# Searching

In the simplest case of an agent reasoning about what it should do, the agent has a state-based model of the world, with no uncertainty and with goals to achieve. The agent can determine how to achieve its goals by searching in its representation of the world state space for a way to get from its current state to a goal state. It can find a sequence of actions that will achieve its goal before it has to act in the world.

This problem can be abstracted to the mathematical problem of finding a path from a start node to a goal node in a graph. This idea of search is the computation inside the agent. It is different from searching in the world, when it may have to act in the world, for example, an agent searching for its keys, lifting up cushions, and so on. It is also different from searching the web, which involves searching for information. Searching in this case means searching in an internal representation for a path to a goal.

The idea of search is straightforward: the agent constructs a set of potential partial solutions to a problem that can be checked to see if they truly are solutions or if they could lead to solutions. Search proceeds by repeatedly selecting a partial solution, stopping if it is a path to a goal, and otherwise extending it by one more arc in all possible ways.

**Defining a search problem**

Following are the five components that are required to define a search problem:

- The initial state that the agent starts in.
- 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. We say that each of these actions is applicable in s.
- A description of what each action does
- The goal test, which determines whether a given state is a goal state. Sometimes there is a clear set of possible goal states, and the test simply checks whether the given state is one of them.
- A path cost function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

## Simple problem solving agent

function SIMPLE-PROBLEM-SOLVING-AGENT(percept ) returns an action

persistent: seq, an action sequence, initially empty

state, some description of the current world state

goal , a goal, initially null

problem, a problem formulation

state←UPDATE-STATE(state, percept )

if seq is empty then

goal ←FORMULATE-GOAL(state)

problem ←FORMULATE-PROBLEM(state, goal )

seq ←SEARCH(problem)

if seq = failure then return a null action

action ←FIRST(seq)

seq ←REST(seq)

return action

**Goal formulation**: Goal formulation is the first step in problem solving. It is based on the current situation and the agent's performance measure. Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.

**Problem formulation:** Problem formulation is the process of deciding what actions and states to consider, given a goal.

## Evaluating the performance of search algorithm:

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

**Optimality**: Does the strategy find the optimal solution? i.e. is the solution best among all possible solutions?

**Time complexity**: How long does it take to find a solution? It is usually measured in terms of number of nodes visited.

**Space complexity**: How much memory is needed to perform the search? It is usually measured in terms of maximum number of nodes stored in the memory at the time of finding solution.

# Blind Search(Uninformed Search)

Blind search, also called uninformed search are the search strategies that have no additional information about states beyond that provided in the problem definition. Suppose, for example, that you are finding your way through a maze. In a blind search you might always choose the far left route regardless of any other alternatives. All these algorithms 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.

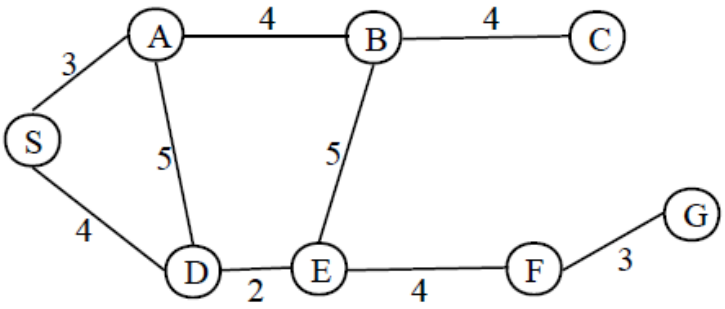



Figure. A graph representation of a map. Here the path from node S to node G is to be determined

## Types of Blind Search Algorithm:

1. Breadth first search
2. Uniform cost search
3. Depth first search
4. Depth limited search
5. Iterative deepening search
6. Bidirectional search

