UNIT-3 Problem Solving by Searching

- **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.
- **Problem-solving agent** is a kind of goal based agent. It is an agent that tries to come up with a sequences of actions that will bring up with the environment into a desired state.
  - → An agent which can find a sequence of actions that achieves its goals when no single action will do.
  - → Intelligent agents are supposed to maximize their performance measure. Achieving this is sometimes simplified if the agent can adopt a **goal** and aim at satisfying it.
- **Searching** is the process of looking for such a sequence, involving systematic exploration of alternative actions.

General steps in problem solving:

1. **Goal formulation**
   - – Goals help to organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.
   - – A goal is considered to be a set of world states—exactly those states in which the goal is satisfied.
   - – **Goal formulation** is based on the current situation and the agent's performance measure
   - – It is the first step in problem solving.
   - – What are the successful world states?

2. **Problem formulation**
   - The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.
   - **Problem formulation** is the process of deciding what actions and states to consider, given a goal.
   - The solution to any problem is a fixed sequence of actions.

3. **Search Solution**
   - The process of looking for a sequence of actions that reaches the goal is called **search**.
   - A search algorithm takes a problem as input and returns a solution in the form of an action sequence.
   - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence

4. **Execution**
   - Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Well-defined problems and solutions

- A problem can be defined formally by five components:
  - **The initial state:** State from which the agent starts in.
  - **The actions:** A description of the possible **actions** available to the agent.
    - Given a particular state **s**, **ACTIONS(s)** returns the set of actions that can be executed in **s**.

  - **The transition model:** A description of what each action does.
    - Specified by a function **RESULT(s, a)** that returns the state that results from **SUCCESSOR** doing action **a** in state **s.**

- **The goal test:** which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.
- **The path cost:** A function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.
  - The cost of a path can be described as the sum of the costs of the individual actions along the path.
  - The step cost of taking action **a** in state **s** to reach state **g** is denoted by **c(s,a, g)**

➢ A **solution** to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the **path cost function**, and **an optimal solution has the lowest path cost among all solutions**

State Space representation

- Together, the initial state, actions, and transition model of a problem implicitly define the **state space** of the problem.
- **State space** is the set of all states reachable from the initial state by any sequence of actions.
- The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions
- A path in the state space is a sequence of states connected by a sequence of actions
- 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

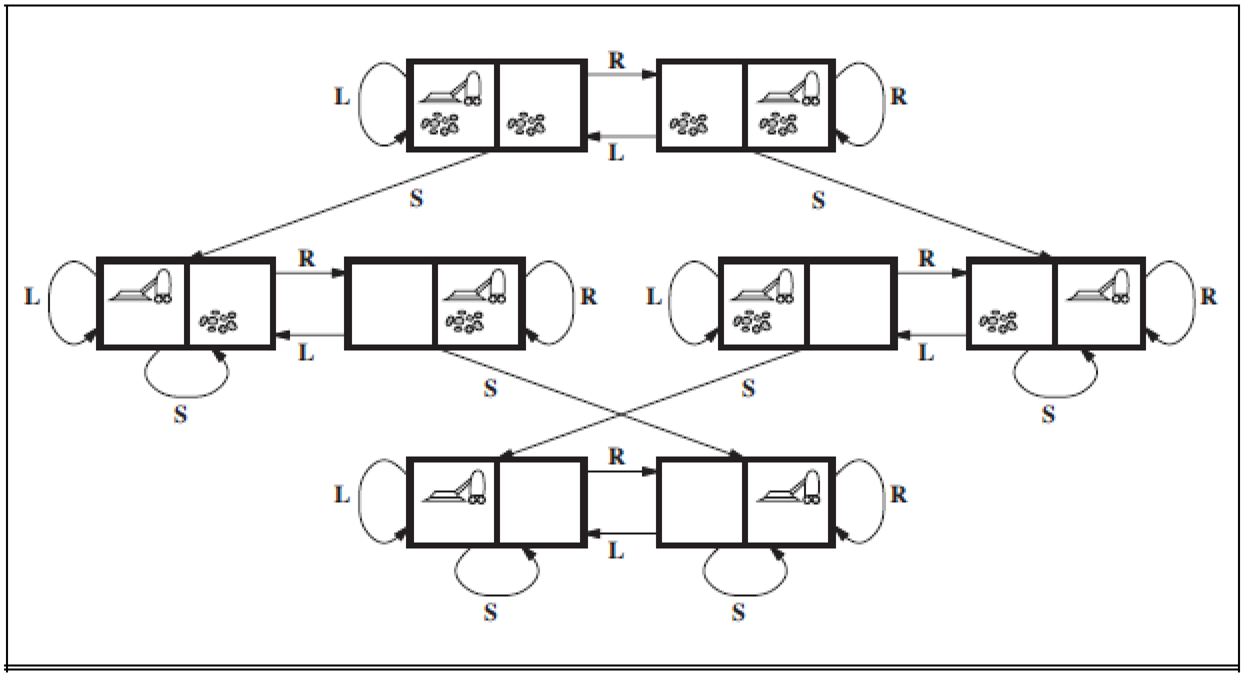


Figure: The state space for the vacuum world. Links denote actions: L = Left, R = Right, S = Suck.

Problem formulation

- Formulation of a problem can be done by representing the problem in terms of the initial state, actions, transition model, goal test, and path cost.
- Such formulation seems reasonable, but it is still a model—an abstract mathematical description—and not the real thing
- The choice of a good abstraction involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

Toy problems vs. Real-World problems

- A toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.
- A real-world problem is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

Formulating toy problem

Example: vacuum world

The vacuum world can be formulated as a problem as follows:

States: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ states.

Initial state: Any state can be designated as the initial state.

Actions: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.

Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in above figure.

Goal test: This checks whether all the squares are clean.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

Other problem: 8 puzzle, 8 queen etc

Formulating Real -World problem

Consider the airline travel problems that must be solved by a travel-planning Web site:

States: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.

Initial state: This is specified by the user's query.

Actions: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.

Transition model: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.

Goal test: Are we at the final destination specified by the user?

Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

Other problems: Travelling salesperson problem, VLSI layout, Robot navigation, automatic assembly sequencing etc.

Solving Problems by Searching

- Having formulated some problems, we now need to solve them.
- A solution is an action sequence, so search algorithms work by considering various possible action sequences.
- The possible action sequences starting at the initial state form a **search tree** with the **initial state at the root; the branches are actions** and the **nodes correspond to states** in the **state space** of the problem

<u>Example: finding a route from Arad to Bucharest (see the map below)</u>

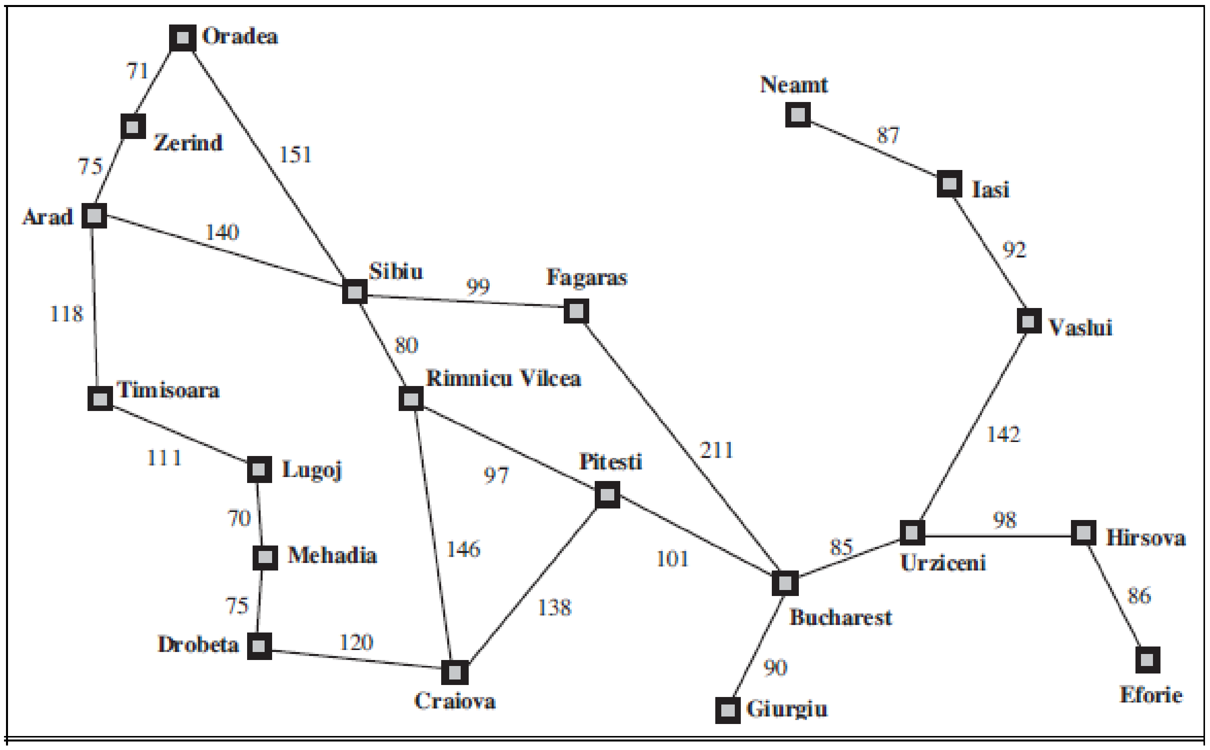


Figure: A simplified road map of part of Romania.

→ The root node of the tree corresponds to the initial state, In(Arad).

→ The first step is to test whether this is a goal state. (Clearly it is not, but it is important to check so that we can solve trick problems like "starting in Arad, get to Arad.")

→ Then we need to consider taking various actions. We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states. In this case, we add three branches from the parent node In(Arad) leading to three new child nodes: In(Sibiu), In(Timisoara), and In(Zerind).

→ Now we must choose which of these three possibilities to consider further.

→ This is the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution

→ Suppose we choose Sibiu first. We check to see whether it is a goal state (it is not) and then expand it to get In(Arad), In(Fagaras), In(Oradea), and In(RimnicuVilcea). We can then choose any of these four or go back and choose Timisoara or Zerind. Each of these six nodes is a **leaf node**, that is, a node with no children in the tree.

→ The set of all leaf nodes available for expansion at any given point is called the **frontier**.(also called as **open list**)

→ The process of expanding nodes on the **frontier** continues until either a solution is found or there are no more states to expand.

→ Search algorithms all share the basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**

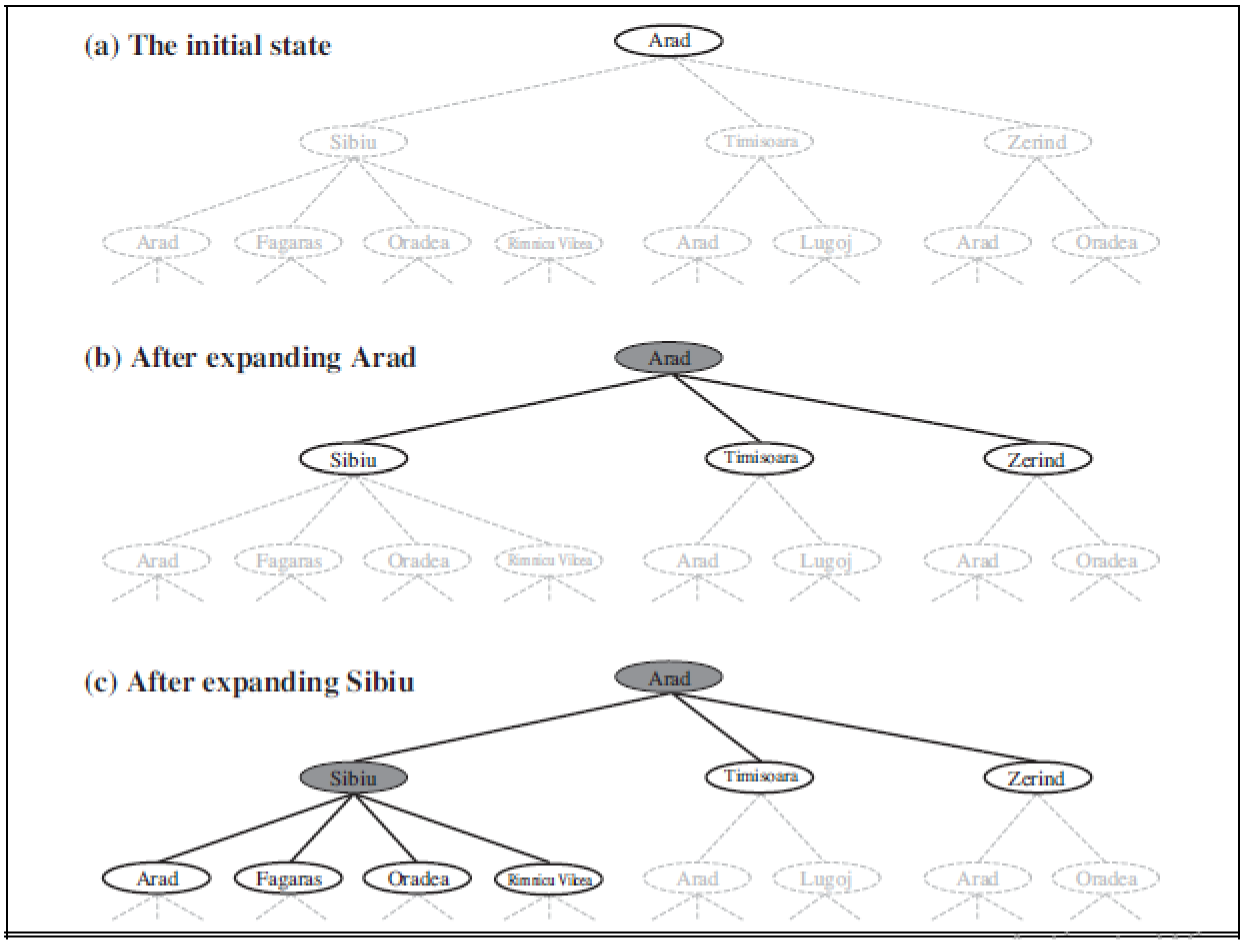


Figure: Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- **n.STATE**: the state in the state space to which the node corresponds;
- **n.PARENT**: the node in the search tree that generated this node;
- **n.ACTION**: the action that was applied to the parent to generate the node;
- **n.PATH-COST**: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

