consider a problem represented with a state space graph. The nodes represent states and the links represent action from one state to another. The number associated to a link represents the cost of links.

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



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

- VLSI layout
- Path planning and 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

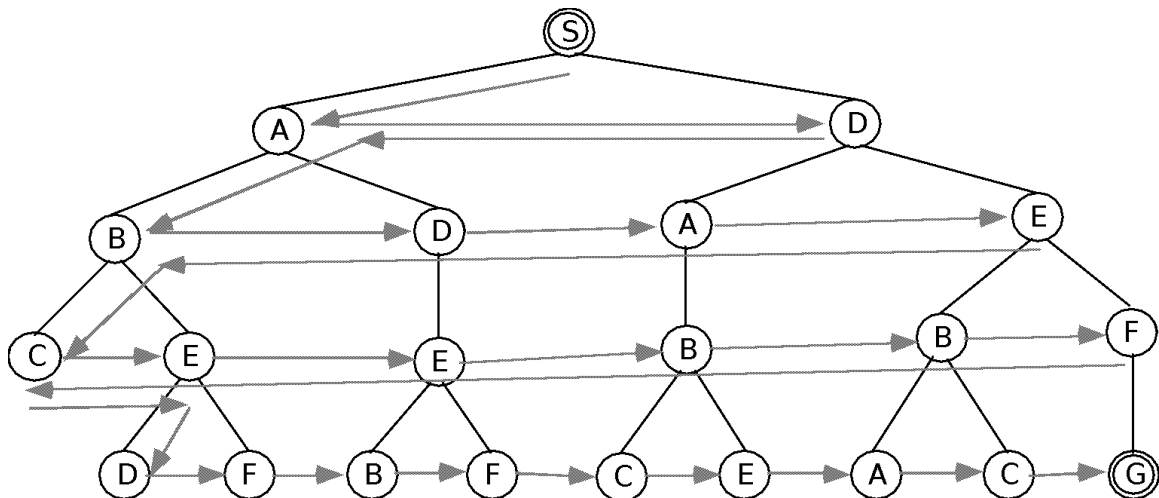
m -- maximum length of any path in the space

Breadth First Search

All nodes are expanded 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:
 $b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$

Space complexity:

- Each node that is generated must remain in memory
- Total no. of nodes in memory:
 $1 + b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$

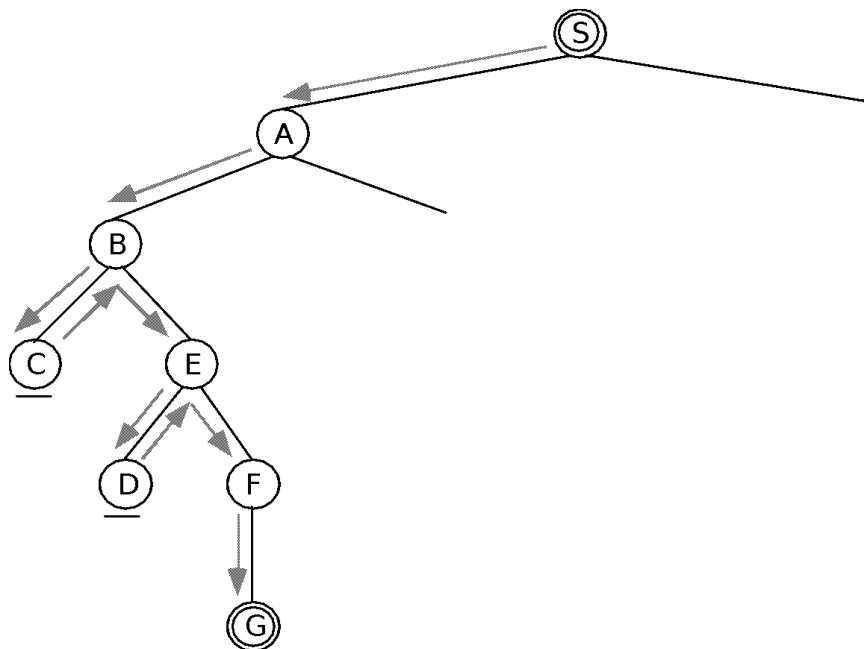
Optimal (i.e., admissible):

- 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

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:
 $b + b^2 + b^3 + \dots + b^m = O(b^m)$

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:
 $1 + b + b + b + \dots + b \text{ } m \text{ times} = O(bm)$

Optimal (i.e., admissible):

- DFS expand deepest node first, if expands entire left 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.

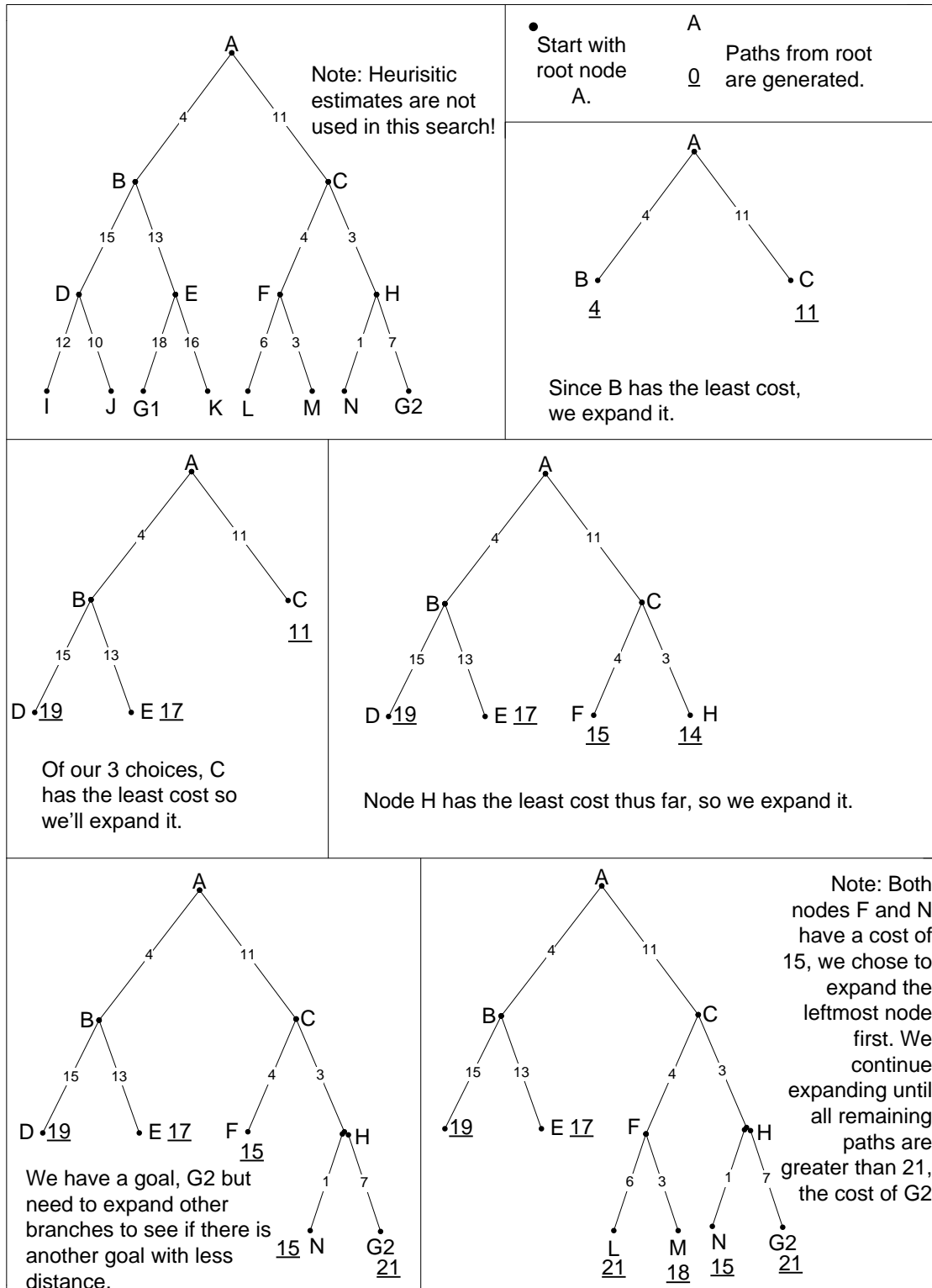
Uniform Cost Search:

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.

Consider an example:



Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d . Instead, let C^* be the cost of the optimal solution,⁷ and assume that every action costs at least ϵ . Then the algorithm's worst-case time and space complexity is $O(b^{1+\lceil C^*/\epsilon \rceil})$, which can be much greater than b^d . This is because uniform-cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, $b^{1+\lceil C^*/\epsilon \rceil}$ is just b^{d+1} . When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

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.

Completeness: If $l < d$ then incompleteness results.

Optimal: If $l > d$ then not optimal.

Time complexity: $O(b^l)$

Space complexity: $O(b \cdot l)$

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.

It 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.

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.

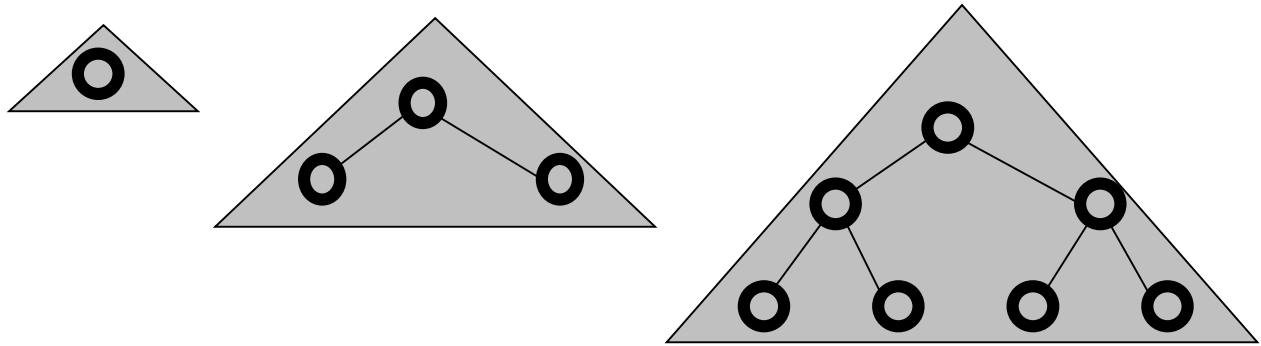
On the other hand, breadth-first search can be guaranteed to terminate if there is a winning state to be found and will always find the 'quickest' solution (in terms of how many steps need to be taken from the root node). It is, however, a very expensive method in terms of memory usage.