## 1. Breadth first search

**Breadth-first search** 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.

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

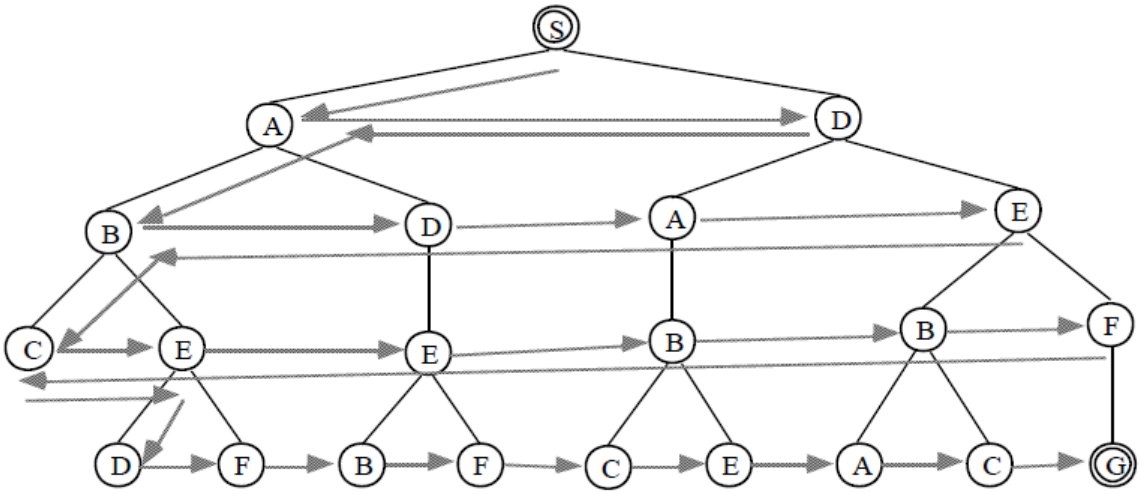



*Figure 1 Solution using BFS algorithm*

**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 $O(b^d)$,** 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.

Problems with BFS

- 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.

## 2. 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.

Analysis

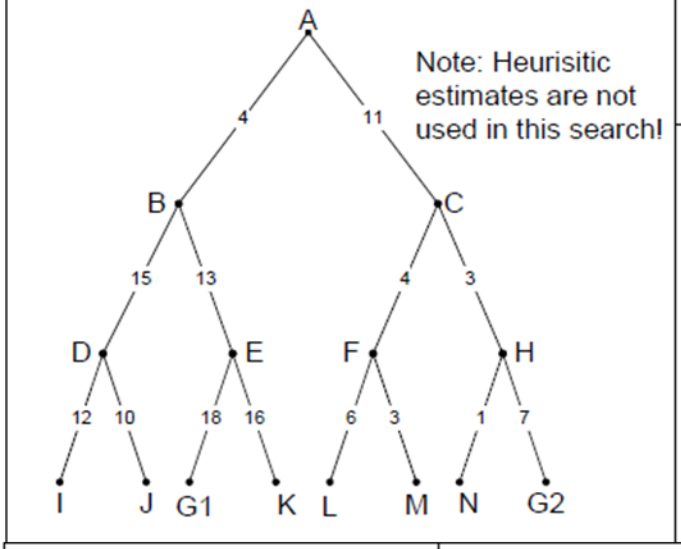
•Complete? Yes, if step cost $\geq \varepsilon$ (small positive number).

•Time? Maximum as of BFS

•Space? Maximum as of BFS.

•Optimal? Yes

**Consider an example:**

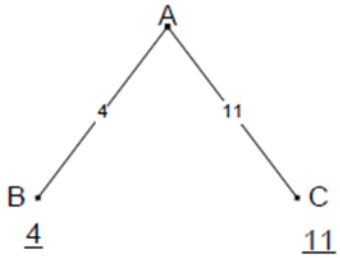



Note: Heuristic estimates are not used in this search!

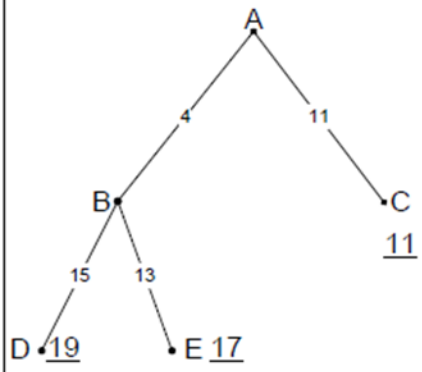
- Start with root node A.

A
0

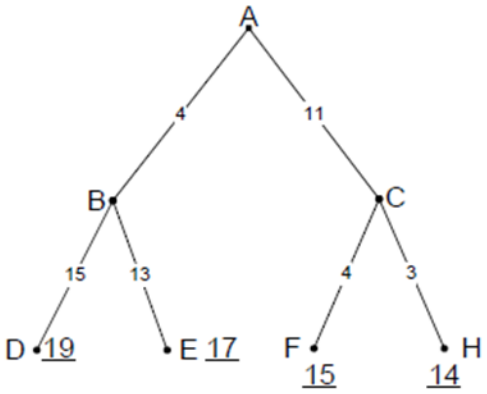
Paths from root are generated.