Measuring problem-solving performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm's performance in four ways:

Completeness: Is the algorithm guaranteed to find a solution when there is one?

Optimality: Does the strategy find the optimal solution? For example, is it the one with minimal cost?

Time complexity: How long does it take to find a solution? Usually measured in terms of the number of nodes expanded.

Space complexity: How much memory is needed to perform the search? Usually measured in terms of maximum number of nodes in memory at a time.

Time and space complexities are measured in terms of:

b → branching factor (maximum number of successor of any node) of the search tree
d→ depth of the shallowest goal node (i.e., the number of steps along the path from the root). [depth of the least cost solution]
m → maximum length of any path in the state space

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

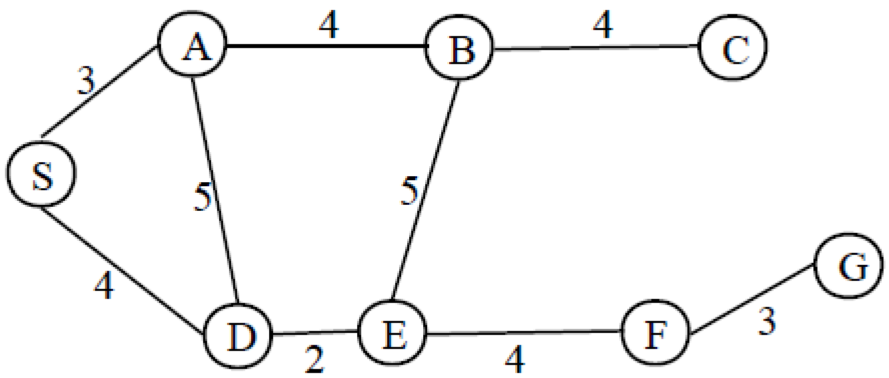


Figure: A graph representation of a map

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 search (or blind search)** methods;
- **Informed search (or heuristic 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.
  - Such strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal

state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded

- Eg: DFS,BFS, etc

- ➢ In the case of the **informed search methods**, one uses domain-dependent (heuristic) information in order to search the space more efficiently.
  - Strategies that know whether one non-goal state is "more promising" than another are called informed search or heuristic search strategies.
  - Eg: A* Search, Hill Climbing search etc.

Breadth-first search

- Breadth-first search (BFS) is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- ☞ Breadth-first search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion.
  - This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
  - There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion.
  - This algorithms, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found.
  - Thus, breadth-first search always has the shallowest path to every node on the frontier

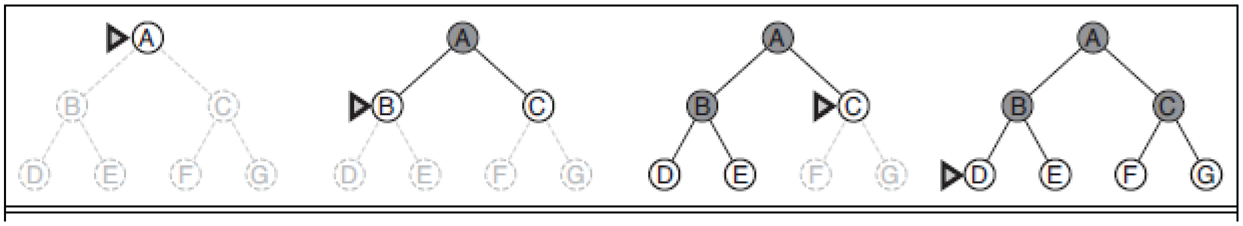


Figure Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

- ♦ Expand shallowest unexpanded node.
- ♦ Constraint: Do not generate as child node if the node is already parent to avoid more loop

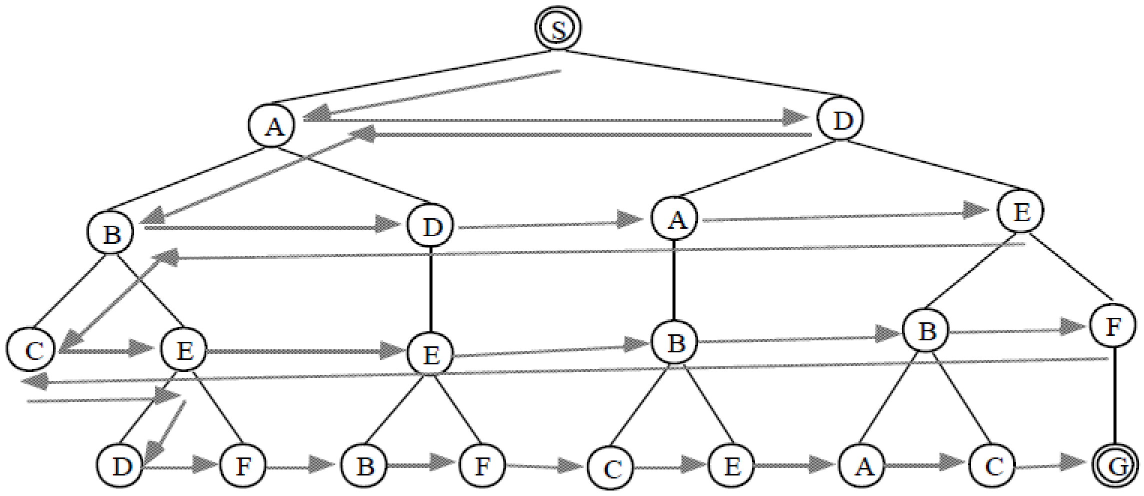


Figure: BFS

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

We can easily see that it is complete—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test.

Time complexity

- Assume a uniform tree where every state has b successors.
- The root of the search tree generates **b** nodes at the first level, each of which generates b more nodes, for a total of $\mathbf{b^2}$ at the second level. Each of these generates **b** more nodes, yielding $\mathbf{b^3}$ nodes at the third level, and so on.
- Now suppose that the solution is at depth **d**.
- In the worst case, it is the last node generated at that level.
- Then the total number of nodes generated is

  $$b + b^2 + b^3 + \cdots + b^d = O(b^d)\ .$$

(If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $O(b^{d+1})$.)

Space complexity

- For any kind of graph search, which stores every expanded node in the **explored** set, the space complexity is always within a factor of **b** of the time complexity.
- For breadth-first graph search in particular, **every node generated remains in memory**.
- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier.
- So the space **complexity is $\mathbf{O(b^d)}$,** i.e., it is dominated by the size of the frontier

Optimality

- Breadth-first search is optimal if the path cost is a non-decreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

<u>Problems with BFS</u>

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

Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- By a simple extension, we can find an algorithm that is optimal with any **step-cost function**.
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by g.
- In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search. The first is that the goal test is applied to a node when it is selected for expansion .The second difference is that a test is added in case a better path is found to a node currently on the frontier.

Example:

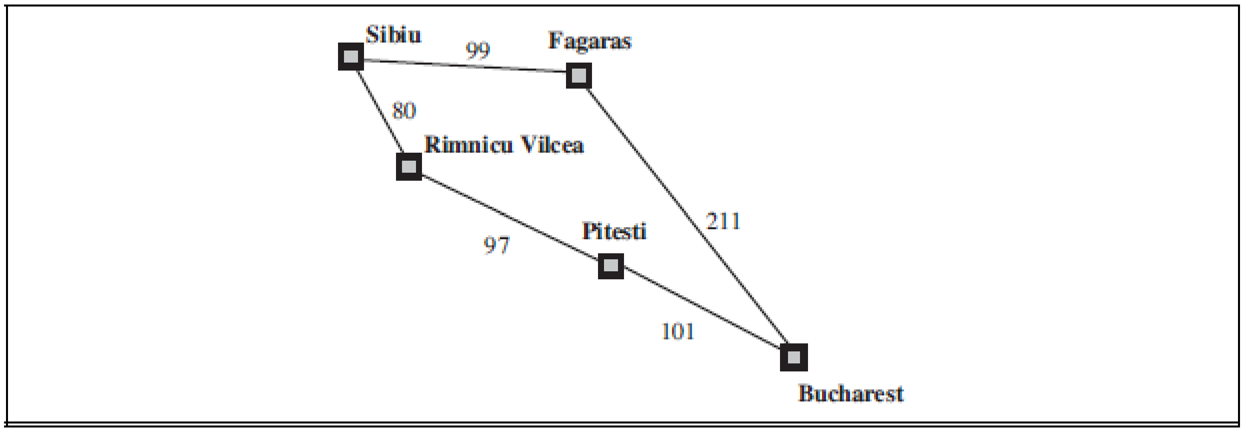


Figure : Part of the Romania state space, selected to illustrate uniform-cost search.

- The problem is to get from Sibiu to Bucharest.
- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively.

- The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177.
- The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310.
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278.
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

Uniform-cost search expands nodes in order of their optimal path cost.

Uniform-cost search does not care about the number of steps a path has, but only about their total cost.

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-first search

→ Depth-first search (DFS) always expands the deepest node in the current frontier of the search tree.

→ The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.

→ 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.

→ Depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

→ The progress of the search is illustrated in following Figure.