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).

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).

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:

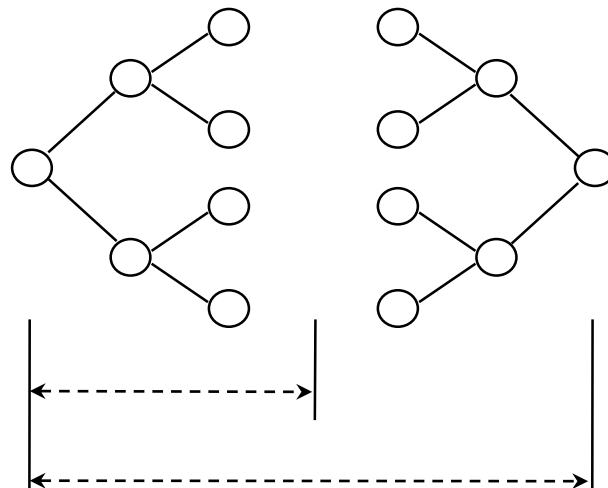
$$d.b + (d-1).b^2 + (d-2).b^3 + \dots + 1.b^d = O(b^d)$$
- Space complexity:
 - It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
 - Total no. of nodes in memory:

$$1 + b + b + b + \dots + b \text{ } d \text{ times} = O(bd)$$
- 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.

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. It checks 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 with Bidirectional Search:

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

Drawbacks of uniform 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

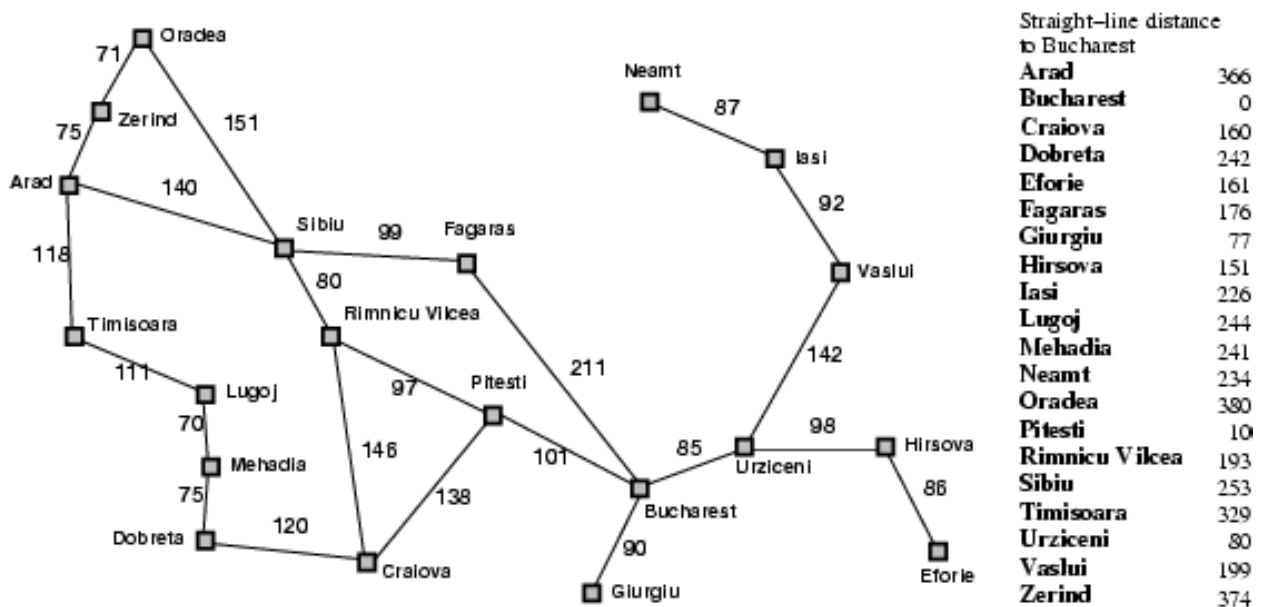
The best-first search part of the name means that it uses an evaluation function to select which node is to be expanded next. The node with the lowest evaluation is selected for expansion because that is the *best* node, since it supposedly has the closest path to the goal (if the heuristic is good). 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.

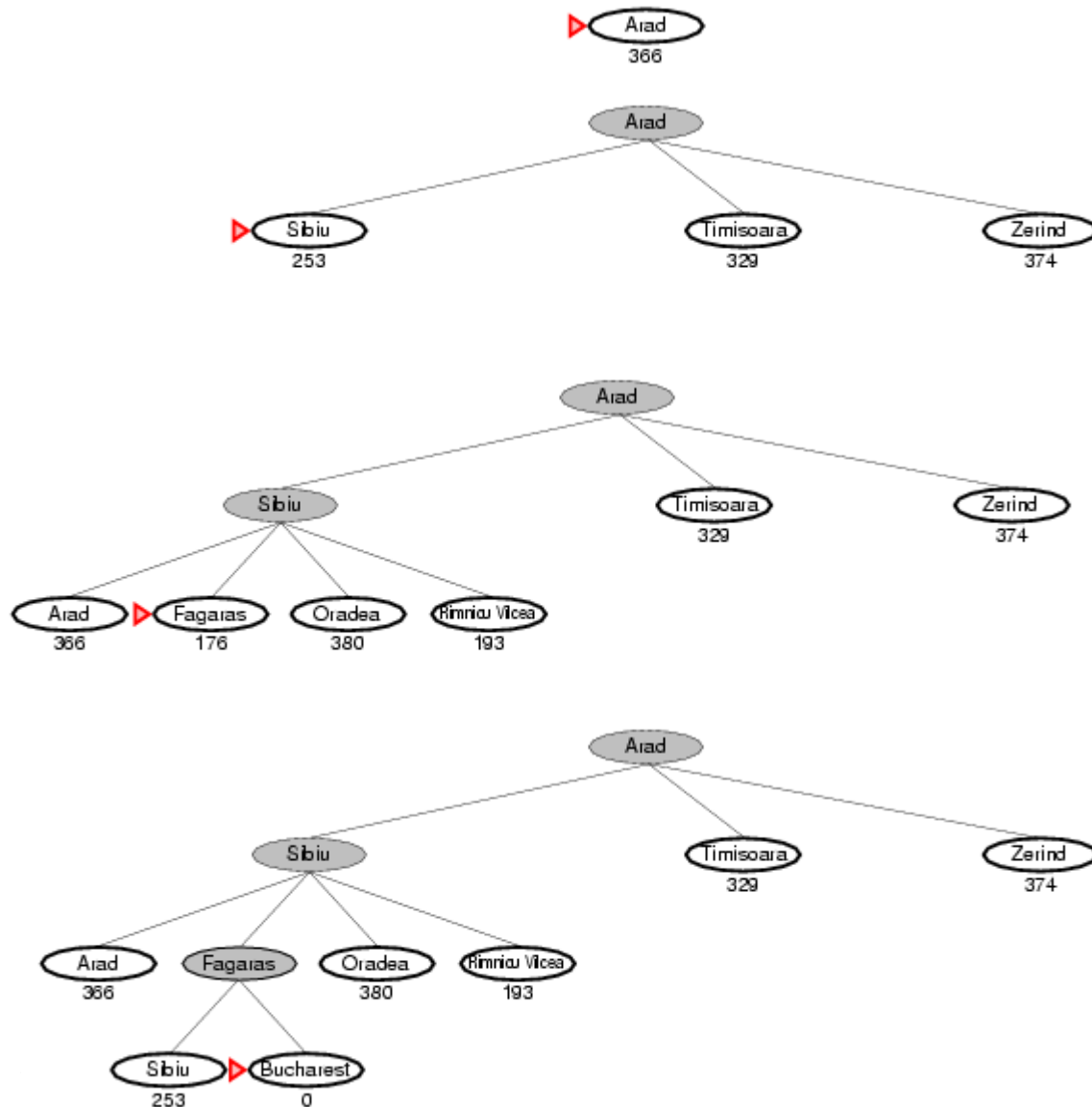
e.g., $h_{SLD}(n)$ = straight-line distance from n to goal

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.



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 $f(n)$.*

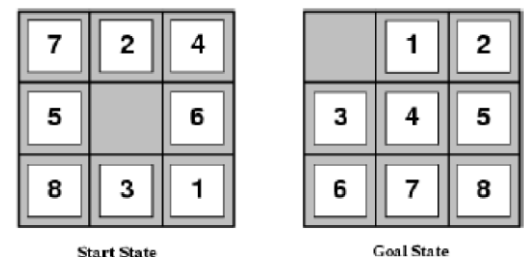
It is obvious that the SLD heuristic function is admissible as we can never find a shorter distance between any two towns.

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 (not diagonal).