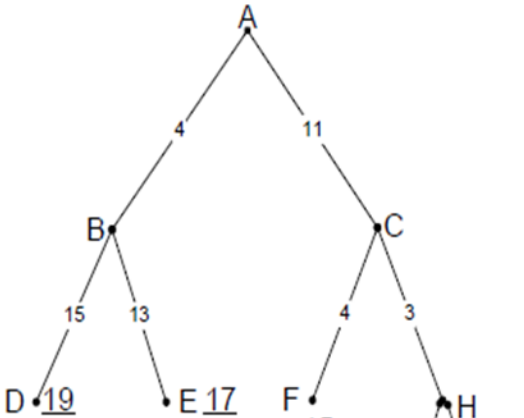

Since B has the least cost, we expand it.



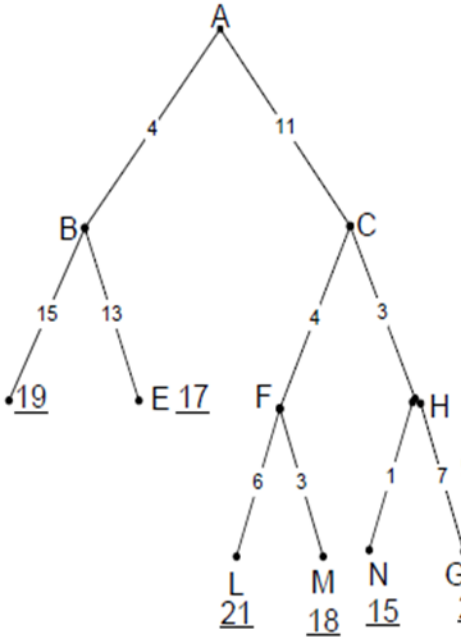

Of our 3 choices, C has the least cost so we'll expand it.