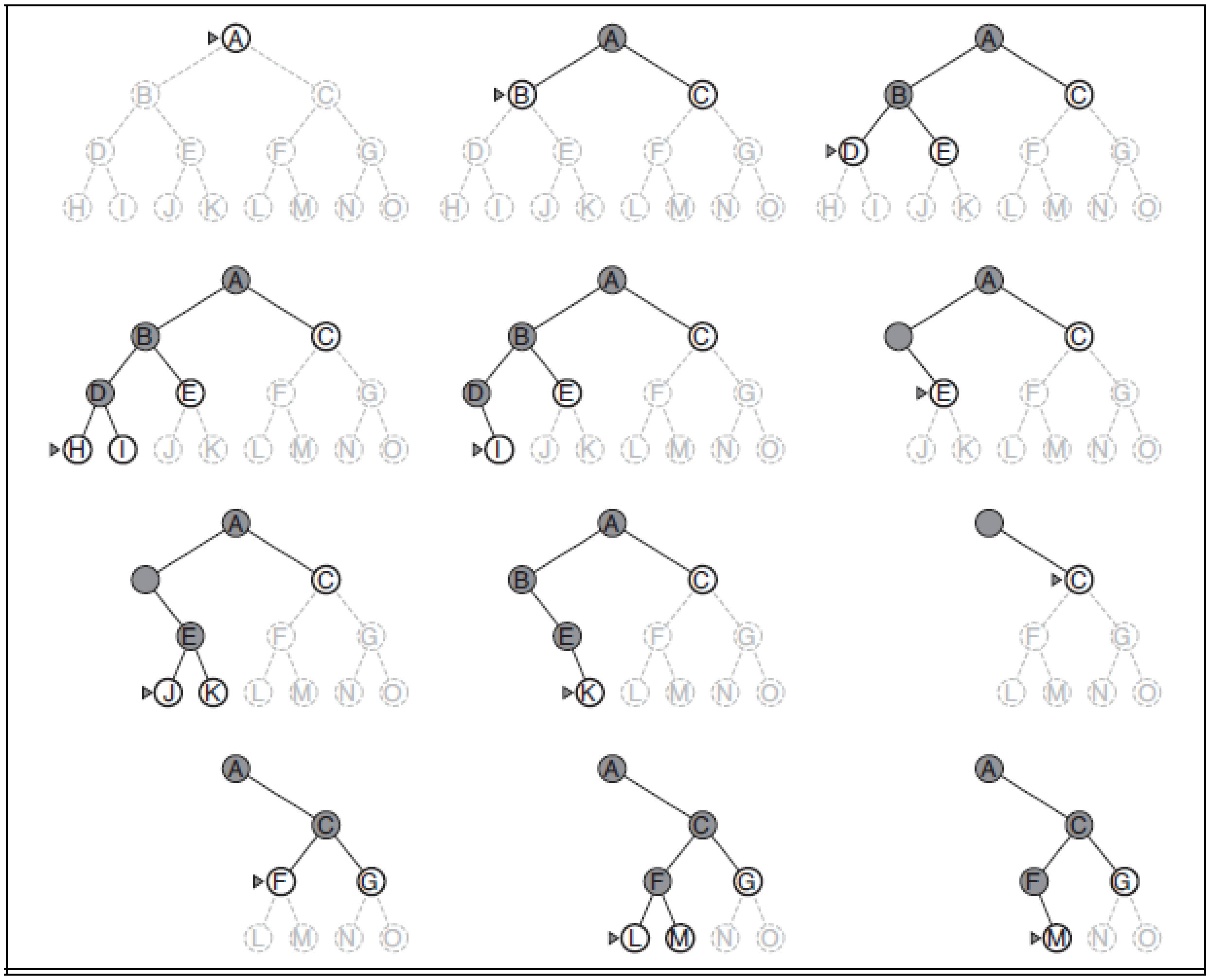


Figure: Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

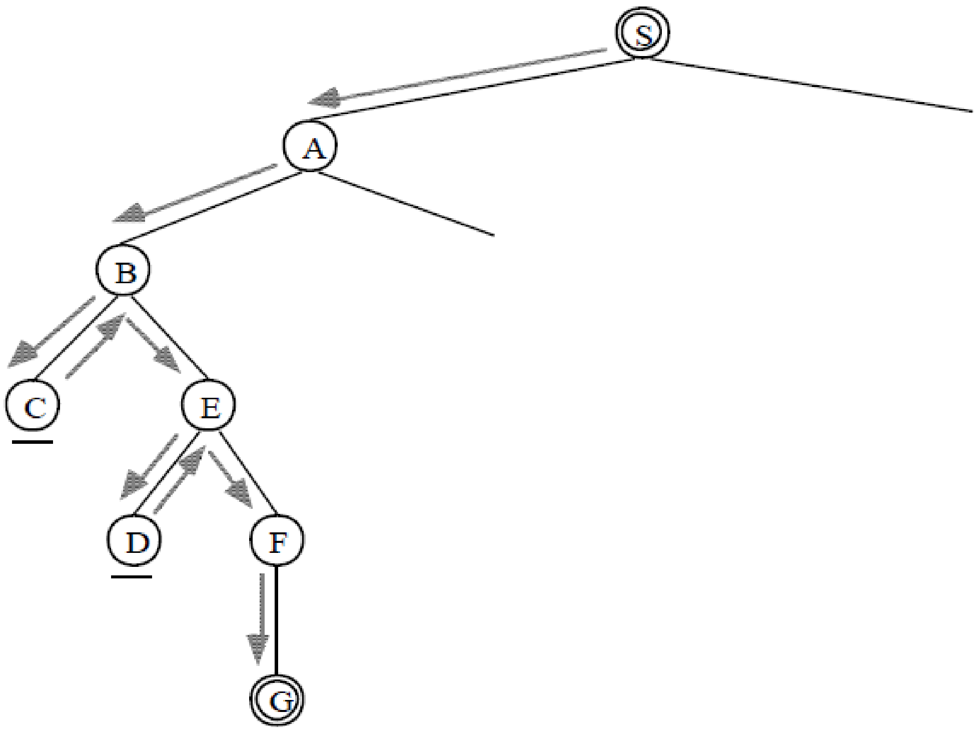


Figure :DFS

DFS Evaluation

Completeness

– 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; total no. of nodes generated:

$$b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots.. + b^m = O(b^m)$$

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 + ………………….. b m times = **O(bm)**

[Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. For a state space with branching factor b and maximum depth m, depth-first search requires storage of only **O(bm)** nodes.]

Optimality

– 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.

Depth-limited search

- The failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors.(i.e. they are not expanded) This approach is called **depth-limited search**.
- The depth limit solves the infinite-path problem.
- Even if it could expand a vertex beyond some depth, it will not do so. That is why it cannot get stuck in cycles.
- It finds solution only if it is within depth limit (l)
- **Problem**
    - If l is not properly chosen, problem arises
    - Let solution lies at depth d
    - If $(l < d)$ (that is, the shallowest goal is beyond the depth limit) then incompleteness occurs
    - If $(l > d)$ then it will be nonoptimal
- Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$
- Time Complexity **- O(b^l)** , Space Complexity- **O(bl)**

Iterative deepening depth-first search

- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search,that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found
- Iterative deepening combines the benefits of depth-first search's space efficiency and breadth-first search's completeness.

  [Like depth-first search, its memory requirements are modest: O(bd) to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node]

- In this technique, DFS is used up to some limits (l). If the goal node is not found, the depth limit is increased and searched again until the goal node is found.
- The iterative deepening search repeatedly applies depth limited search with increasing limits.
- Every time a limit is increased, the process starts from root node.
- Iterative deepening is preferred because it is the DFS with lower bound on how deep the search can go and can find solution at lower level without having to go deep on any particular node.
- In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.
- The root node is visited multiple times.
- The last level nodes are visited once, second last level nodes are visited twice and so on.

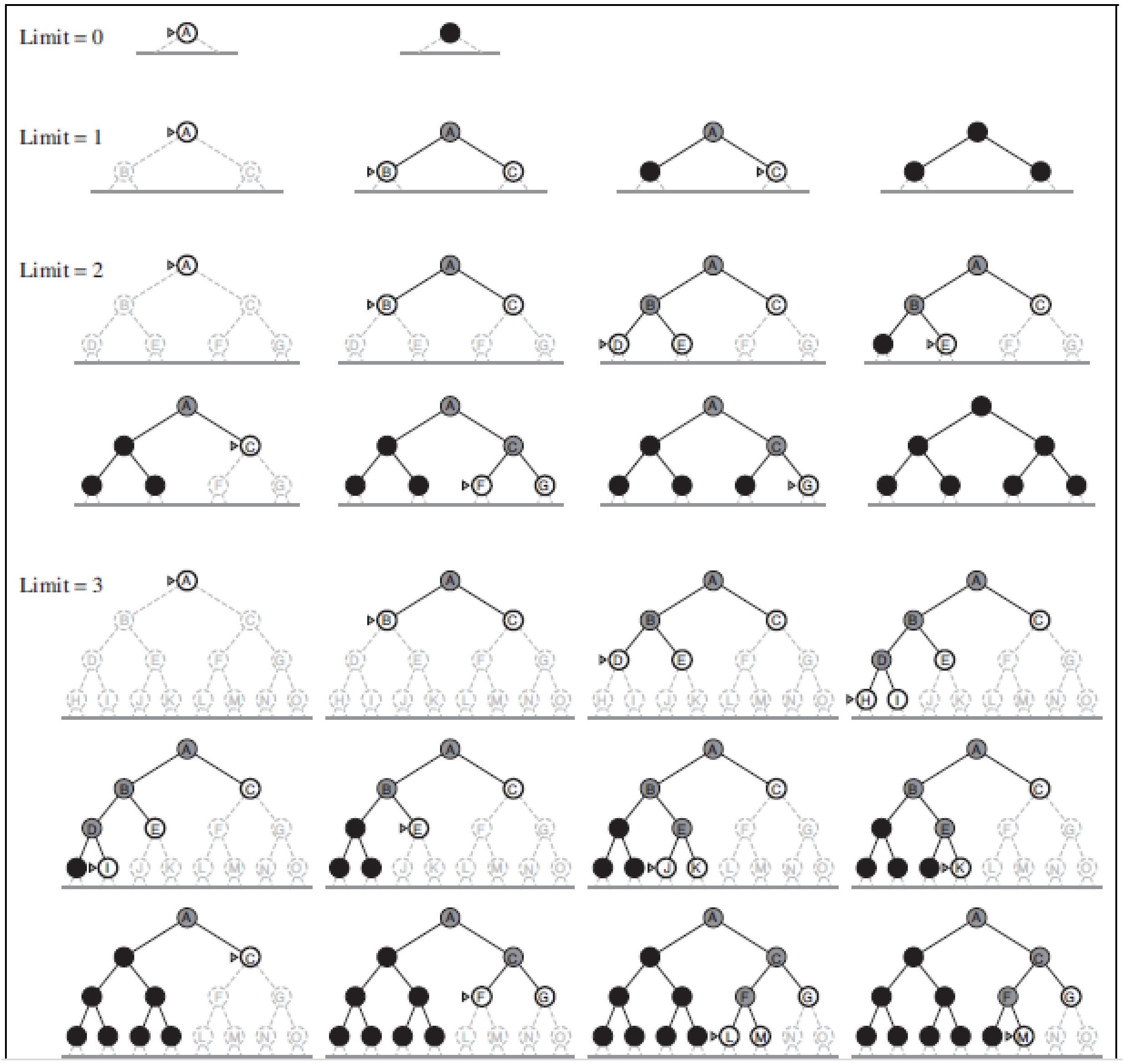


Figure : Four iterations of iterative deepening search on a binary tree.

Completeness: Yes

-Has no infinite path

Time Complexity: In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is : $(d)b + (d-1)b^2 + \cdots + (1)b^d$ So, time complexity = **O(b^d)**

Space Complexity:

- As in DFS, 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)**

Optimality:

Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.

Bidirectional search

- The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

  Time complexity:

  (using breadth-first searches in both directions) is $O(b^{d/2})$.

  Space complexity:

  The space complexity is also $O(b^{d/2})$.

  Optimality :

  May not be optimal (even if breadth first search is used in both direction)

  Completeness: Yes

- We can reduce the space complexity by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done. This space requirement is the most significant weakness of bidirectional search. The reduction in time complexity makes bidirectional search attractive, but how do we search backward? This is not as easy as it sounds. Let the predecessors of a state x be all those states that have x as a successor. Bidirectional search requires a method for computing predecessors. When all the actions in the state space are reversible, the predecessors of x are just its successors. Other cases may require substantial ingenuity.
- Bidirectional search is difficult to use in some problems like in n – queens problem. [if the goal is an abstract description, such as the goal that "no queen attacks another queen" in the n-queens problem

Informed (heuristic) Search

- Informed search strategy uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy
- 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.

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

- General approach
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function, f(n).** The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search except for the use of **f** instead of **g** to order the priority queue.
- The choice of **f** determines the search strategy.
- Most best-first algorithms include as a component of **f** a heuristic function, denoted **h(n)**

  h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

  - **h(n)** takes a node as input, but, unlike g(n), it depends only on the state at that node.
  - **h(n)** takes a node **n** and returns a non-negative real number that estimates the path cost from node **n** to the goal node.
  - if **n** is a goal node, then **h(n)=0**.

Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.

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

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

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

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

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

Special cases: based on the evaluation function.

- greedy best-first search
- A*search

Greedy best-first search

- ☞ Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.
- ☞ Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

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

- ☞ It is straight line distance (SLD) heuristic

➢ Greedy best-first search expands the node that appears to be closest to goal.

Let us see how this works for route-finding problems in Romania; we use the straight line distance heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in following figure. For example, *hSLD*(In(Arad))=366. Notice that the values of *hSLD c*annot be computed from the problem description itself.