$h_1(S) = ?$ 8

$h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

$h_n(S) = \max\{h_1(S), h_2(S)\} = 18$

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 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,

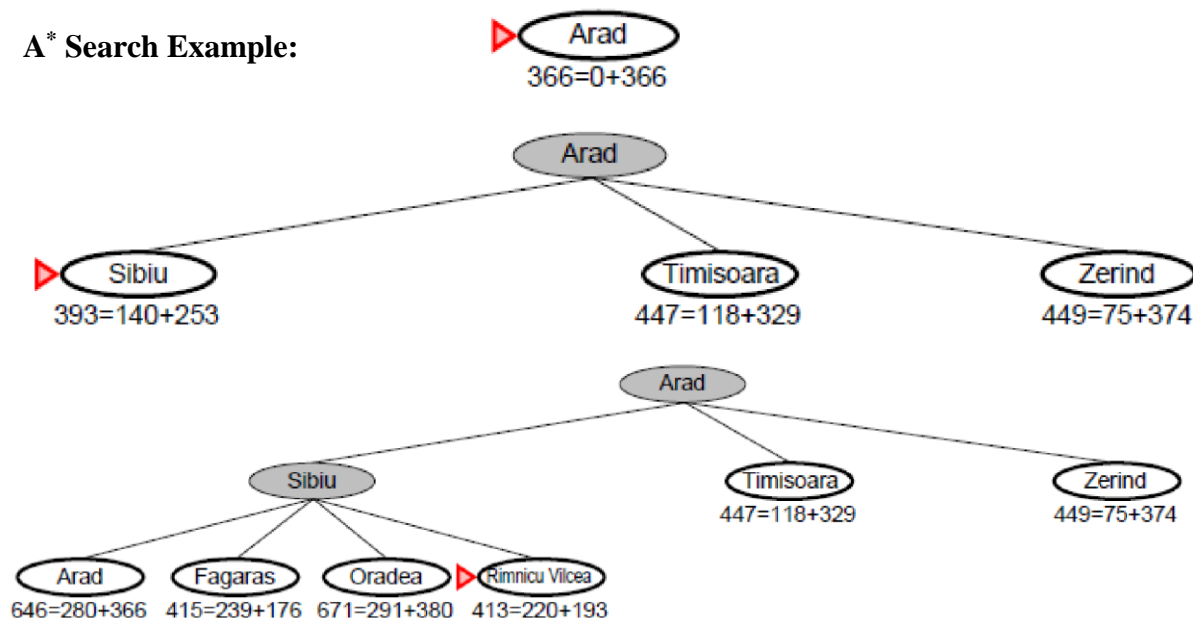
$g(n)$ = cost so far to reach n from root

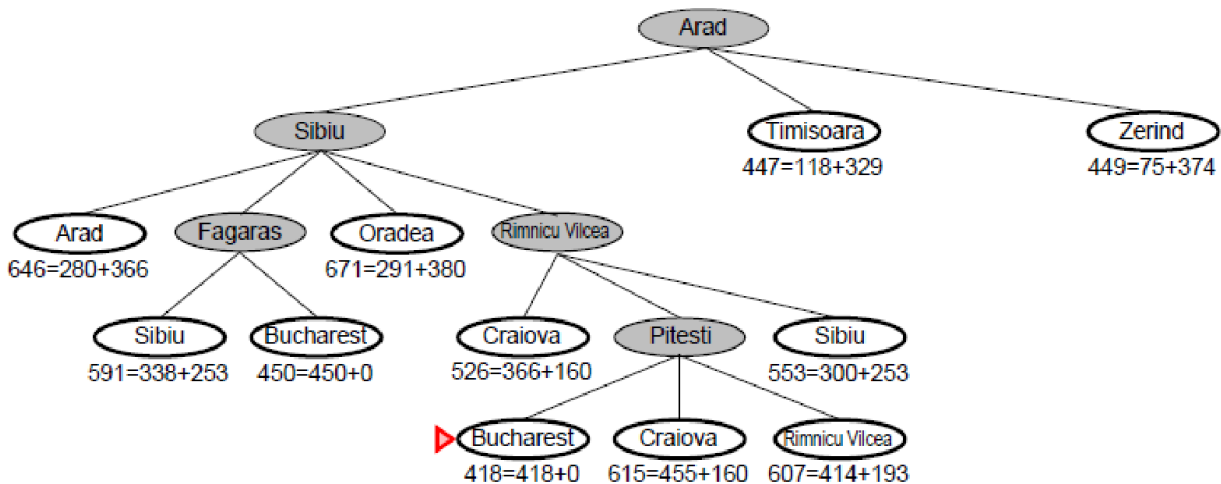
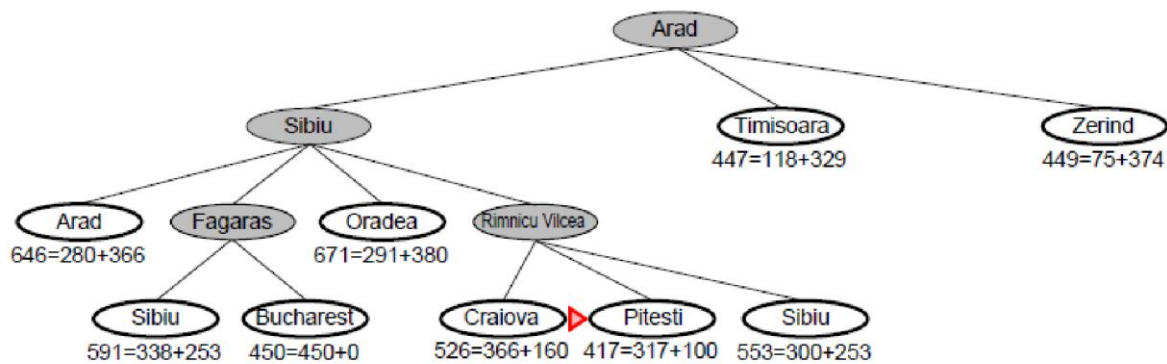
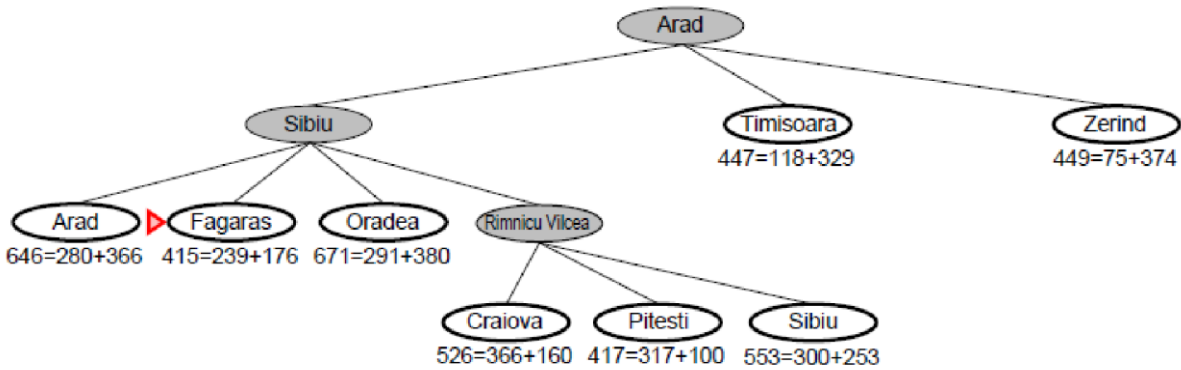
$h(n)$ = estimated cost to goal from n

$f(n)$ = 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 $0 \leq h(n) \leq h^*(n)$, 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 G_2 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.

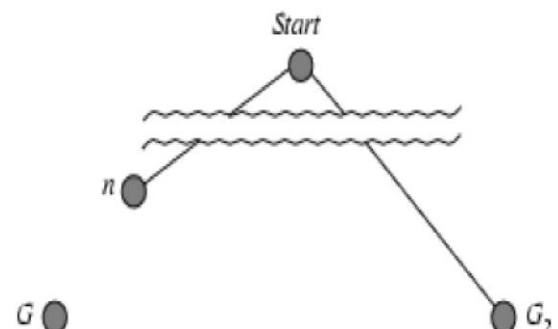
$$\begin{aligned} f(G_2) &= h(G_2) + g(G_2) \\ f(G_2) &= g(G_2), \text{ since } h(G_2) = 0 \\ f(G_2) &> C^* \quad \dots\dots\dots(1) \end{aligned}$$

Again, since $h(n)$ is admissible, It does not overestimates the cost of completing the solution path.

$$f(n) = g(n) + h(n) \leq C^* \quad \dots\dots\dots(2)$$

Now from (1) and (2)

$$f(n) \leq C^* < f(G_2)$$



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,

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

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

A heuristic is consistent if

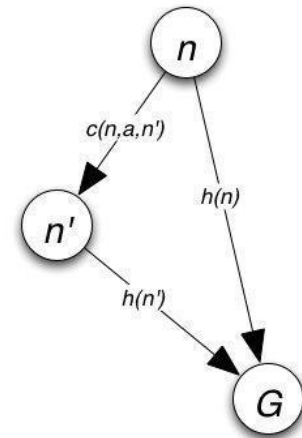
$$h(n) \leq C(n, a, n') + h(n') \dots\dots\dots(2)$$

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.



One more example: Maze Traversal (for A^* Search)

Problem: To get from square **A3** to square **E2**, one step at a time, avoiding obstacles (black squares).

Operators: (in order)

- **go_left(n)**
- **go_down(n)**
- **go_right(n)**

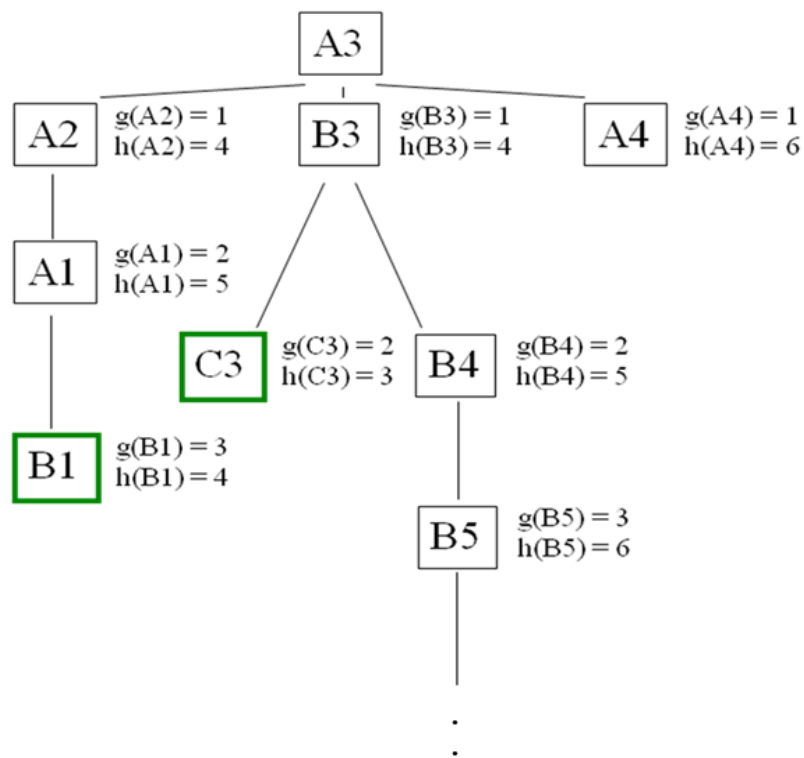
Each operator costs 1.