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




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

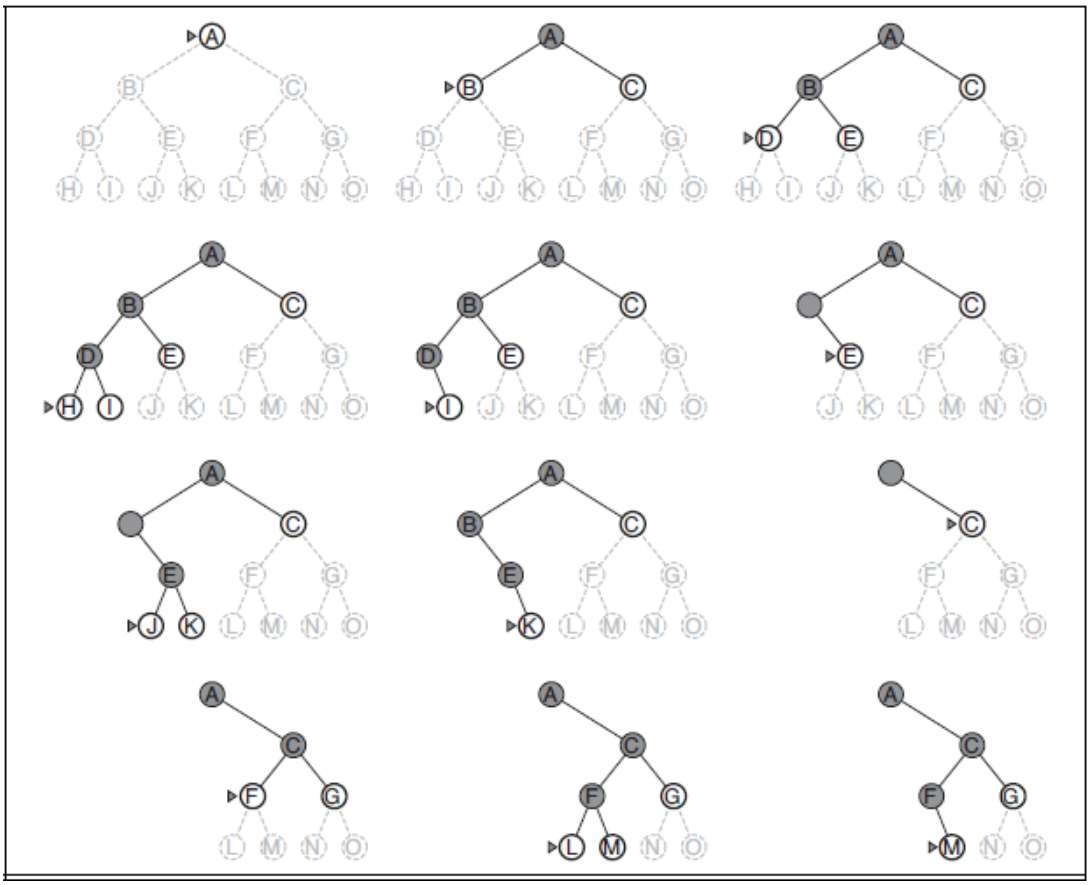



Note: Both nodes F and N have a cost of 15, we chose to expand the leftmost node first. We continue expanding until all remaining paths are greater than 21, the cost of G2

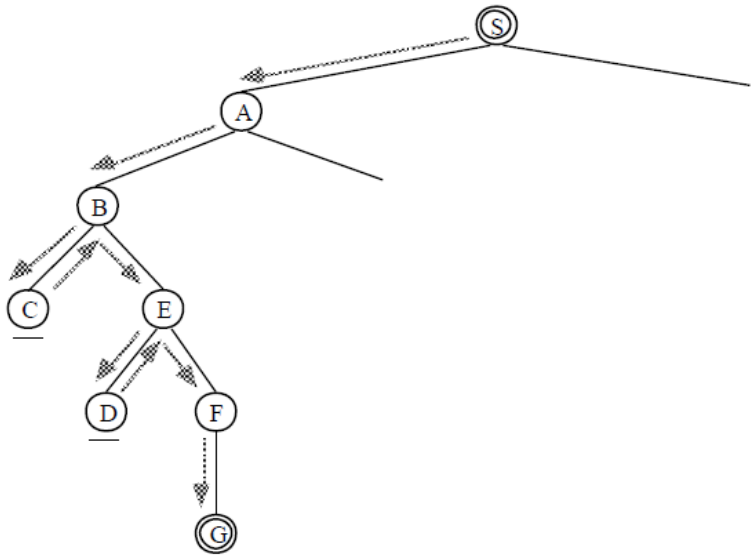
## 3. Depth-first search

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. This algorithm 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.



For example, to find path from S to G in the earlier graph,




Analysis

**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:

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

**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.

## 4. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

**Advantages:**

Depth-limited search is Memory efficient.

**Disadvantages:**

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Time Complexity **- $O(b^l)$** , Space Complexity- **O(bl)**

## 5. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found. This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

**Advantages:**

- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

**Disadvantages:**

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Analysis

**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 : $(d)b + (d-1)b^2 + \ldots + (1)b^d$ So, time complexity = $\mathbf{O(b^d)}$