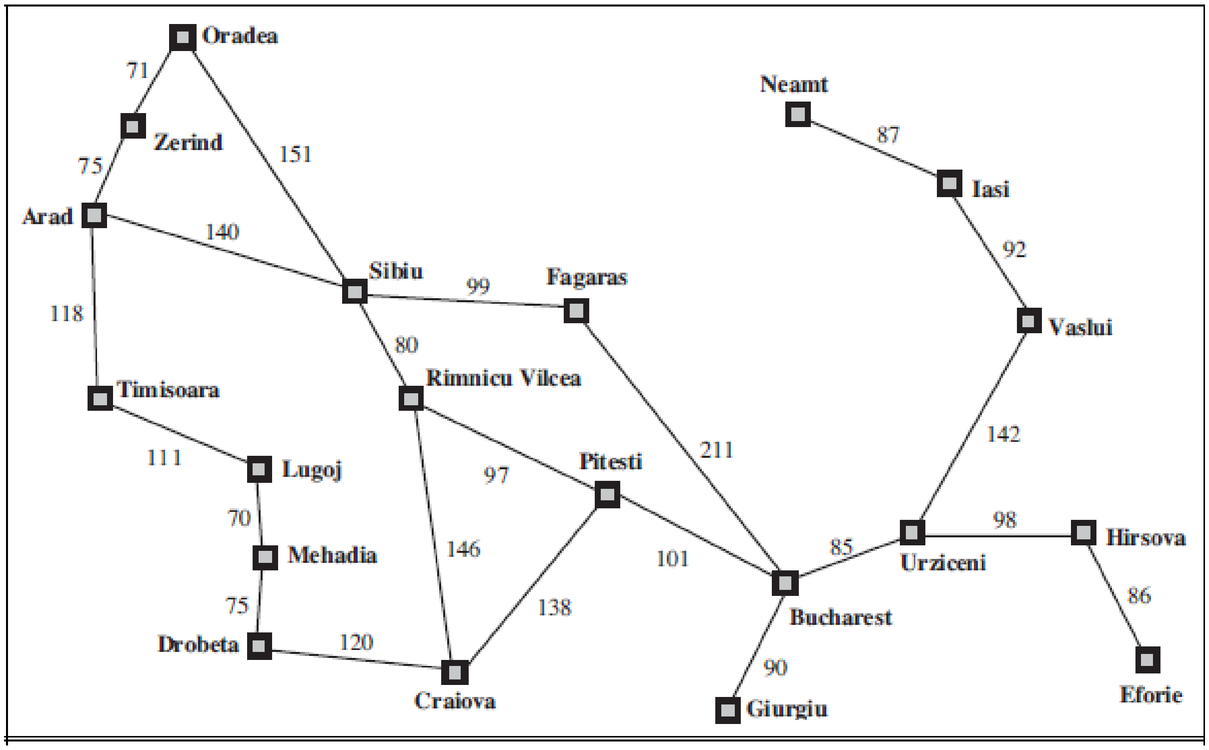


Figure: Graph of cities.

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure: Values of hSLD—straight-line distances to Bucharest.

The following figure shows the progress of a greedy best-first search using *hSLD* to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is closest. Fagaras in turn generates Bucharest, which is the goal.

➢ At each step it tries to get as close to the goal as it can. So this algorithm is called greedy

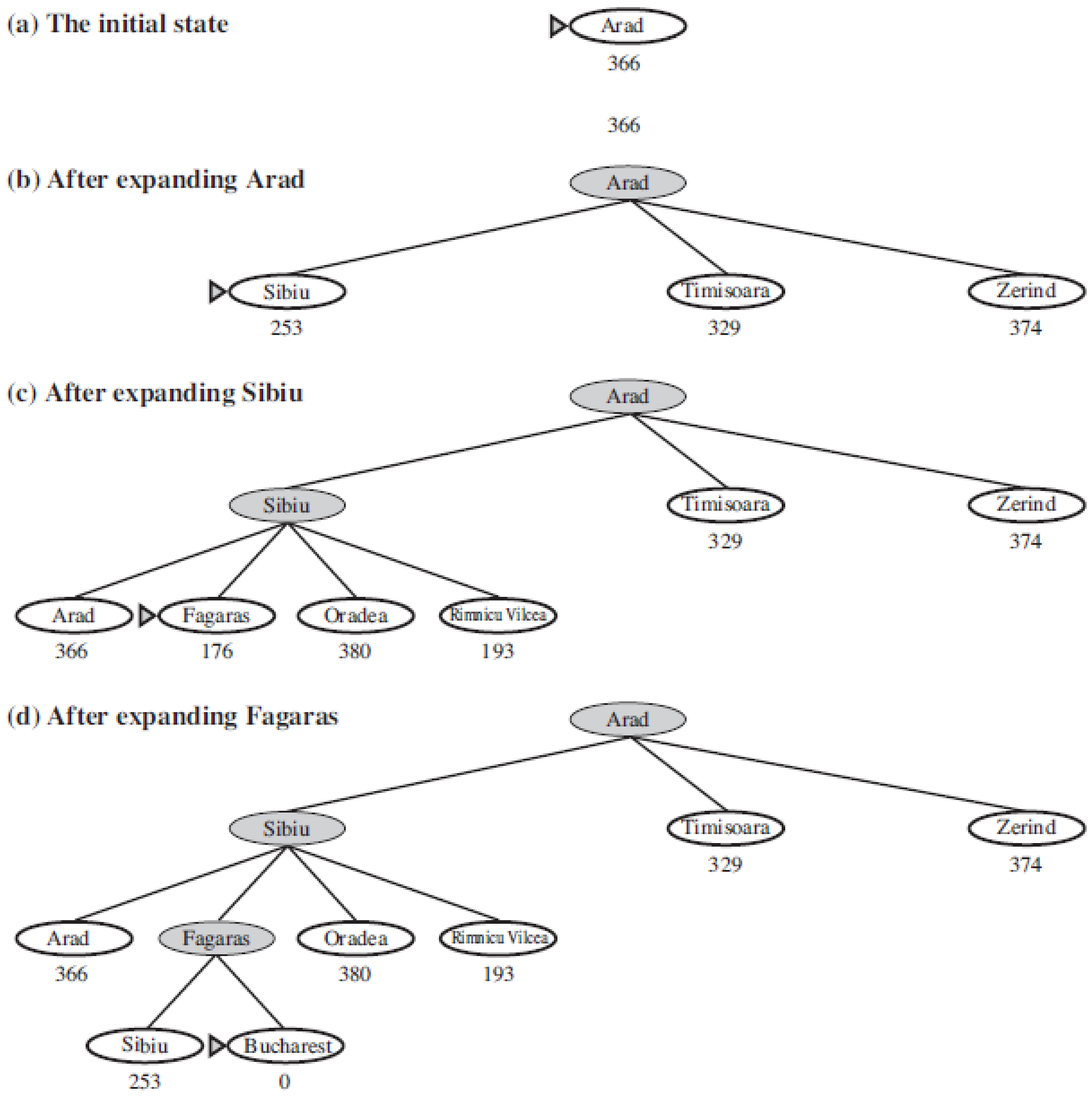


Figure: Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic hSLD. Nodes are labeled with their h-values

- Greedy Best-first search minimizes estimated cost h(n) from current node n to goal
- It is informed search but it is not optimal and it is incomplete.

A* search

- The most widely known form of best-first search is called A* search (pronounced "A-star search"). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

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

- Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = estimated\ cost\ of\ the\ cheapest\ solution\ through\ n\ .$$

- The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses **g + h** instead of **g.**

Greedy Best-first search

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

Uniform cost search

– minimizes actual cost g(n) to current node n;
– is (in most cases) optimal and complete but uninformed.

A* search

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

Idea: avoid expanding paths that are already expensive.

It finds a minimal cost-path joining the start node and a goal node for node n.

Evaluation function: $\boldsymbol{f(n) = g(n) + h(n)}$

Where,

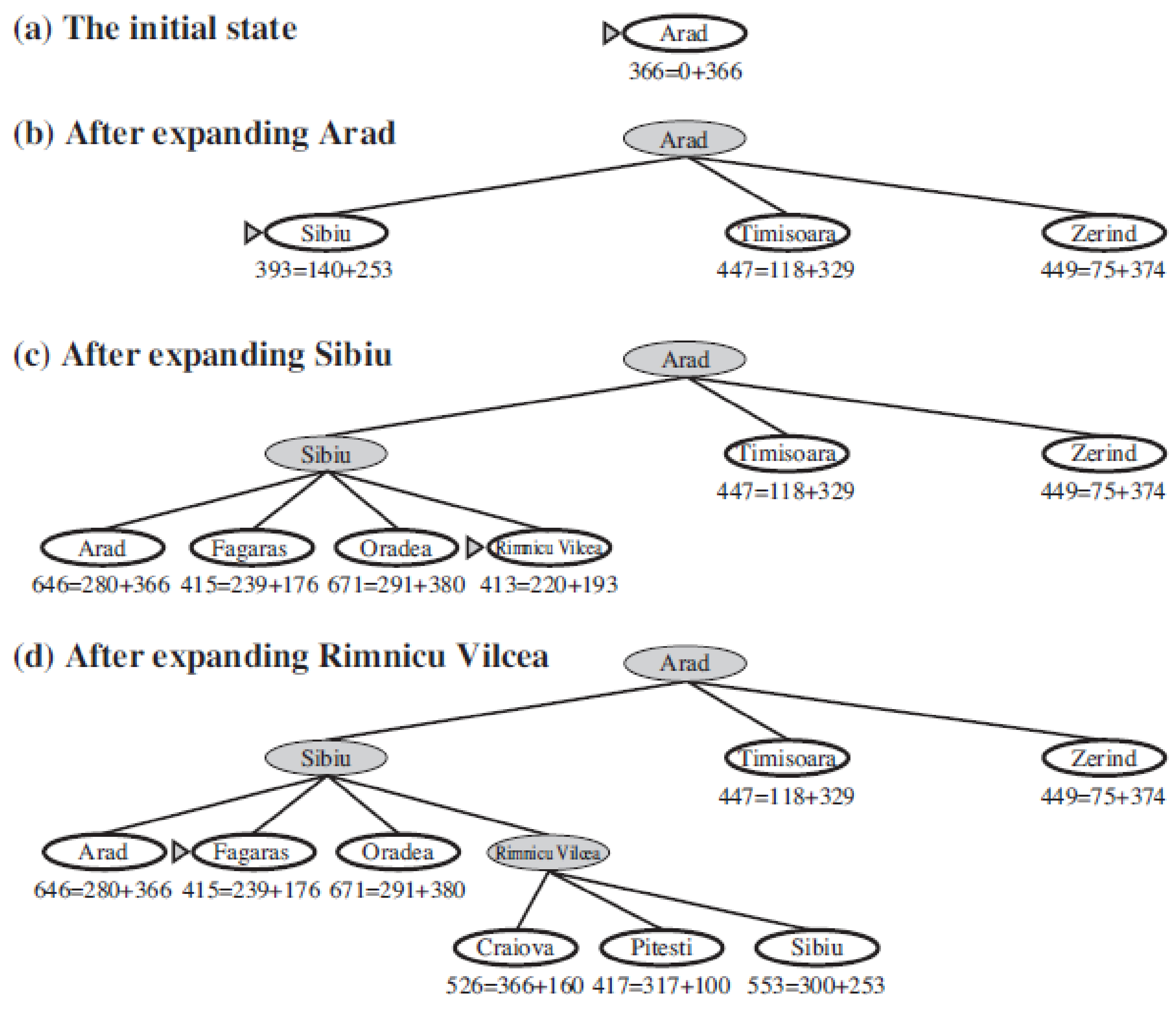
$g(n)$ = cost so far to reach n from root

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

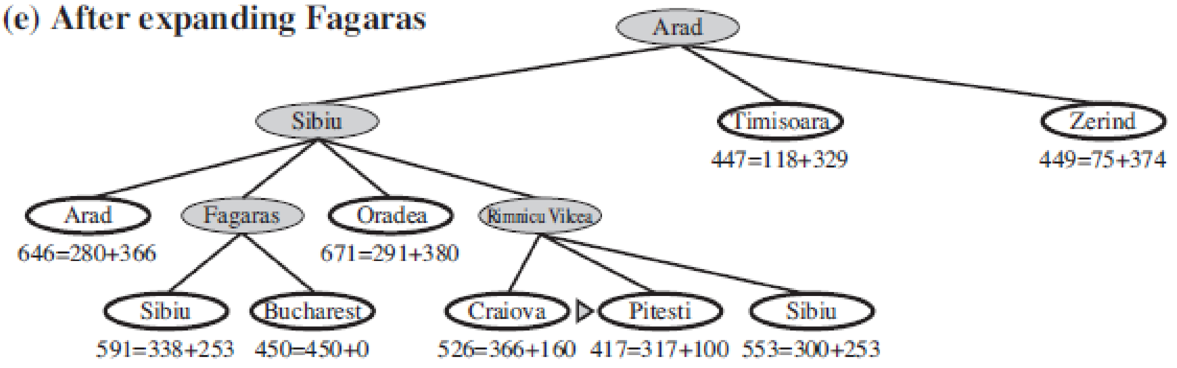
$f(n)$ = estimated total cost of path through n to goal