Heuristic: Manhattan distance

Start Position: A3

Goal: E2

A					
B					
C					
D					
E					
	1	2	3	4	5



A	A1	A2		A4	
B	B1		B3	B4	B5
C			C3		
D					
E					
	1	2	3	4	5

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.

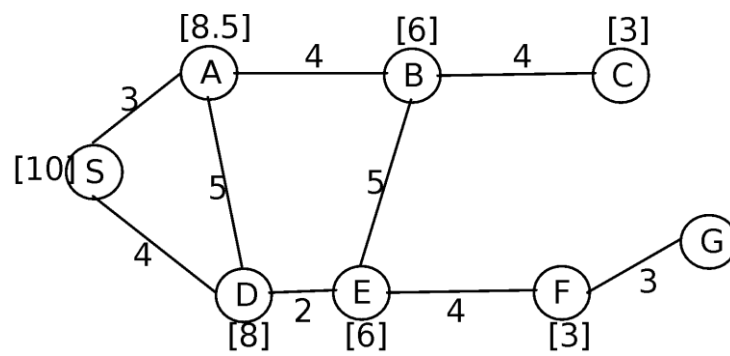
Hill climbing terminates when there are no successors of the current state which are better than the current state itself.

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.

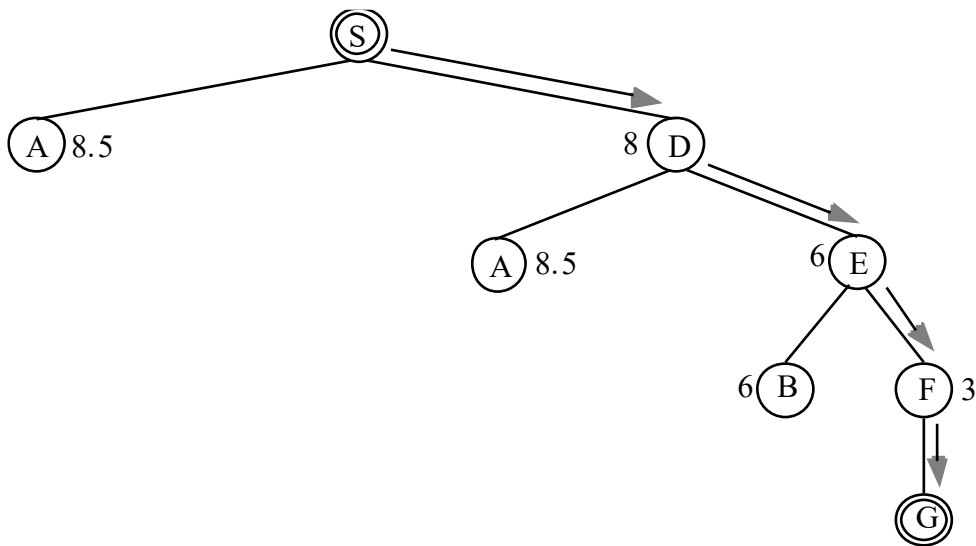
Simple Hill climbing : It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node.

Steepest-Ascent Hill climbing : It first examines all the neighboring nodes and then selects the node closest to the solution state as next node.

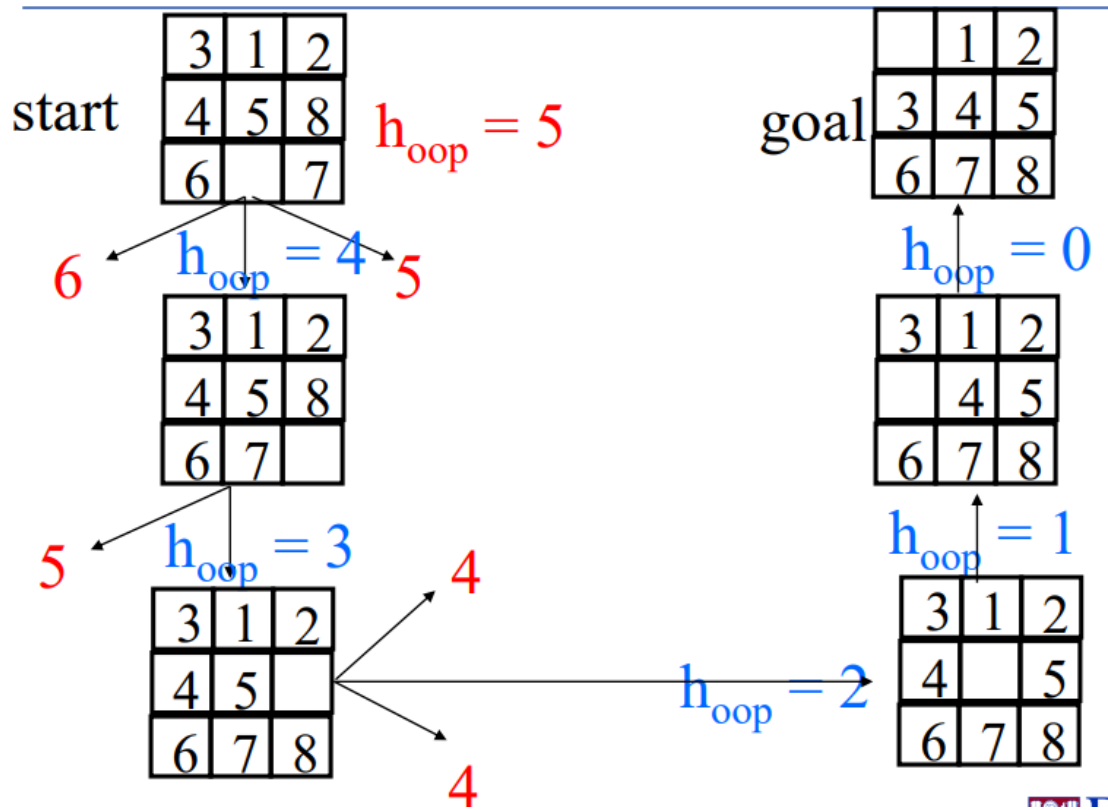
For instance, consider that the most promising successor of a node is the one that has the shortest straight-line distance to the goal node G. In figure below, the straight line distances between each city and goal G is indicated in square brackets, i.e. the heuristic.



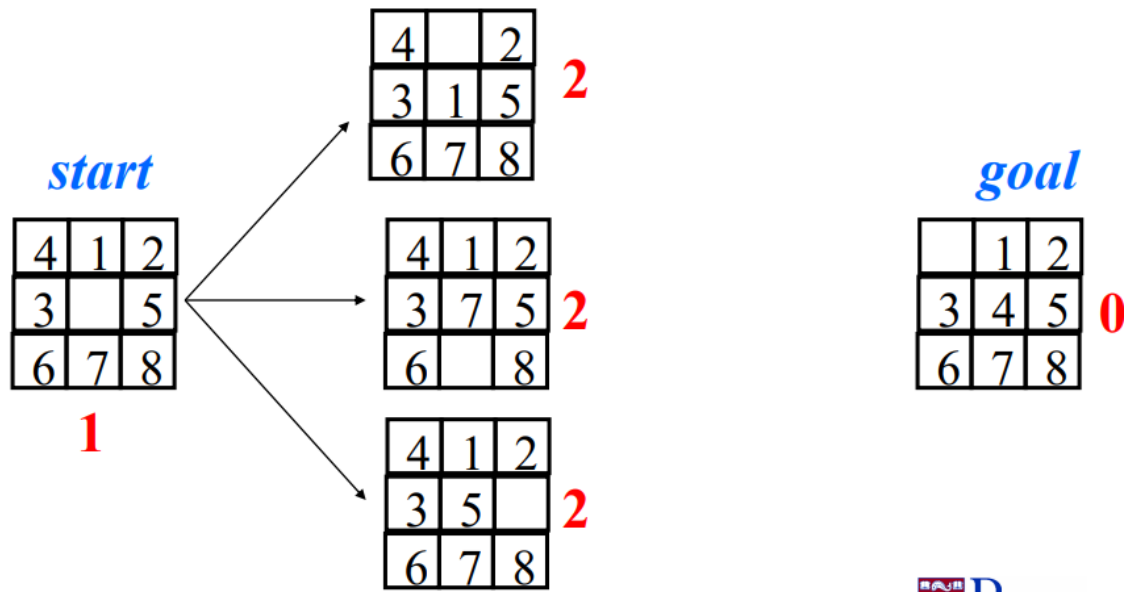
The hill climbing search from S to G proceeds as follows:



Hill climbing example I (*minimizing h*)



Toy Example of a local "maximum"



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

$$P = e^{(-c/t)}$$

Where

c	=	the change in the evaluation function i.e. if there is a bad move from A to B, then $c=f(B)-f(A)$
t	=	the current value

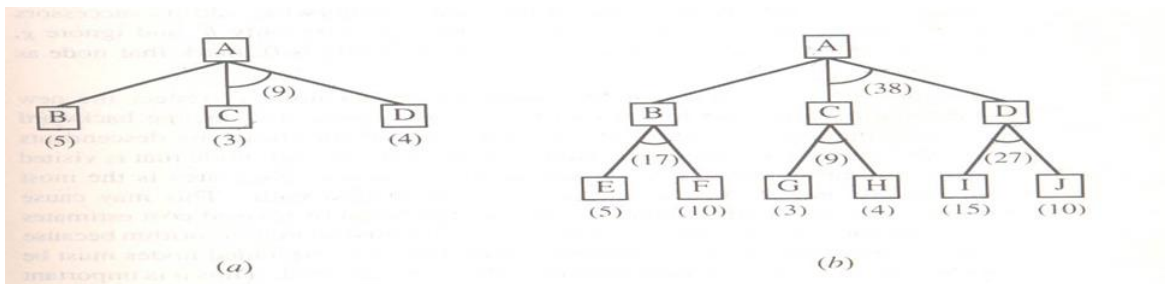
The probability of accepting a worse state is a function of both the current value and the change in the cost function. The most common way of implementing an SA algorithm is to implement hill climbing with an accept function and modify it for SA

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.**

AO* Search

AO* search Algorithm is based on problem decomposition. When a problem can be divided or decomposed into a set of sub problems, where each sub problem can be solved separately and for each subproblem, sub solution is evaluated and a combination of these sub solutions will be a whole solution, AND OR graphs or AND OR trees are used for representing this solution.