**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.

For example,

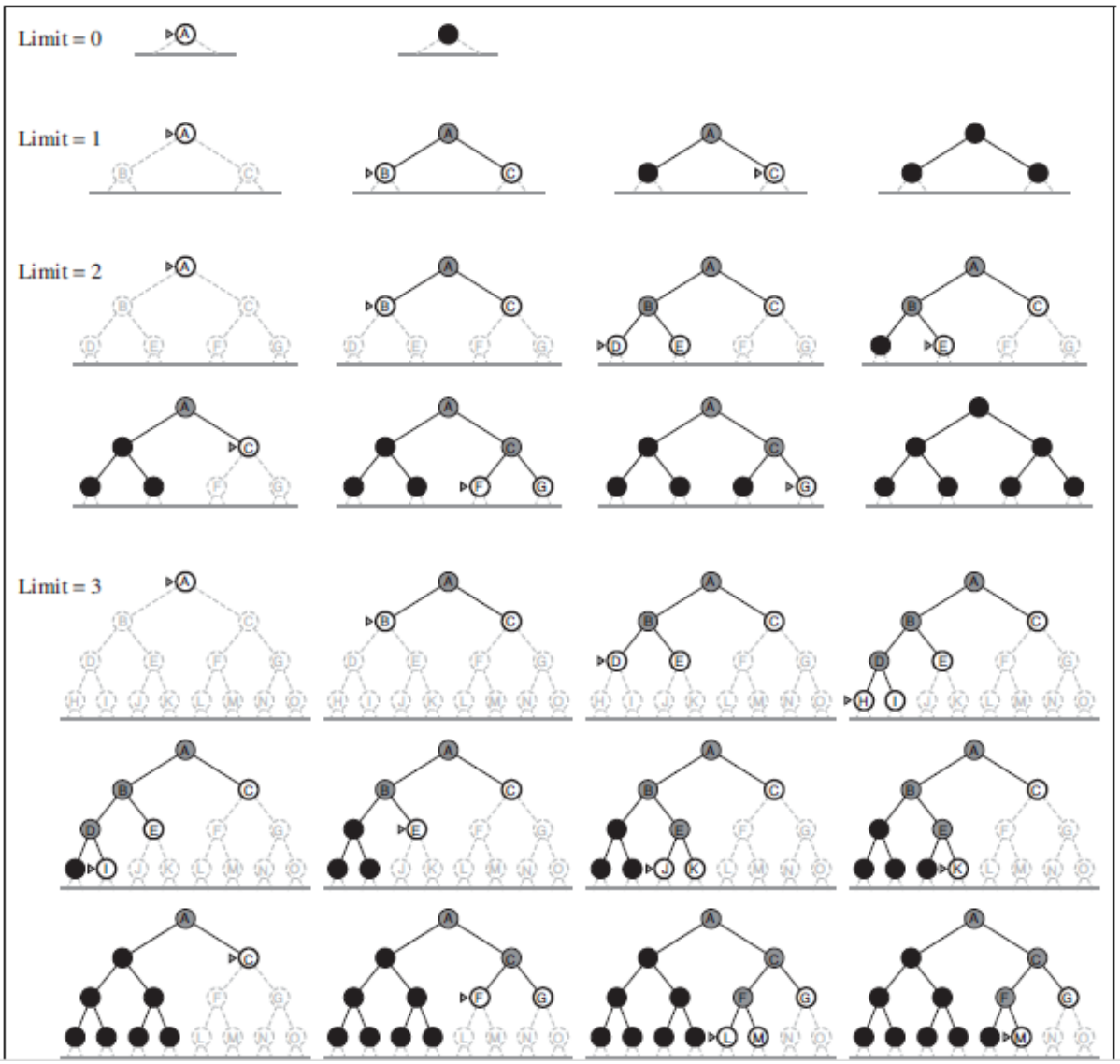



*Figure 2 Four iterations of iterative deepening search on a binary tree.*

## 6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these

two graphs intersect each other. Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

**Advantages:**

- Bidirectional search is fast.
- Bidirectional search requires less memory

**Disadvantages:**

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Analysis

**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

# Heuristic Search (Informed Search)

Heuristic search or informed search is a searching strategy that uses problem-specific knowledge beyond the definition of the problem itself and can find solutions more efficiently than uninformed strategy.

Informed or heuristic search algorithms include as a component of *f* a heuristic function, denoted h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. Heuristics are the foundation of strong AI and it is heuristics that distinguish AI methods from traditional computer science methods. Heuristics distinguish pure

algorithms from human problem-solving methods, which are inexact, intuitive, creative, sometimes powerful, and hard to define.

### Types of Informed Search Algorithms:

- Hill Climbing Search
- Best First Search (Greedy Search)
- A* Search

## 1. Hill Climbing Search

Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.

Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

In the above definition, mathematical optimization problems implies that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by the salesman.

**Generate and test approach**:

Hill climbing search is a variant of generate and test algorithm. The generate and test algorithm is as follows:

1. Generate possible solutions.
2. Test to see if this is the expected solution.
3. If the solution has been found quit else go to step 1.

Hence we call Hill climbing as a variant of generate and test algorithm as it takes the feedback from the test procedure. Then this feedback is utilized by the generator in deciding the next move in search space.

At any point in state space, the search moves in that direction only which optimizes the cost of function with the hope of finding the optimal solution at the end.

**Algorithm for Hill climbing Search :**

**Step 1:** Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.

**Step 2:** Loop until the solution state is found or there are no new operators present which can be applied to the current state.

a) Select a state that has not been yet applied to the current state and apply it to produce a new state.

b) Perform these to evaluate new state

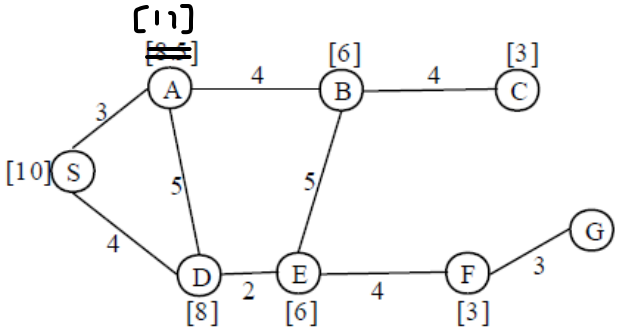
i. If the current state is a goal state, then stop and return success.

ii. If it is better than the current state, then make it current state and proceed further.