Example:

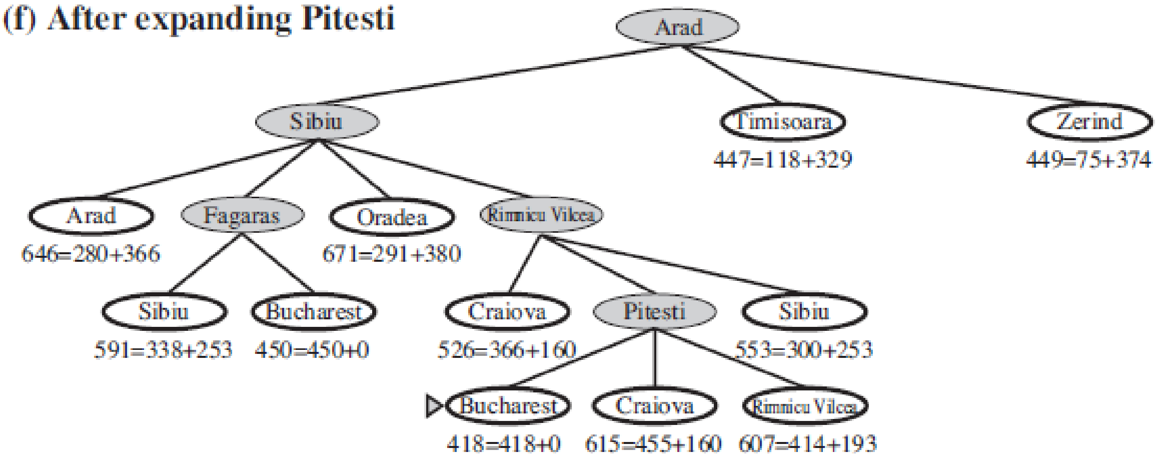
Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from above figure.



(e) After expanding Fagaras



(f) After expanding Pitesti



Conditions for optimality in A*: Admissibility and consistency

- The first condition we require for optimality is that $h(n)$ be an **admissible heuristic**.
  - An admissible heuristic is one that never overestimates the cost to reach the goal.
  - Because $g(n)$ is the actual cost to reach n along the current path, and $f(n) = g(n) + h(n)$, we have as an immediate consequence that $f(n)$ never overestimates the true cost of a solution along the current path through n
  - Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate
- A second, slightly stronger condition called **consistency** (or sometimes **monotonicity**) is required only for applications of A* to graph search

- Heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' :

$$h(n) \leq c(n, a, n') + h(n') .$$

- This is a form of the general triangle inequality, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides

→ **The tree-search version of A* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent**

Local search algorithm

→ The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path. When a goal is found, the path to that goal also constitutes a solution to the problem. In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.

→ If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node.

→ Typically, the paths followed by the search are not retained.

→ Although local search algorithms are not systematic, they have two key advantages:

(1) they use very little memory—usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

→ In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.

Hill Climbing Search

→ A local search algorithm

→ 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.

This can be described as follows:

- Start with *current-state* = *initial-state.*
- 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 sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.

The hill-climbing search algorithm (steepest-ascent version) is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbors of the current state.

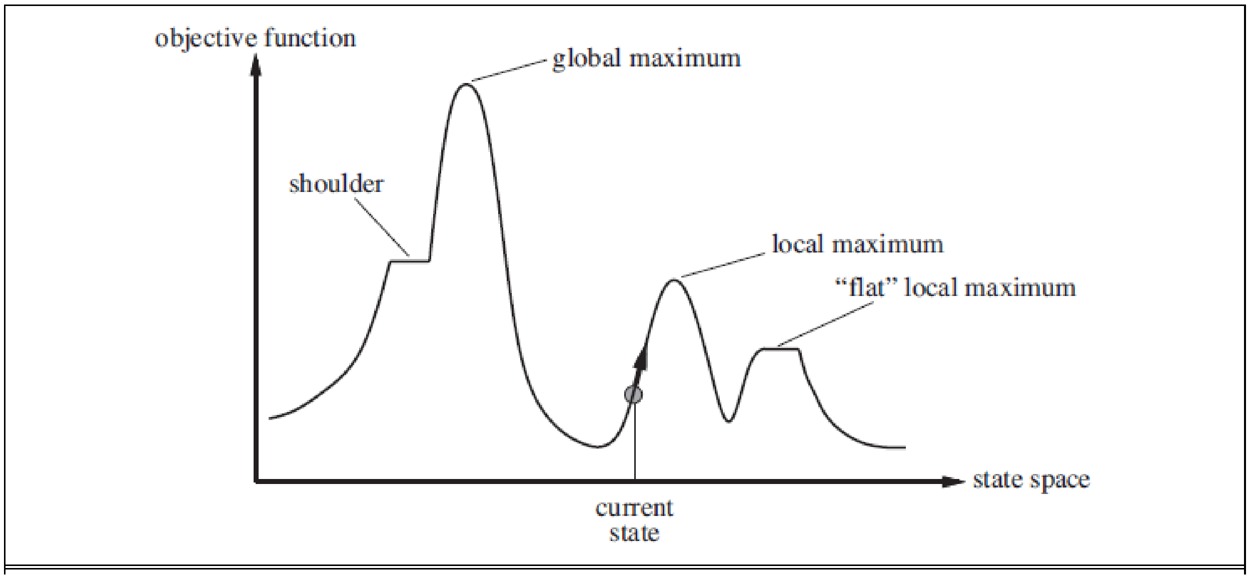


Figure : A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Problem with hill climbing search

Hill Climbing Search may fail to find a solution i.e. terminate without finding goal state from which no better states can be generated. Some reasons are as follows:

a) **Local maxima**
   - A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum
   - Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.

b) **Ridges**
   - Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate
   - Special kind of local maxima
   - Higher than the neighboring regions but the search directs towards downhill rather than top

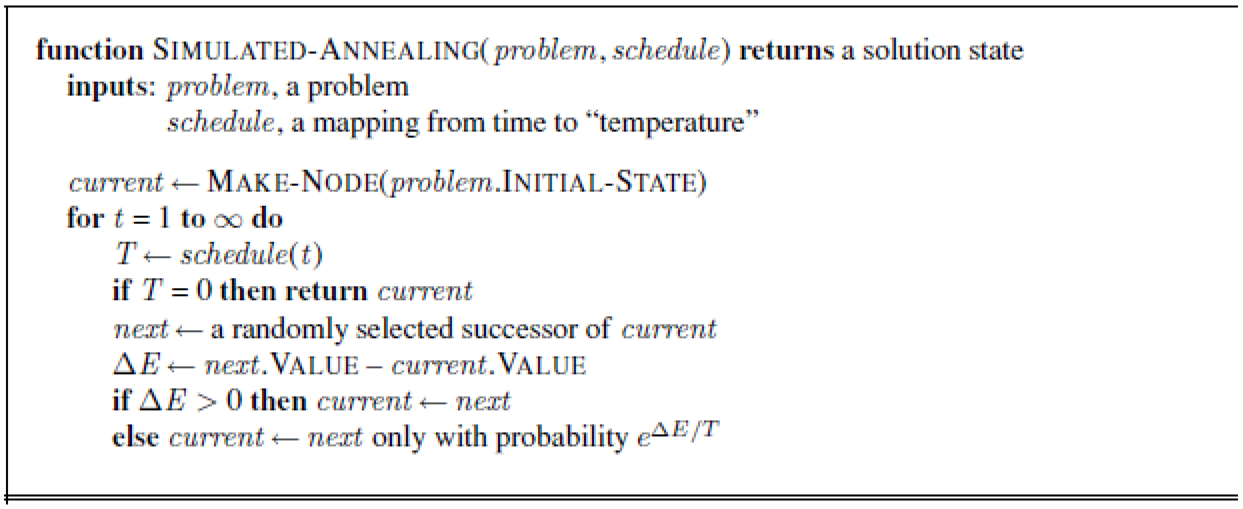
c) **Plateaux**
   - A plateau is a flat area of the state-space landscape.
   - It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible.
   - A hill-climbing search might get lost on the plateau

Simulated annealing

- A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.
- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.
- Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. **Simulated annealing** is such an algorithm.
- In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.
- To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the

deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

- The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The schedule input determines the value of the temperature T as a function of time.



```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
     T ← schedule(t)
     if T = 0 then return current
     next ← a randomly selected successor of current
     ΔE ← next.VALUE – current.VALUE
     if ΔE > 0 then current ← next
     else current ← next only with probability $e^{\Delta E/T}$
```

- The innermost loop of the simulated-annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move.
- If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened.
- The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

- Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

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 = 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 Playing

Adversarial Search (Game Playing): Competitive environments in which the agents goals are in conflict, give rise to adversarial search, often known as games.

- In AI, games means deterministic, fully observable, multi agent environments in which there are two agents whose actions must alternate and in which utility values at the end of the game are always equal and opposite. E.g., *If first player wins, the other player necessarily loses.* This opposition between the agent's utility functions make the situation *adversarial.*
  (Adversarial: dictionary meaning→involving people opposing or disagreeing with each other)
- Games are forms of multi agent environment (in which each agent needs to consider the actions of other agents and how they affect its own welfare)
  - 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 have well defined rules.
- Mathematical game theory, a branch of economics, views any multi agent environment as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive
- In AI, the most common games are of a rather specialized kind—what game theorists call deterministic, turn-taking, two-player, zero-sum games of perfect information (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial.
- Game Theory – Rational choice in multi agent scenario
- 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 with the following elements:
  - **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 1/2. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.
- 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.
  - ▪ In a game tree, node represents game states and branch represents moves between these states.
  - ▪ Game tree is used to do evaluation of any move.

<u>Game Trees</u>

- Problem spaces for typical games represented as trees, in which:
  - o **Root node:** Represents the state before any moves have been made.
  - o **Nodes:** Represents possible states of the games. Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1. And

- **Arcs:** Represents the possible legal moves for a player. Moves are represented on alternate levels of the game tree so that all edges leading from root node to the first level represent moves for the first (MAX) player and edges from the first level to second represent moves for the second (MIN) player and so on.
- **Terminal nodes** represent end-game configurations

Evaluation Function:

- → An evaluation function is used to evaluate the "goodness" of a game position.
  - – i.e., estimate of the expected utility of the game position
- → The performance of a game playing program depends strongly on the quality of its evaluation function.
- → An inaccurate evaluation function will guide an agent toward positions that turn out to be lost.
- → A good evaluation function should:
  - – Order the terminal states in the same way as the true utility function:
    - — i.e., States that are wins must evaluate better than draws, which in turn must be better than losses. Otherwise, an agent using the evaluation function might err even if it can see ahead all the way to the end of the game.
  - – For non-terminal states, the evaluation function should be strongly correlated with the actual chances of winning.
  - – The computation must not take too long i.e., evaluate faster.

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 search space of Tic-Tac-Toe.