- This method generates arc which is called as AND-OR arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for an arc to point to a solution. AND-OR graph is used to represent various kind of complex problem solutions.



In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D. The numbers at each node represent the value of f' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation (i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components. With the available information till now, it appears that C is the most promising node to expand since its $f' = 3$, the lowest but going through B would be better since to use C we must also use D' and the cost would be $9(3+4+1+1)$. Through B it would be $6(5+1)$.

Thus the choice of the next node to expand depends not only on a value but also on whether that node is part of the current best path form the initial mode.

Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f' value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of $(17+1=18)$.

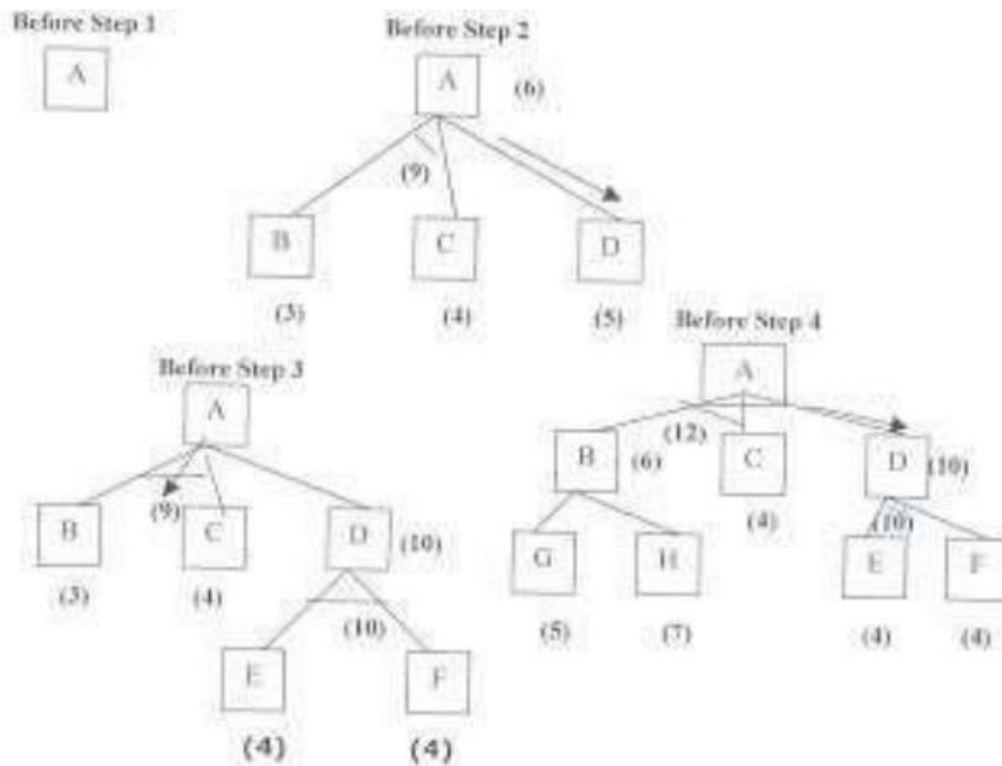
Thus to search an AND-OR graph, the following must be done;

- Traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
- Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f' (cost of the remaining distance) for each of them.

- Change the f' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

The propagation of revised cost estimation backward in the tree is not necessary in A* algorithm. This is because in AO* algorithm expanded nodes are re-examined so that the current best path can be selected.

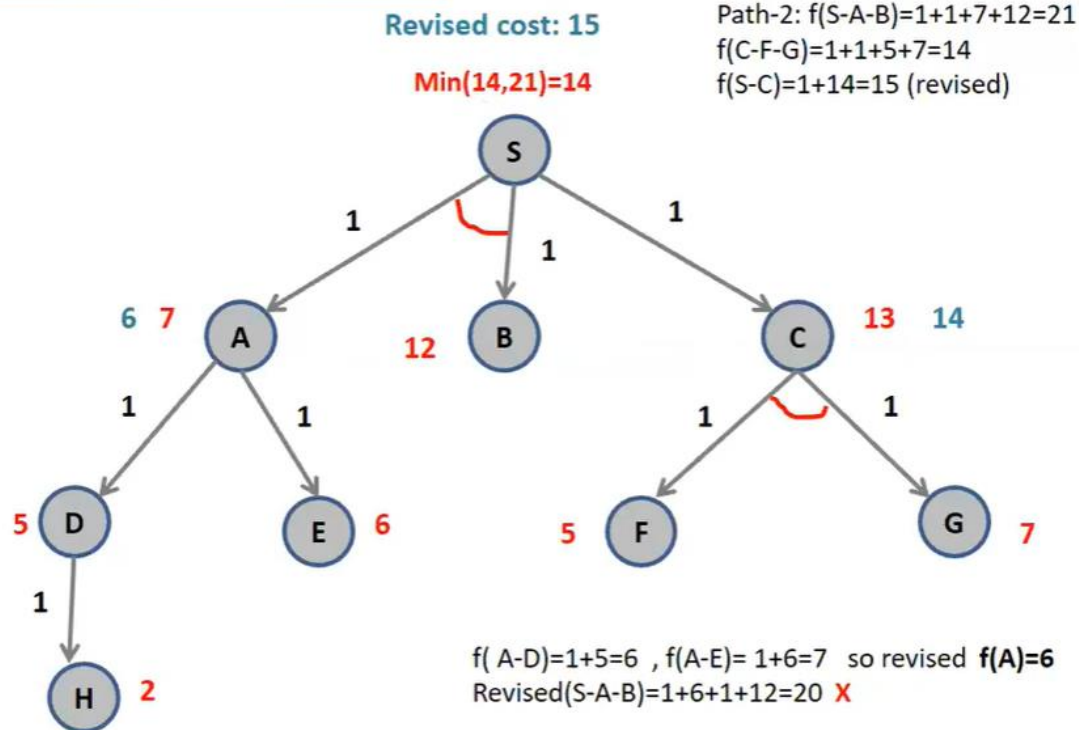
The working of AO* algorithm is illustrated in figure as follows:

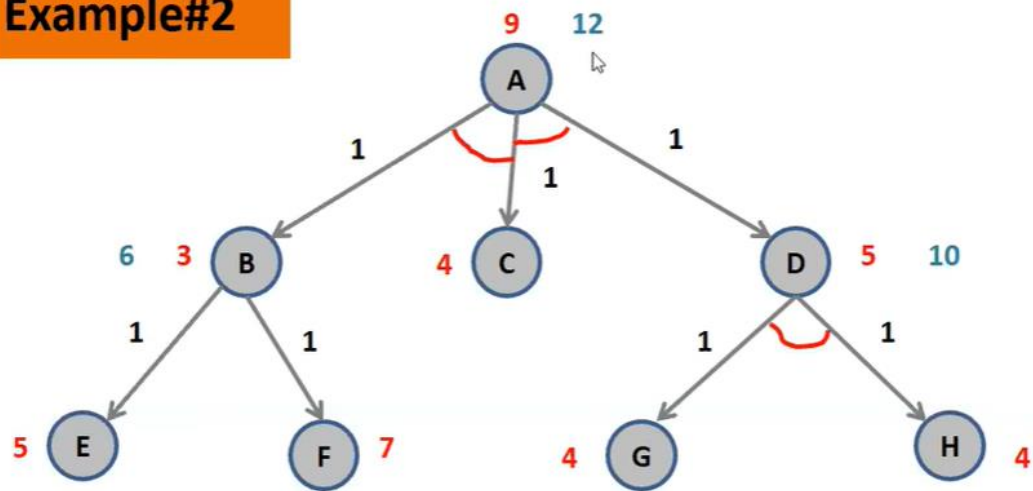


Referring the figure. The initial node is expanded and D is marked initially as promising node. D is expanded producing an AND arc E-F. f' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better. It is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution.

Algorithm:

1. Initialize the graph to start node.
2. Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved.
3. Pick any of these nodes and expand it and if it has no successors call this value futility otherwise calculate only f' for each of the successor
4. If f' is 0 then mark the node as solved
5. Change the value of f' for the newly created node to reflect its successor by back propagation.
6. Wherever possible use the most promising routes and if node is marked as solved then mark the parent node as solved.
7. If starting node is solved or value is greater than futility stop else go to step 2 and repeat.

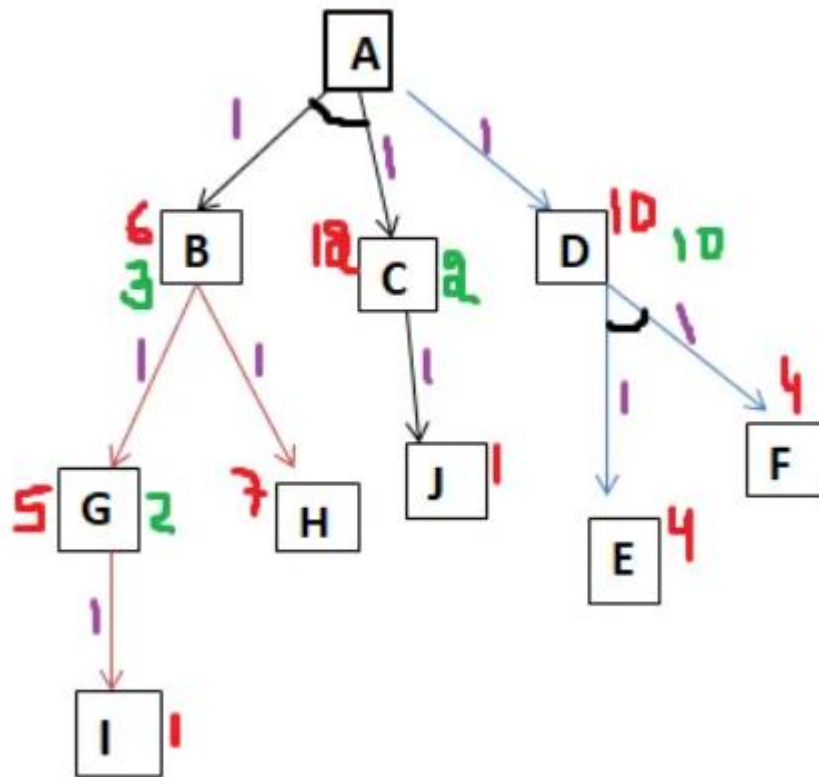
AO*: Example #1

Example#2

Path -1: $f(A-B-C) = 1+1+3+4 = 9$
 $f(B-E) = 1+5 = 6$ $f(B-F) = 1+7 = 8$
 $f(A-B-C) = 1+1+6+4 = 12$

Path-2: $f(A-C-D) = 1+1+4+5 = 11$
 $f(D-G-H) = 1+1+4+4 = 10$
 $f(A-C-D) = 1+1+4+10 = 16$

Consider a state space as below;



The AO* search algorithm always moves towards a **lower cost value**.