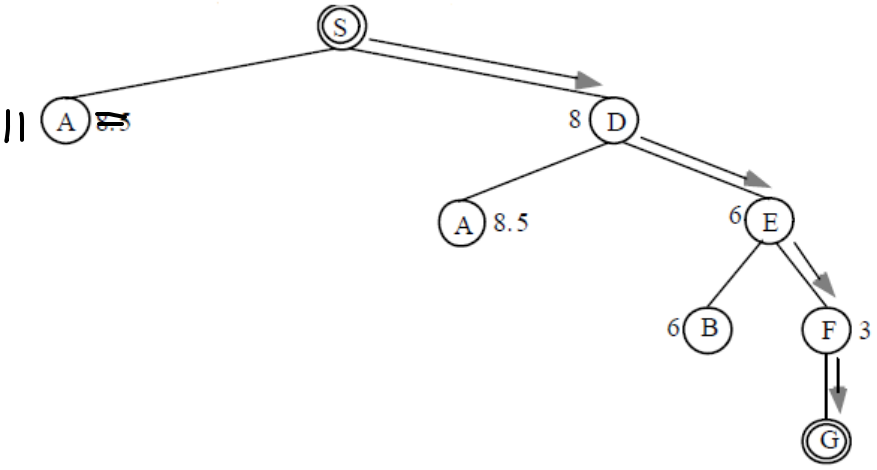
iii. If it is not better than the current state, then continue in the loop until a solution is found.

**Step 3:** Exit.

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:



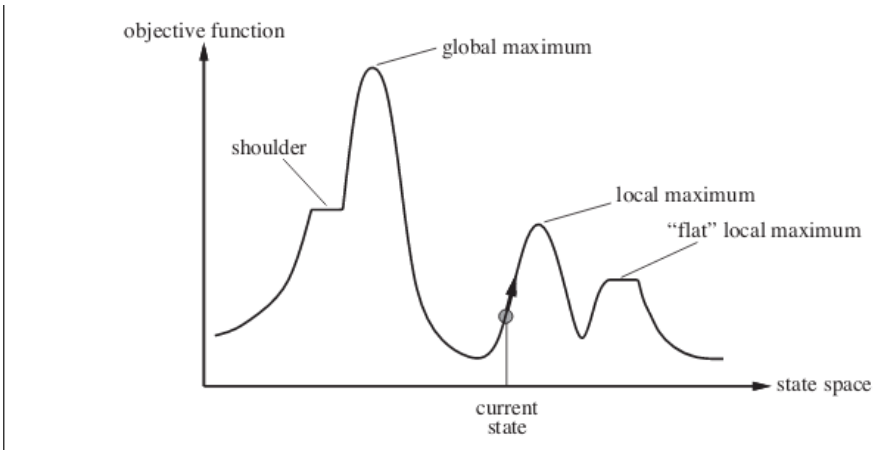

**State Space diagram for Hill Climbing**

State space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).

X-axis : denotes the state space ie states or configuration our algorithm may reach.

Y-axis : denotes the values of objective function corresponding to a particular state.

The best solution will be that state space where objective function has maximum value(global maximum).




**Different regions in the State Space Diagram:**

**Local maximum:** It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here the value of the objective function is higher than its neighbors.