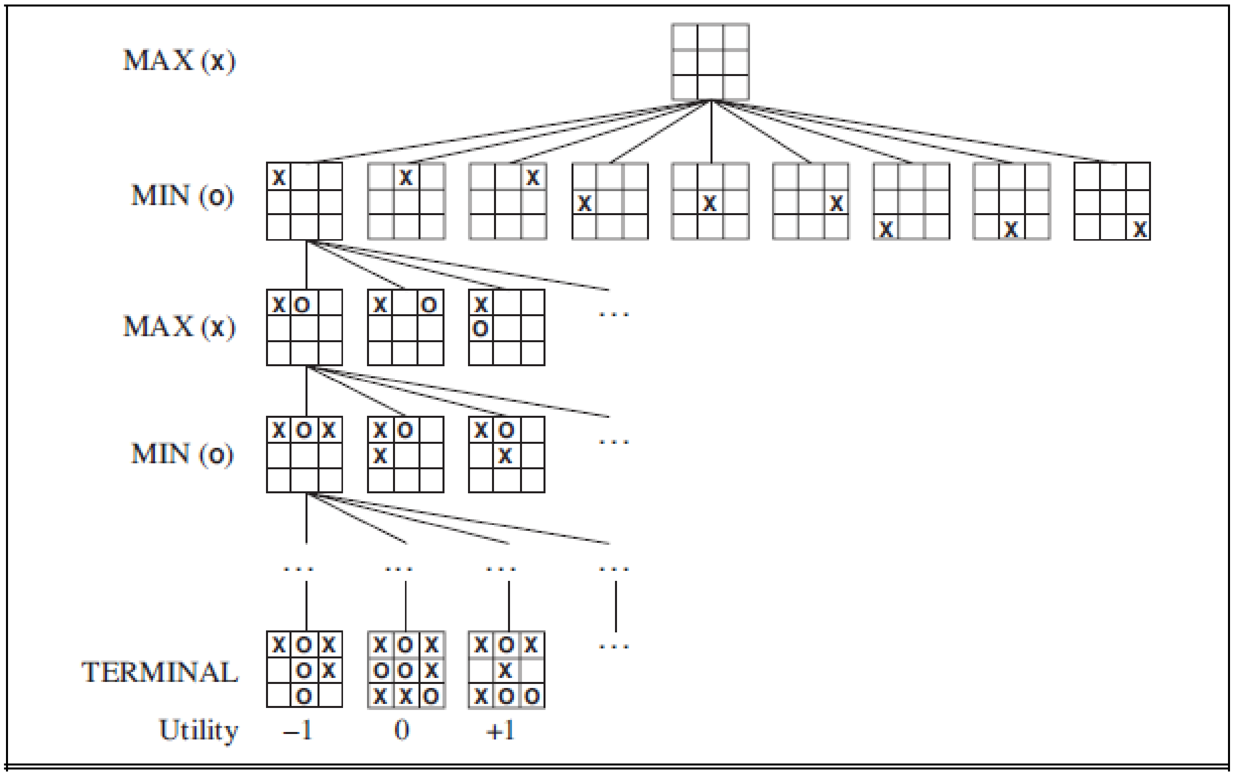


Figure : A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

- ☞ 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 above shows the search space of Tic-Tac-Toe.

The Mini-max Algorithm

- ♦ Min Max evaluation is a technique which is used to transfer the values from heuristic evaluation upward so that information can be used by player X before they make first move.
  [Heuristic Evaluation is the process of estimating the goodness of current state based on the factors you believe will contribute to win.]
- ♦ Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn based games like Tic-Tac-Toe, chess etc.
- ♦ In Minimax two players are called maximizer (MAX) and minimizer and minimizer (MIN). The maximizer tries to get highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Steps:

- Construct the complete game tree.
- Evaluate scores for leaves using evaluation function
- Back- up scores from leaves to root, considering the player type
  - for MAX player select child with maximum score
  - for MIN player select the child with minimum score
- At the root node, choose node with max value and perform corresponding move.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is **m** and there are **b** legal moves at each point, then the **time complexity** of the minimax algorithm is **O(bm).** The **space complexity** is **O(bm)** for an algorithm that generates all actions at once, or **O(m)** for an algorithm that generates actions one at a time . For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

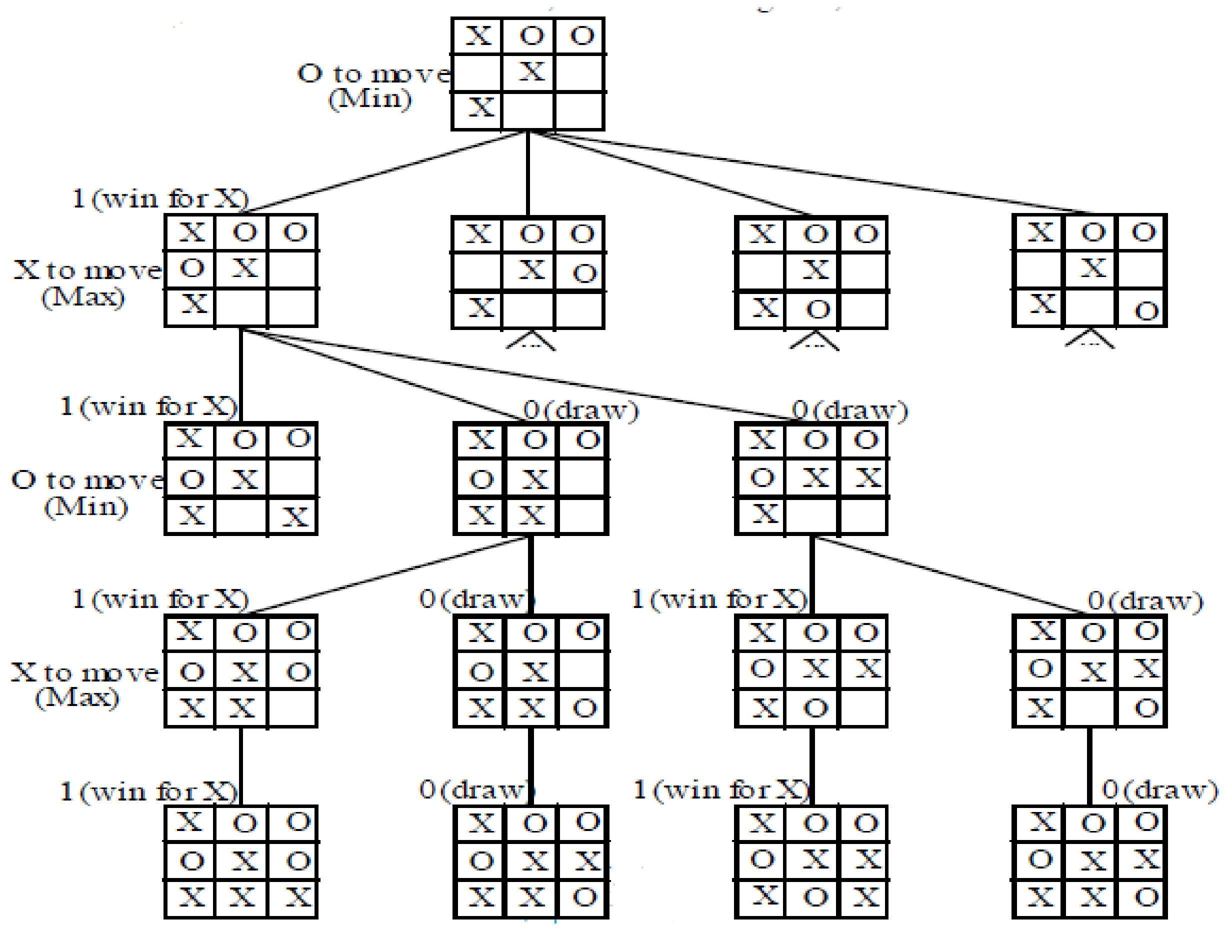
Example: Tic-Tac-Toe

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:

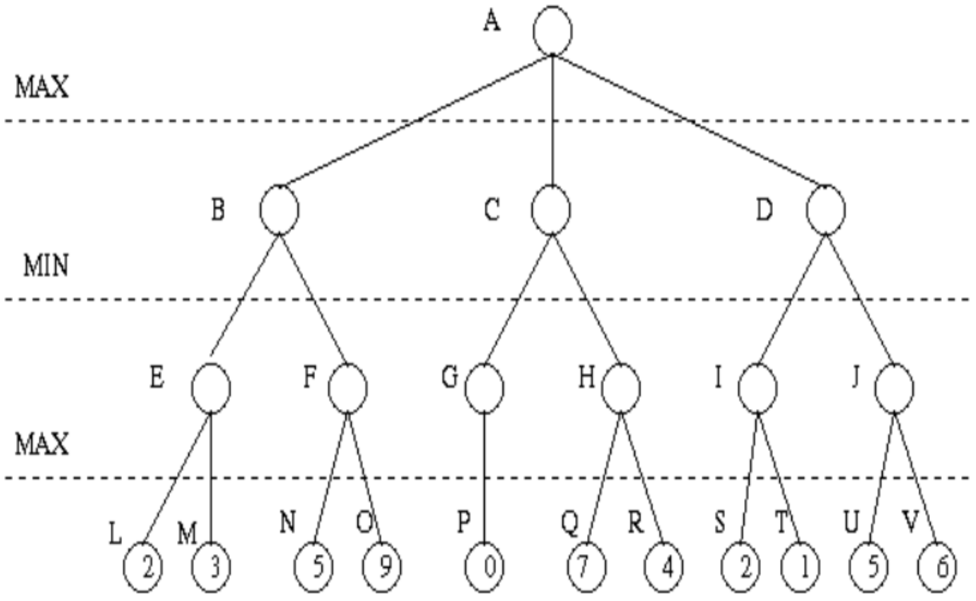
- the value of a node where it is the turn of player X to move is the maximum of the values of its successors (because X tries to maximize its outcome);
- the value of a node where it is the turn of player O to move is the minimum of the values of its successors (because O tries to minimize the outcome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

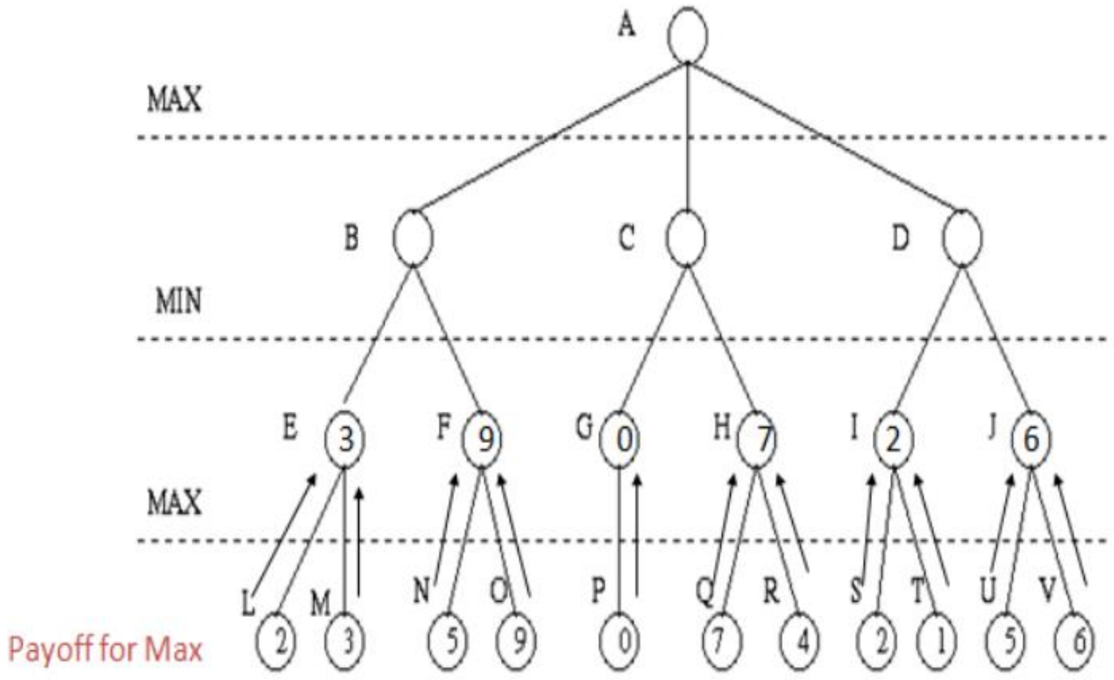
- 1 if there are three X in a row, column or diagonal;
- -1 if there are three O in a row, column or diagonal;
- 0 otherwise

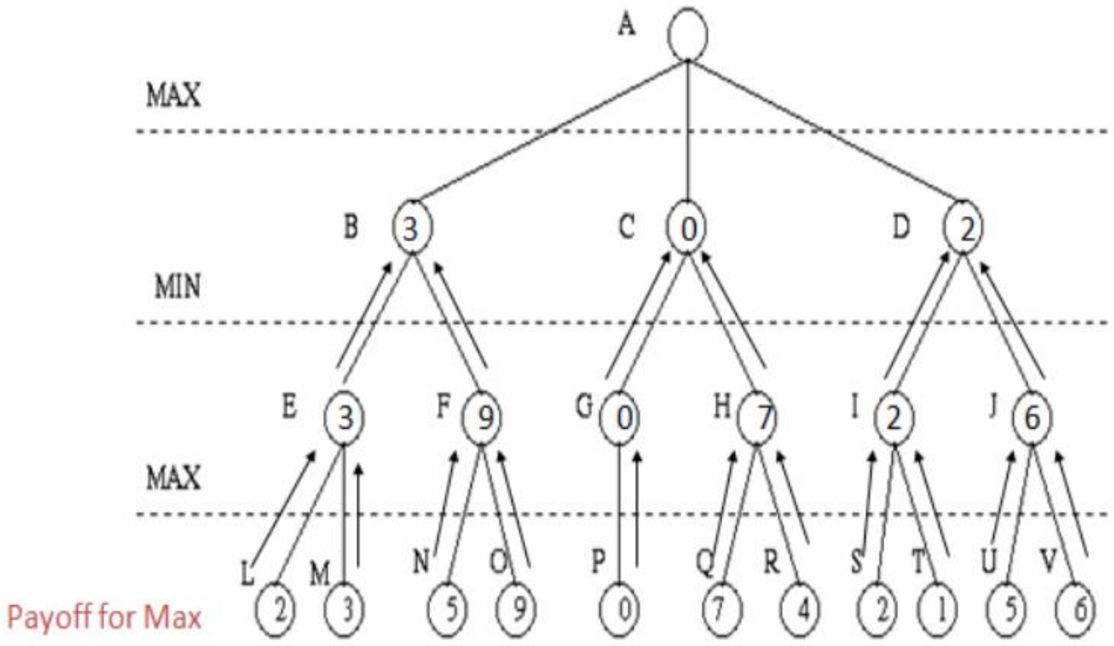

O to move (Min)
1 (win for X)
X to move (Max)
1 (win for X)
0 (draw)
0 (draw)
O to move (Min)
1 (win for X)
0 (draw)
1 (win for X)
0 (draw)
X to move (Max)
1 (win for X)
0 (draw)
1 (win for X)
0 (draw)