Basically, We will calculate the **cost function** here ($F(n) = G(n) + H(n)$)

H: heuristic/ estimated value of the nodes. and **G: actual cost** or edge value (here unit value).

Here we have taken the **edges value 1**, meaning we have to focus solely on the **heuristic value**.

1. The **Purple color** values are **edge values** (here all are same that is one).
2. The **Red color** values are **Heuristic values** for nodes.
3. The **Green color** values are **New Heuristic values** for nodes.

Procedure:

1. In the above diagram we have two ways from A to D or A to B-C (because of and condition). calculate cost to select a path
2. $F(A-D) = 1 + 10 = 11$ and $F(A-BC) = 1 + 1 + 6 + 12 = 20$
3. As we can see $F(A-D)$ is less than $F(A-BC)$ then the algorithm choose the path $F(A-D)$.

4. From D we have one choice that is **F-E**.
5. **$F(A-D-FE) = 1+1+4+4=10$**
6. Basically **10** is the cost of reaching **FE from D**. And **Heuristic value of node D** also denote the cost of reaching **FE from D**. So, the new Heuristic value of D is 10.
7. And the Cost from A-D remain same that is **11**.

Suppose we have searched this path and we have got the **Goal State**, then we will never explore the other path. (this is what AO* says but here we are going to explore other path as well to see what happen)

Let's Explore the other path:

1. In the above diagram we have two ways from **A to D** or **A to B-C** (because of and condition). calculate cost to select a path
2. **$F(A-D) = 1+10 = 11$** and **$F(A-BC) = 1 + 1 + 6 + 12 = 20$**
3. As we know the cost is more of **F(A-BC)** but let's take a look
4. Now from B we have two path G and H , let's calculate the cost
5. **$F(B-G) = 5+1=6$** and **$F(B-H) = 7 + 1 = 8$**
6. So, cost from **F(B-H)** is more than **F(B-G)** we will take the path B-G.
7. The Heuristic value from **G to I** is **1** but let's calculate the cost form **G to I**.
8. **$F(G-I) = 1 + 1 = 2$** . which is less than **Heuristic value 5**. So, the new **Heuristic value form G to I is 2**.
9. If it is a new value, then the cost from **G to B** must also have changed. Let's see the new **cost form (B to G)**
10. **$F(B-G) = 1+2=3$** . Mean the **New Heuristic value of B is 3**.
11. **But A is associated with both B and C .**
12. As we can see from the diagram **C only have one choice or one node to explore that is J**. The Heuristic value of C is 12.
13. Cost form C to **J**= **$F(C-J) = 1+1= 2$** Which is less than Heuristic value
14. Now **the New Heuristic value of C is 2**.
15. **And the New Cost from A- BC that is $F(A-BC) = 1+1+2+3 = 7$ which is less than $F(A-D)=11$.**
16. In this case Choosing path A-BC is more cost effective and good than that of A-D.

But this will only happen when the algorithm explores this path as well. But according to the algorithm, algorithm will not accelerate this path (**here we have just did it to see how the other path can also be correct**).

But it is not the case in all the cases that it will happen in some cases that the algorithm will get optimal solution.

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.

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

- **S0**: The **initial state**, which specifies how the game is set up at the start.
- **PLAYER(s)**: Defines which player has the move in a state.
- **ACTIONS(s)**: Returns the set of legal moves in a state.
- **RESULT(s, a)**: The **transition model**, which defines the result of a move.
- **TERMINAL-TEST(s)**: A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- **UTILITY(s, p)**: A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state *s* for a player *p*. In chess, the outcome is a win, loss, or draw, with values +1, 0, or -1

The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the nodes are game states and the edges are moves.

A 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.

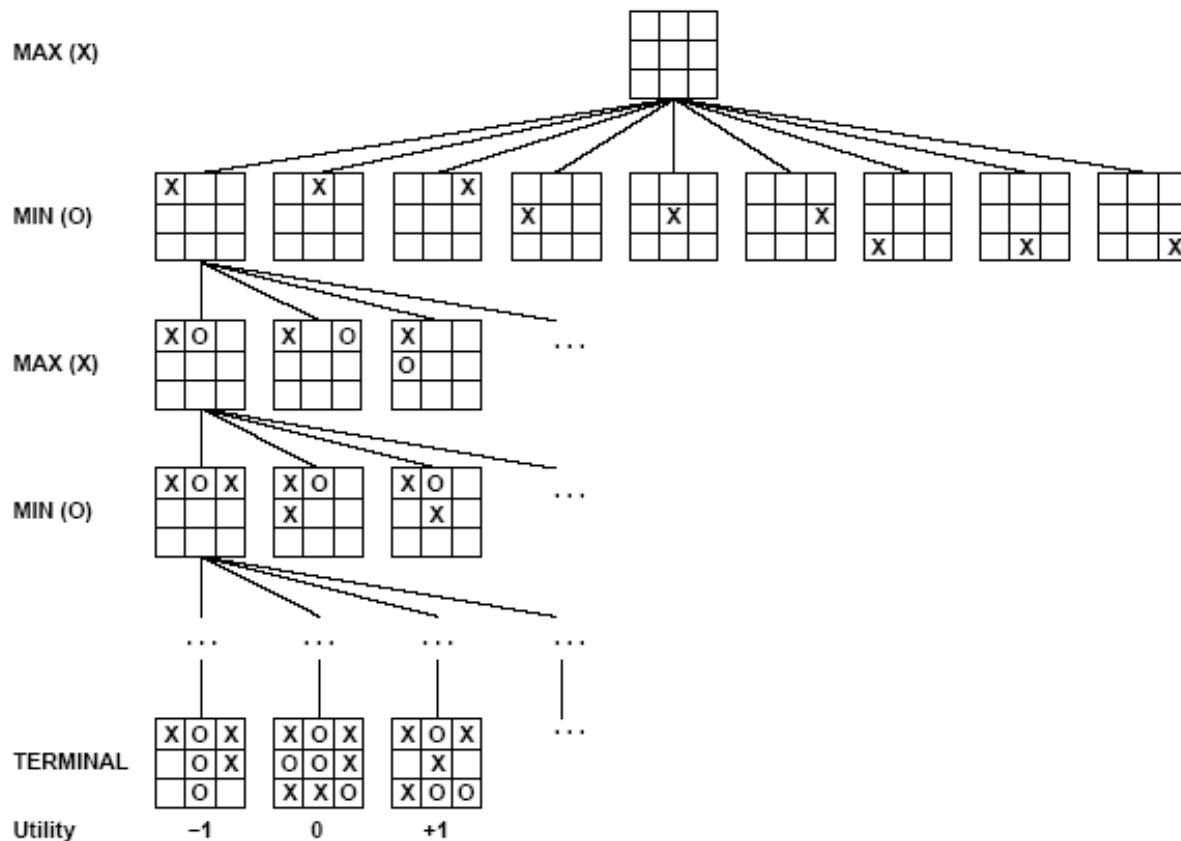


Fig: Partial game tree for Tic-Tac-Toe

The Minimax Algorithm:

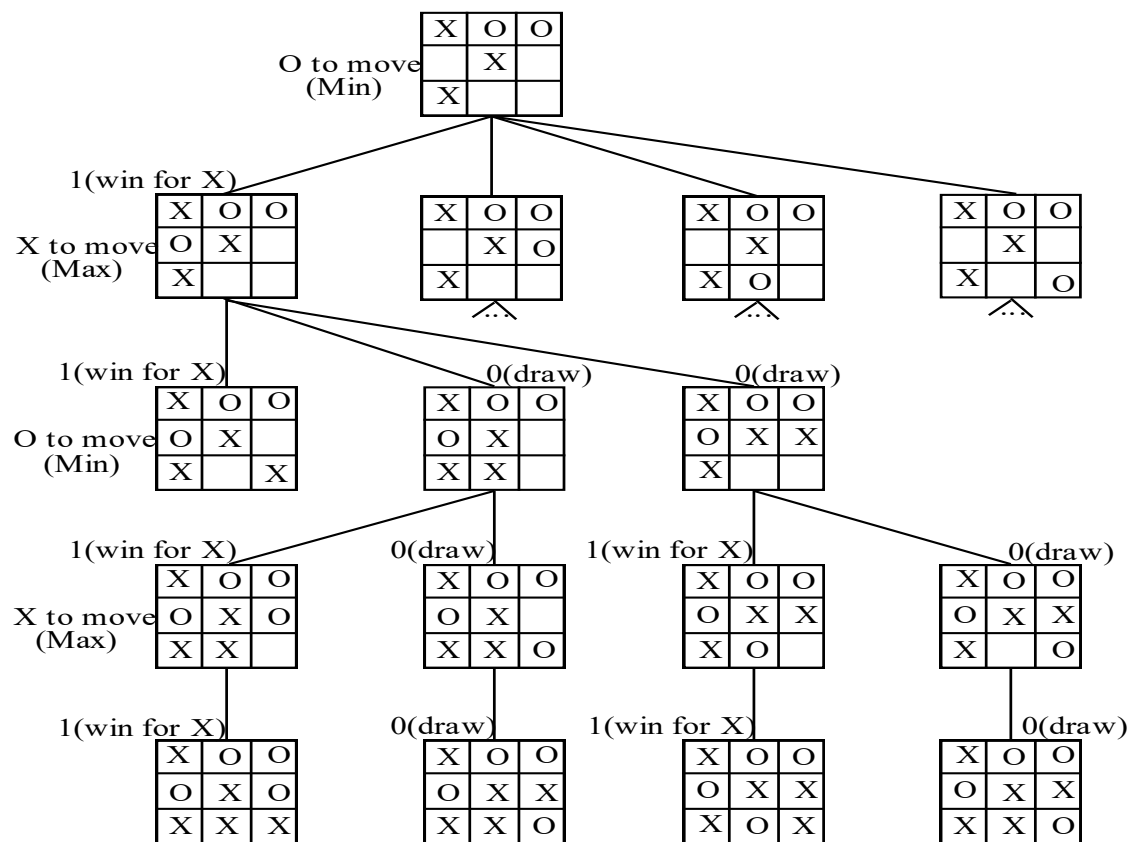
Let us assign the following values for the game: 1 for win by X, 0 for draw, -1 for loss by X.