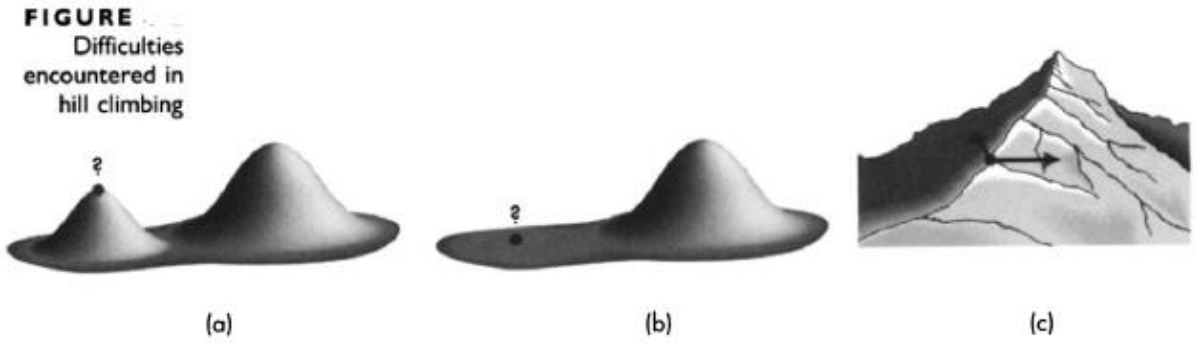
**Global maximum:** It is the best possible state in the state space diagram. This because at this state, objective function has highest value.

**Plateua/flat local maximum:** It is a flat region of state space where neighboring states have the same value.

**Ridge**: It is region which is higher than its neighbours but itself has a slope. It is a special kind of local maximum.

**Current state**: The region of state space diagram where we are currently present during the search.

**Shoulder:** It is a plateau that has an uphill edge.




**Problems with hill climbing**

**Local maximum**: At a local maximum all neighboring states have a values which is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.

To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.

**Plateau:** On plateau all neighbors have same value . Hence, it is not possible to select the best direction.

To overcome plateaus : Make a big jump. Randomly select a state far away from the current state. Chances are that we will land at a non-plateau region.

**Ridge:** The final problem with hill climbing involves ridges. A ridge involves a situation where although there is a direction in which we would like to move, none of the allowed steps (indicated by arrows in the figure) actually takes us in that direction.

To overcome Ridge: In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

## 2. Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

f(n)= h(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

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

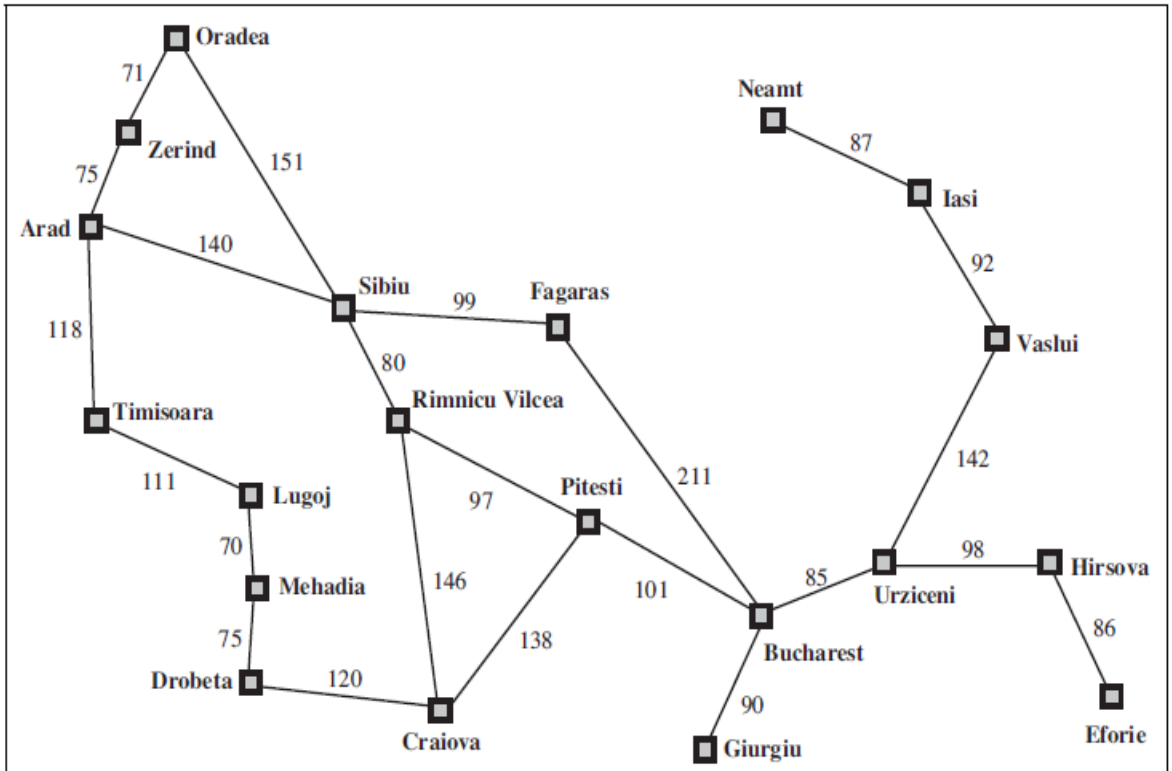



**Figure** A simplified road map of part of Romania.

| | | | |
|---|---|---|---|
| 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 $h_{SLD}$—straight-line distances to Bucharest.