Mini-max search Example:

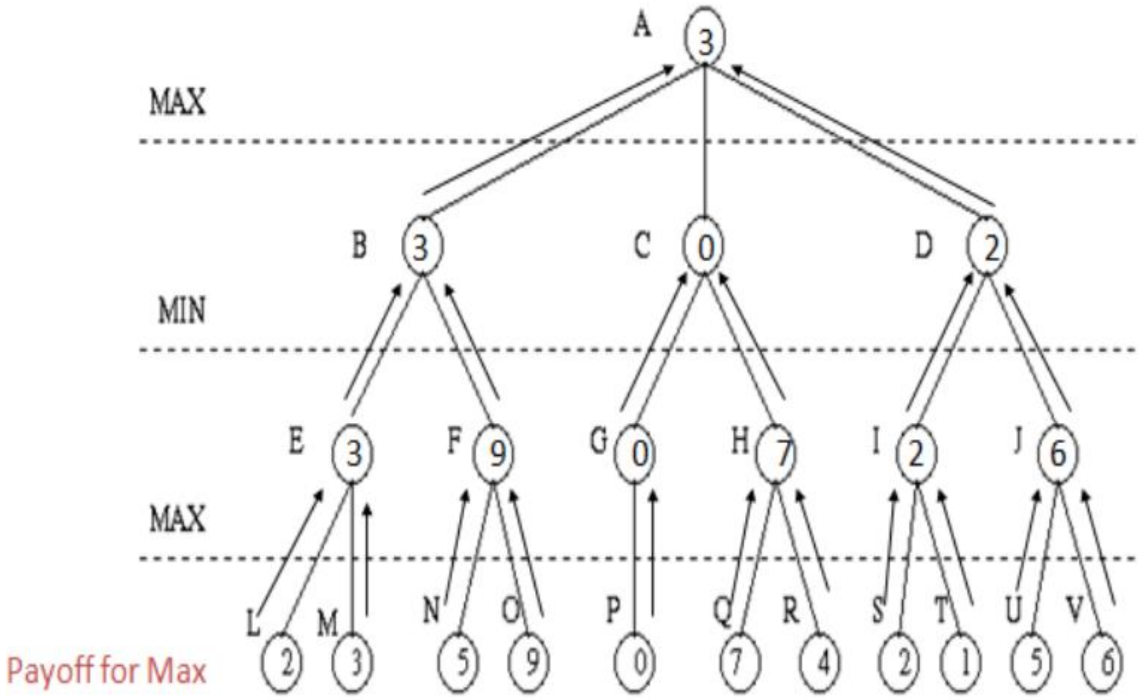


-===



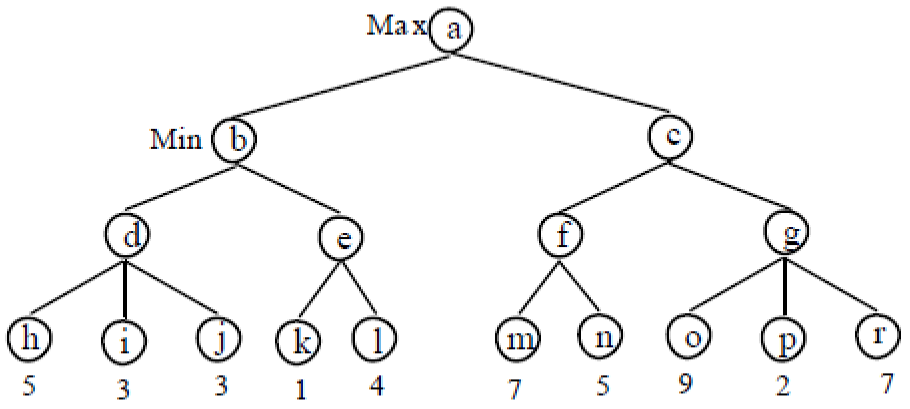

A
MAX
B
3
C
0
D
2
MIN
E
3
F
9
G
0
H
7
I
2
J
6
MAX
L
M
N
O
P
Q
R
S
T
U
V
Payoff for Max
2
3
5
9
0
7
4
2
1
5
6

===


A
3
MAX
B
3
C
0
D
2
MIN
E
3
F
9
G
0
H
7
I
2
J
6
MAX
L
M
N
O
P
Q
R
S
T
U
V
Payoff for Max
2
3
5
9
0
7
4
2
1
5
6

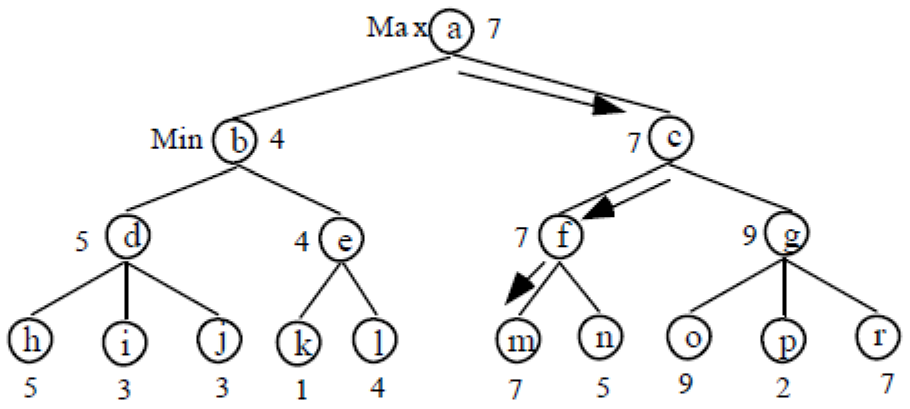
An Example:

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



Limitations of Mini-max search: –

- — Mini-max search traverse the entire search tree but it is not always feasible to traverse entire tree.
- — Time limitations

Alpha-Beta Pruning

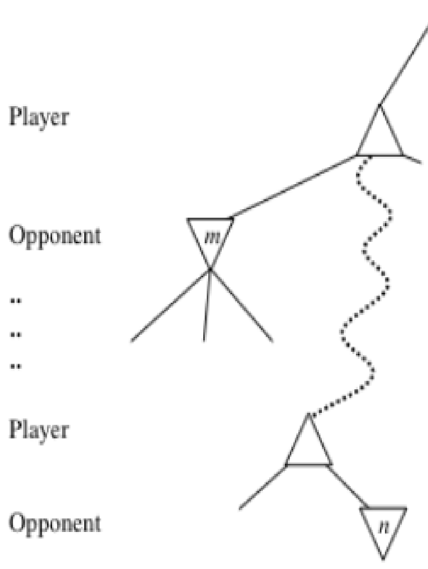
- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.(i.e. number of game states grows exponentially)
- Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half.
- The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree, which is the concept behind pruning. (i.e. we can use technique for not visiting the states which are guaranteed to have not better utility value than current state)
- 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.

We can improve on the performance of the mini-max algorithm through alpha-beta pruning.

<u>Basic idea:</u> If a move is determined worse than another move already examined, then there is no need for further examination of the node.

For Example: Consider node n in the tree.

- If player has a better choice at:
  - Parent node of n
  - Or any choice point further up
- Then n is never reached in play.
- So, When that much is known about n, it can be pruned.



Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When 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 procedure:

- $\Rightarrow$ Alpha-Beta pruning is a technique for evaluating nodes of a game tree that eliminates unnecessary evaluations. It uses two parameters, **Alpha(** α) and **Beta(** β **).**
- $\Rightarrow$ Where,
  - — α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  - — β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

  [initially $\alpha = -\infty$ and $\beta = +\infty$]

- $\Rightarrow$ Traverse the search tree in depth-first order.
- $\Rightarrow$ During traversing Alpha and Beta values inherited from the parent to child never from the children.

  Initially **Alpha = - infinity** and always try to increase, and

  Beta* = + *infinity and always try to decrease.

$\Rightarrow$ Alpha value updates only at max node and beta value update only at min node.

Max player	**Min player**
• Is **Val> Alpha? (Val** is Value backed up from the children of max player) $\rightarrow$ **Update Alpha** • Is **Alpha>= Beta?** $\rightarrow$ **Prune** (called **alpha cutoff**) • Return **Alpha**	• Is **Val< Beta?** (**Val** is Value backed up from the children of min player) $\rightarrow$ **Update Beta** • Is **Alpha>= Beta?** $\rightarrow$ **Prune** (called **beta cutoff**) • Return **Beta**

The main condition which required for alpha-beta pruning is:

$$\alpha >= \beta$$

- 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.
- Successors are examined in depth first approach.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Pseudo-code for Alpha-Beta Pruning

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, false)
  maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha
 break  //beta cut-off
 return maxEva