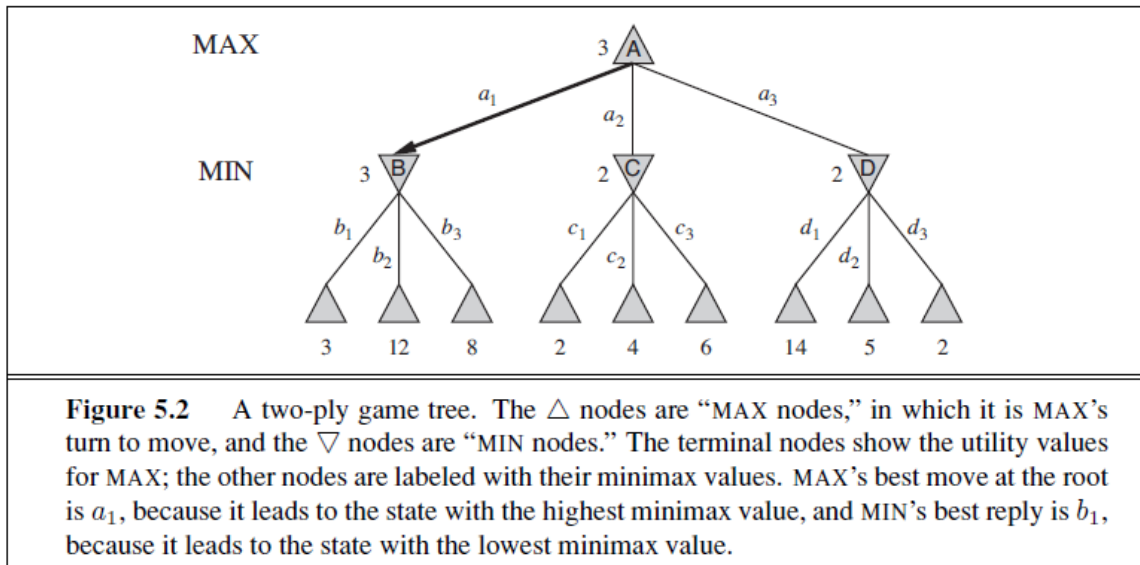
Given the values of the terminal nodes (win for X (1), loss for X (-1), or draw (0)), the values of the non-terminal nodes are computed as follows:

- 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).

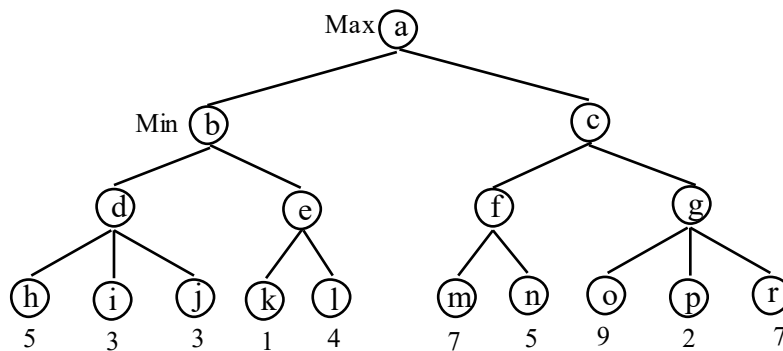
Figure below shows how the values of the nodes of the search tree are computed from the values of the leaves of the tree. The values of the leaves of the tree are given by the rules of the game:

- 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



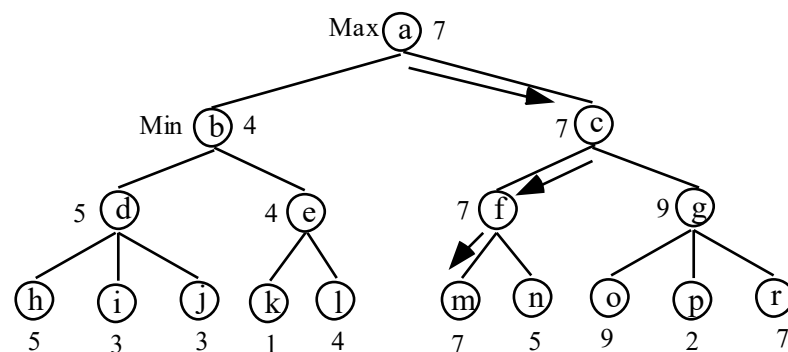
An Example:

Consider the following 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:

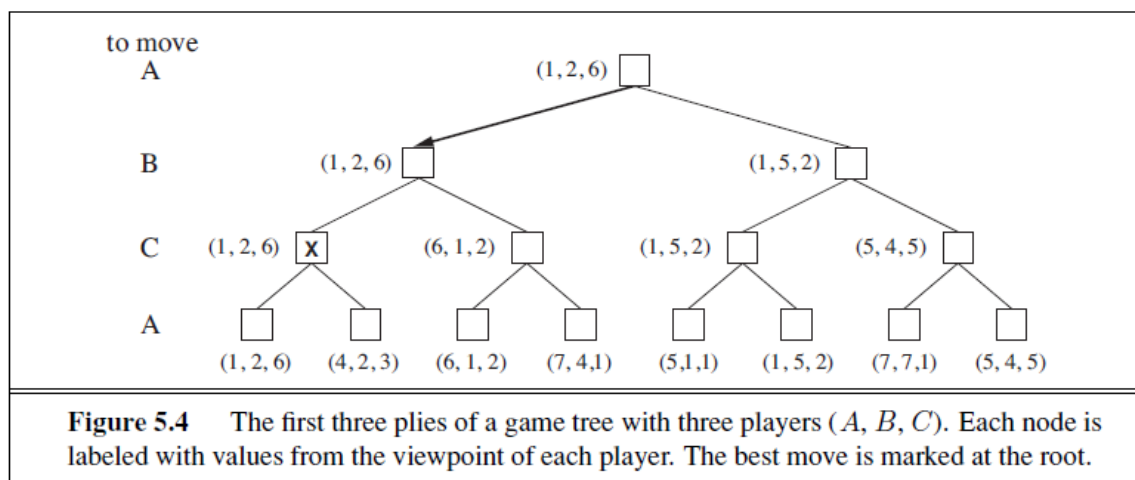


Multi Player Game:

Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.

First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players A, B, and C, a vector $\langle v_A, v_B, v_C \rangle$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure 5.4. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$.



Alpha-Beta Pruning:

The problem with minimax search is that the number of game states it has to 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.

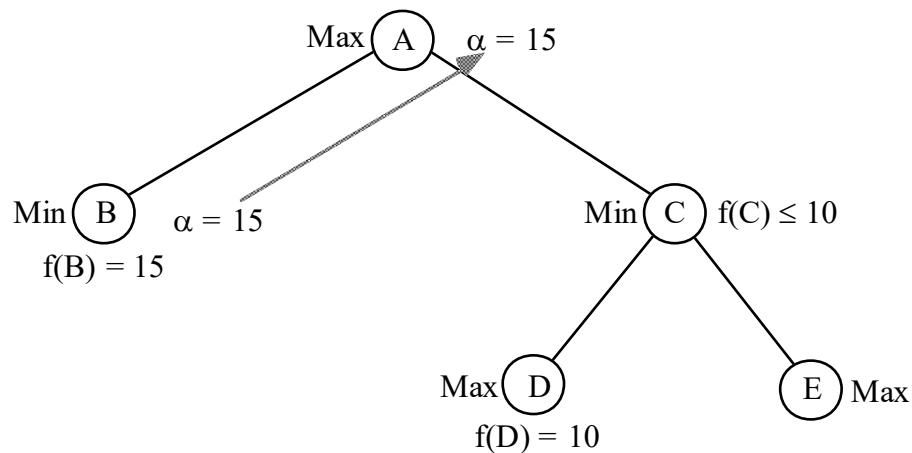


Illustration of the alpha cut-off.

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 \square 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:

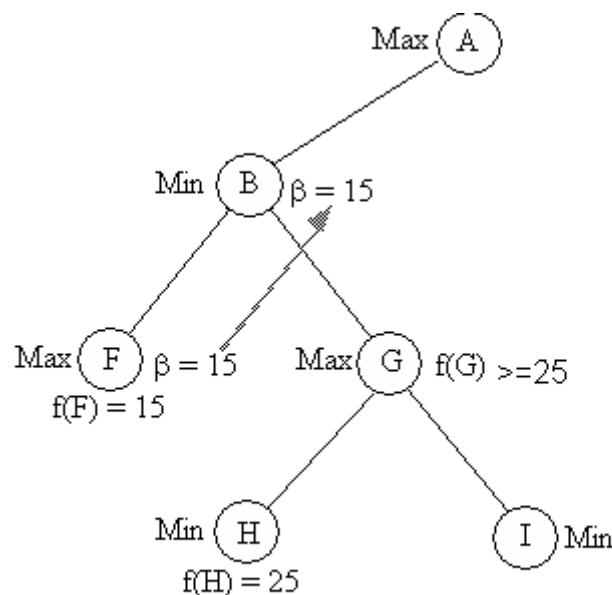


Illustration of the beta cut-off.

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 \square 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 \square , 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 \square (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.

Constraint Satisfaction Problem:

A **Constraint Satisfaction Problem** is characterized by:

- a *set of variables* $\{x_1, x_2, \dots, x_n\}$,
- for each variable x_i a *domain* D_i with the possible values for that variable, and
- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognizing function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n , a value in D_i for x_i so that all constraints are satisfied. Means that, we must find a value for each of the variables that satisfies all of the constraints.

A CS problem can easily be stated as a sentence in first order logic, of the form:

$$(\text{exist } x_1) \dots (\text{exist } x_n) (D_1(x_1) \& \dots D_n(x_n) \Rightarrow C_1 \dots C_m)$$

A CS problem is usually represented as an undirected graph, called *Constraint Graph* where the nodes are the variables and the edges are the binary constraints. Unary constraints can

be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyperarcs. In the following we restrict our attention to the case of unary and binary constraints.

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a domain of values, and C is a set of constraints. Every constraint is in turn a pair $\langle t, R \rangle$, where t is a tuple of variables and R is a set of tuples of values; all these tuples having the same number of elements; as a result R is a relation. An evaluation of the variables is a function from variables to values, $v : X \rightarrow D$. Such an evaluation satisfies a constraint $\langle (x_1, \dots, x_n), R \rangle$ if $(v(x_1), \dots, v(x_n)) \in R$. A solution is an evaluation that satisfies all constraints.

Constraints

- A constraint is a relation between a **local** collection of variables.
- The constraint restricts the values that these variables can simultaneously have.
- For example, **all-diff(X1, X2, X3)**. This constraint says that X1, X2, and X3 must take on different values. Say that $\{1,2,3\}$ is the set of values for each of these variables then:
X1=1, X2=2, X3=3 OK X1=1, X2=1, X3=3 NO

The constraints are the key component in expressing a problem as a CSP.

- The constraints are determined by how the variables and the set of values are chosen.
- Each constraint consists of;
 - A set of variables it is over.
 - A specification of the sets of assignments to those variables that satisfy the constraint.
- The idea is that we break the problem up into a set of distinct conditions each of which have to be satisfied for the problem to be solved.

Example: In N-Queens: Place N queens on an N x N chess board so that queen can attack any other queen.

- No queen can attack any other queen.
- Given any two queens Q_i and Q_j they cannot attack each other.
- Now we translate each of these individual conditions into a separate constraint.
 - Q_i cannot attack $Q_j (i \neq j)$
 - Q_i is a queen to be placed in column i , Q_j is a queen to be placed in column j .
 - The value of Q_i and Q_j are the rows the queens are to be placed in.

Queens can attack each other,

1. *Vertically*, if they are in the same column---this is impossible as Q_i and Q_j are placed in different columns.
2. *Horizontally*, if they are in the same row---we need the constraint $Q_i \neq Q_j$.
3. *Along a diagonal*, they cannot be the same number of columns apart as they are rows apart: we need the constraint $|i-j| \neq |Q_i - Q_j|$ ($| |$ is absolute value)

Representing the Constraints;

Between every pair of variables (Q_i, Q_j) ($i \neq j$), we have a constraint C_{ij} .

For each C_{ij} , an assignment of values to the variables $Q_i = A$ and $Q_j = B$, satisfies this constraint if and only if;

$$\begin{aligned} A &\neq B \\ |A - B| &\neq |i - j| \end{aligned}$$

Solutions:

- A solution to the N-Queens problem will be any assignment of values to the variables Q_1, \dots, Q_N that satisfies all of the constraints.
- Constraints can be over any collection of variables. In N-Queens we only need binary constraints---constraints over pairs of variables.

Map Coloring Problem:

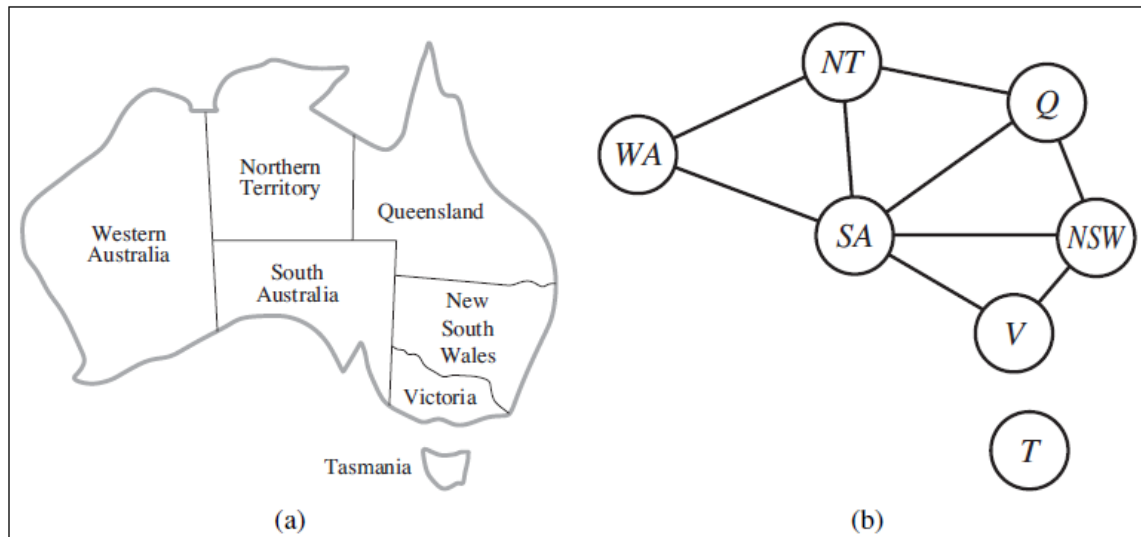


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After $WA=red$	(R)	G B	R G B	R G B	R G B	G B	R G B
After $Q=green$	(R)	B	(G)	R B	R G B	B	R G B
After $V=blue$	(R)	B	(G)	R	(B)		R G B

Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables *NT* and *SA*. After $Q = green$ is assigned, *green* is deleted from the domains of *NT*, *SA*, and *NSW*. After $V = blue$ is assigned, *blue* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

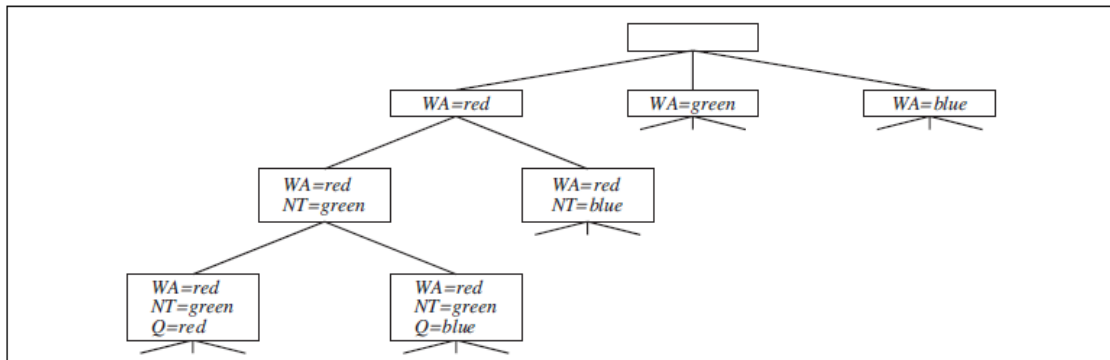


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

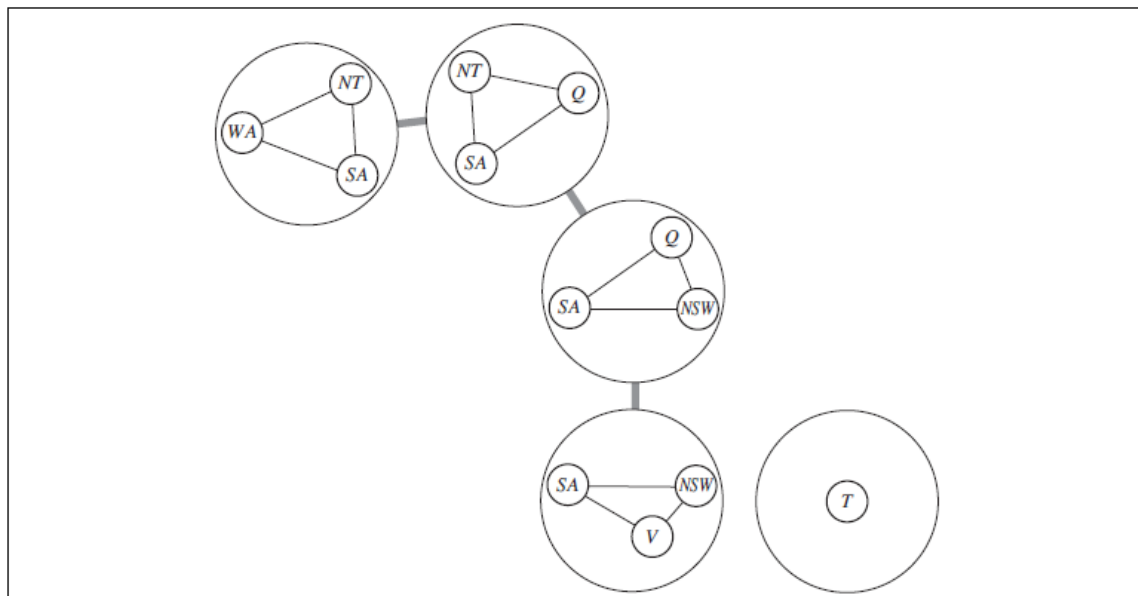


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

We solve each subproblem independently; if anyone has no solution, we know the entire problem has no solution. If we can solve all the subproblems, then we attempt to construct a global solution as follows. First, we view each subproblem as a “mega-variable” whose domain is the set of all solutions for the subproblem. For example, the leftmost subproblem in Figure 6.13 is a map-coloring problem with three variables and hence has six solutions one is $\{WA = \text{red}, SA = \text{blue}, NT = \text{green}\}$. Then, we solve the constraints connecting the subproblems, using the efficient algorithm for trees given earlier. The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables. For example, given the solution $\{WA = \text{red}, SA = \text{blue}, NT = \text{green}\}$ for the first subproblem, the only consistent solution for the next subproblem is $\{SA = \text{blue}, NT = \text{green}, Q = \text{red}\}$.