else                          // for Minimizer player
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha
  break    // alpha cut-off
 return minEva
```

Key points about alpha-beta pruning:

Alpha = max(current_alpha_value, value_from_child)

Beta = min(current_beta_value, value_from_child)

Node_value = maximum value among children if it's the turn of Max

= minimum value among children if it's turn of Min

- ☞ The Max player will only update the value of alpha.
- ☞ The Min player will only update the value of beta.
- ☞ While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- ☞ Only alpha and beta values are passed to the child nodes.

Example: Discussed in class.

Constraint Satisfaction Problems

In problem solving, we can use a factored representation for each state: a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a constraint satisfaction problem, or CSP.

A CSP consists of three components, X,D, and C:

- X is a set of variables, {X1, . . . ,Xn}.
- D is a set of domains, {D1, . . . ,Dn}, one for each variable.
- C is a set of constraints that specify allowable combinations of values.

✓ Each domain Di consists of a set of allowable values, {v1, . . . , vk} for variable Xi.

✓ Each constraint Ci consists of a pair *(scope, rel)* where *scope* is a tuple of variables that participate in the constraint and *rel* is a relation that defines the values that those variables can take on.

✓ A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation.

For example, if X1 and X2 both have the domain {A,B}, then the constraint saying the two variables must have different values can be written as

$\langle (X_1, X_2), [(A,B),(B,A)] \rangle$ or as $\langle (X_1, X_2), X_1 \neq X_2 \rangle$

- ☞ To solve a CSP, we need to define a state space and the notion of a solution
- ☞ A state is defined as an assignment of values to some or all variables.
- ☞ A consistent assignment (legal assignment) does not violate the constraints.
- ☞ A complete assignment is one in which every variable is assigned, and a solution to a CSP is a consistent, complete assignment.
- ☞ A partial assignment is one that assigns values to only some of the variables.

Example problem: Map coloring

"Given a map of countries/cities, and a fixed set of colors, assign a color to each region in the map in such a way that no two adjacent regions have the same color."

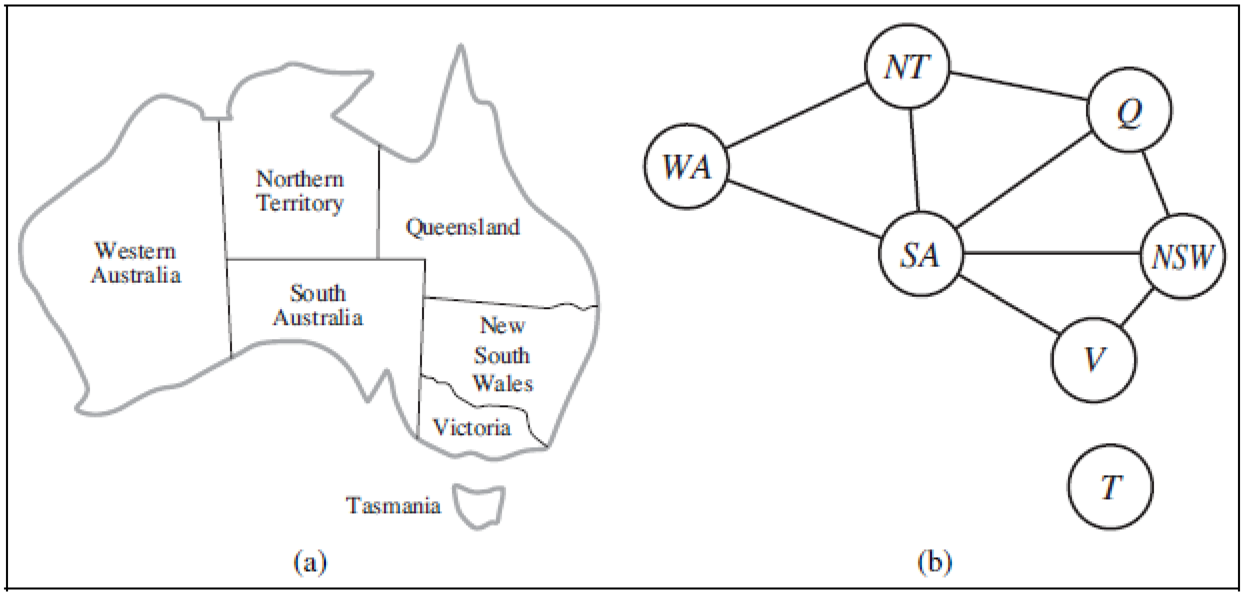


Figure: (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Consider above map of Australia. We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.

To formulate this as a CSP, we define the variables to be the regions

$$X = \{WA, NT, Q, NSW, V, SA, T\} .$$

The domain of each variable is the set Di = {red , green, blue}.

The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$\begin{aligned} C = \{ & SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ & WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\} . \end{aligned}$$

Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), SA \neq WA \rangle$, where $SA \neq WA$ can be fully enumerated in turn as

{(red , green), (red , blue), (green, red), (green, blue), (blue, red), (blue, green)} .

There are many possible solutions to this problem, such as

{WA=red ,NT =green,Q=red ,NSW =green, V =red ,SA=blue, T =red }.

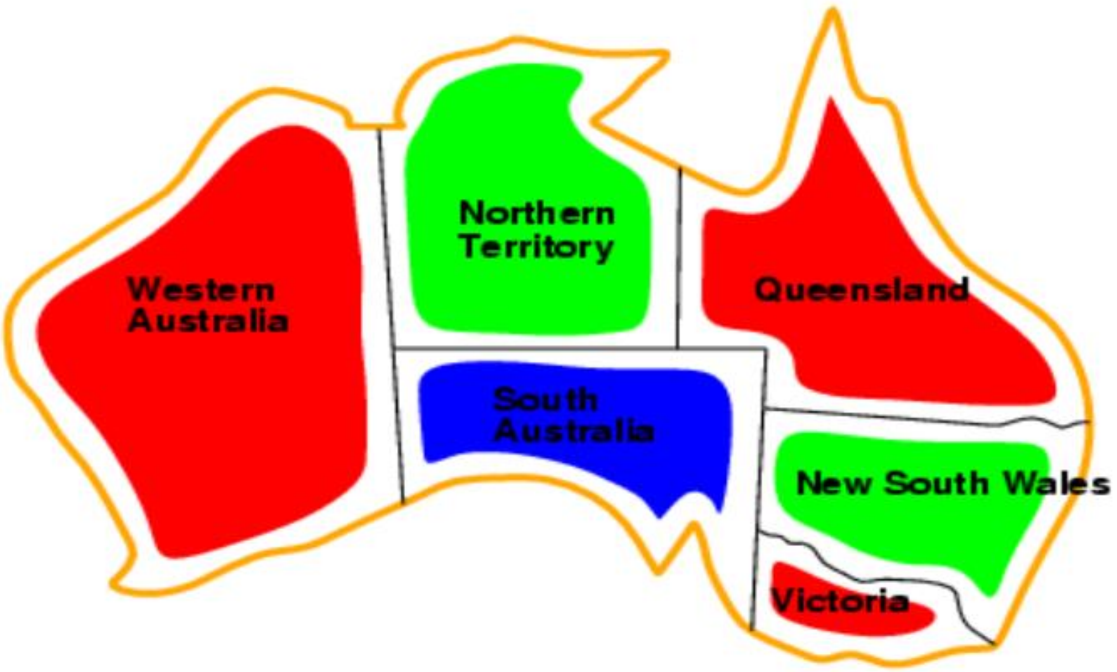


Figure: A solution of given map coloring problem

- It can be helpful to visualize a CSP as a constraint graph, as shown in above Figure (b). The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

Cryptoarithmetic problems

- Many problems in AI can be considered as problems of constraint satisfaction, in which the goal state satisfies a given set of constraint.
- Example of such a problem is Crypt-Arithmetic problem
- Class of mathematical puzzel in which each letter/ alphabet is replaced by a unique digit.
- Cryptoarithmetic puzzel involves arbitrary number of variables. In such problem, all the variables involved must have different values.
    - Values are to be assigned to letters from 0 to 9 only
    - No two letters should have the same value.
    - Each letter/symbol represents only one digit throughout the problem.
    - Numbers must not begin with zero i.e. 0567 (wrong), 567 (correct).
    - After replacing letters by their digits, the resulting arithmetic operations must be correct.

Example problems:

Solve the following puzzle by assigning numeral (0-9) in such a way that each letter is assigned unique digit which satisfy the following addition.

$$\begin{array}{rcccc} & & T & W & O \\ + & & T & W & O \\ \hline & F & O & U & R \end{array}$$

$$\begin{array}{rr} & \text{YOUR} \\ + & \text{YOU} \\ \hline & \text{HEART} \end{array}$$

$$\begin{array}{rccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

$$\begin{array}{rr} & \text{TOM} \\ + & \text{NAG} \\ \hline & \text{GOAT} \end{array}$$

```
  CROSS          BASE          FOUR
+ ROADS        + BALL         +FOUR
 DANGER         GAMES         EIGHT
```

Solution for TWO + TWO = FOUR

Variables: {F, T, U, W, R, O, c1, c2, c3 }

Domains: {0,1,2,3,4,5,6,7,8,9}

Constraints: All diff (F,T,U,W,R,O)

• where c1, c2, and c3 are auxiliary variables representing the digit (0 or 1) carried over into the next column.

- $O + O = R \longrightarrow c_1$
- $c_1 + W + W = U \longrightarrow c_2$
- $c_2 + T + T = O \longrightarrow c_3$
- $c_3 = F, T \neq 0, F \neq 0$

```
c3  c2  c1
     T   W   O
+    T   W   O
F    O   U   R
```

- Here we are adding two three letters words but getting a four letters word. This indicates that F= c3=1
- Now, c2+T+T= O+ 10……. Because c3=1
- C2 can be 0 or 1 . Let c2=0 then T should be > 5 i.e T can be {6, 7,8,9}
- Let T= 9 then C2+T+T= O+10 0+9+9=O+10 from this O= 8
- Now O+O=R+10 8+8= R+10 From this R=6
- Now, c1+W+W=U here c1=1 and U and W can be {2,3,4,5,7}
- But , c2= 0 so let W= 2 then 1+2+2=U i.e., U=5
- Now replacing each letter in the puzzle by its corresponding digit and testing their arithmetic correctness:

```
   T W O          928
 + T W O        + 928
 -------        -----
 F O U R         1856
```

- This assignment satisfies all the constraint so this is the final solution

Solution for FOUR + FOUR = EIGHT

Variables:{F,O,U,R,E,I,G,H,T, C1,C2, C3 ,C4}

Domains: {0,1,2,3,4,5,6,7,8,9}

Constraints: Alldiff (F,O,U,R,E,I,G,H,T)

where C1, C2, and C3 are auxiliary variables representing the digit (0 or 1) carried over into the next column.

```
 c4 c3 c2 c1
    FOUR
  + FOUR
  ------
   EIGHT
```

- Here we are adding two three letters words but getting a four letters word. This indicates that E= c4=1 ….. Because E is left most letter so it should not be 0.
- Now c3+F+F= I+10 , here c3 can be 0 or 1 and F should be greater than 5. i.e {6,7,8,9}
- Let c3= 0 and F= 9 then 0+9+9=I+10 from this I=8
- Now, c2+O+O=G…… since c3=0
- C2 can be 0 or 1 and O can be {2,3,4}. Let c2= 0 and O =2 Then G= 4.
- R+R= T here R can be {3,5,6,7}
- If we let R= 3 this leads to dead end so let R=5 then T=0 and c1=1
- C1+U+U=H here c1= 1 and c2=0 and U can be{3}
- Form this U=3 then H=7

```
 FOUR       9235
+FOUR     + 9235
------    ------
EIGHT     18470
```

Problem: Crypt-Arithmetic puzzle : Solve the following puzzle by assigning numeral (0-9) in such a way that each letter is assigned unique digit which satisfy the following addition.

```
    S  E  N  D
 +  M  O  R  E
 -------------
 M  O  N  E  Y
 -------------
```

Initial problem state:

– S = ? M= ? C1 = ?

– E = ? O = ? C2 = ?

– N = ? R = ? C3 = ?

– D = ? E = ? C4 = ?

Goal states: A goal state is a problem state in which all letters have been assigned a digit in such a way that all constraints are satisfied

Carries:

$$C_4 = ? \,; C_3 = ? \,; C_2 = ? \,; C_1 = ?$$

$$\begin{array}{cccccc}
C_4 & C_3 & C_2 & C_1 & \longleftarrow & \text{Carry} \\
 & S & E & N & D & \\
+ & M & O & R & E & \\
\hline
M & O & N & E & Y & \\
\hline
\end{array}$$

Constraint equations:

$$\begin{aligned}
Y &= D + E \longrightarrow C_1 \\
E &= N + R + C_1 \longrightarrow C_2 \\
N &= E + O + C_2 \longrightarrow C_3 \\
O &= S + M + C_3 \longrightarrow C_4 \\
M &= C_4
\end{aligned}$$

- We can easily see that M has to be non zero digit, so the value of C_4 is 1.

1. $M = C_4 \Rightarrow \mathbf{M = 1}$
2. $O = S + M + C_3 \longrightarrow C_4$

   For $C_4 = 1$, $S + M + C_3 > 9 \Rightarrow S + 1 + C_3 > 9 \Rightarrow S + C_3 > 8$. If $C_3 = 0$, then $S = 9$ else if $C_3 = 1$, then $S =$ 8 or 9.

- We see that for $S = 9$
  - $C_3 = 0$ or 1
  - It can be easily seen that $C_3 = 1$ is not possible as $O = S + M + C_3 \Rightarrow O = 11 \Rightarrow$ O has to be assigned digit 1 but 1 is already assigned to M, so not possible.
  - Therefore, for $C_3 = 0$, $O = 10$ and thus O is assigned 0 (zero).

Therefore, $O = 0$ $\boxed{\mathbf{M = 1, O = 0}}$

---

3. $N = E + O + C_2 \longrightarrow C_3 = 0$
   - Since $O = 0$, $N = E + C_2$. Since $N \neq E$, therefore, $C_2 = 1$.

Hence $N = E + 1$

- Now E can take value from 2 to 8 {0,1,9 already assigned so far }
  - If $E = 2$, then $N = 3$.
    - Since $C_2 = 1$, from $E = N + R + C_1$, we get $12 = N + R + C_1$
    - If $C_1 = 0$ then $R = 9$, which is not possible as we are on the path with $S = 9$
    - If $C_1 = 1$ then $R = 8$, then
      - ❑ From $Y = D + E$, we get $10 + Y = D + 2$.
      - ❑ For no value of D, we can get Y.
  - Try similarly for E = 3, 4. We fail in each case.

- If E = 5, then N = 6
  - Since $C_2 = 1$, from E = N + R + C_1, we get 15 = N + R + C_1,
  - If $C_1 = 0$ then R = 9, which is not possible as we are on the path with S = 9.
  - If $C_1 = 1$ then R = 8, then
    - ❑ From Y = D + E , we get 10 + Y= D + 5 i.e., 5 + Y = D.
    - ❑ If Y = 2 then D = 7. These values are possible.

- Hence we get the final solution as given below and on backtracking, we may find more solutions.

S = 9 ; E = 5 ; N = 6 ; D = 7 ; M = 1 ; O = 0 ; R = 8 ;Y = 2