The solution is as follows:

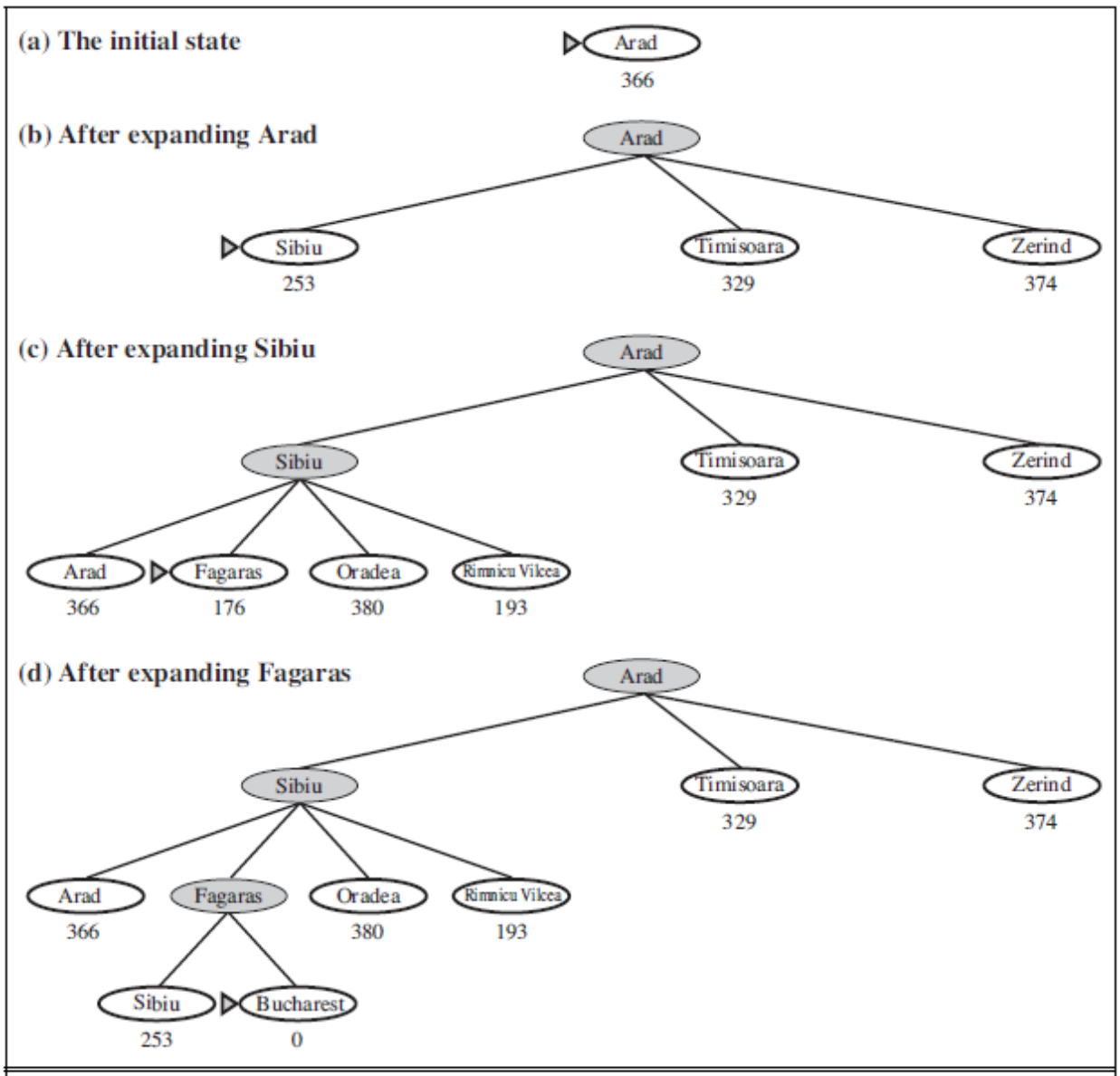
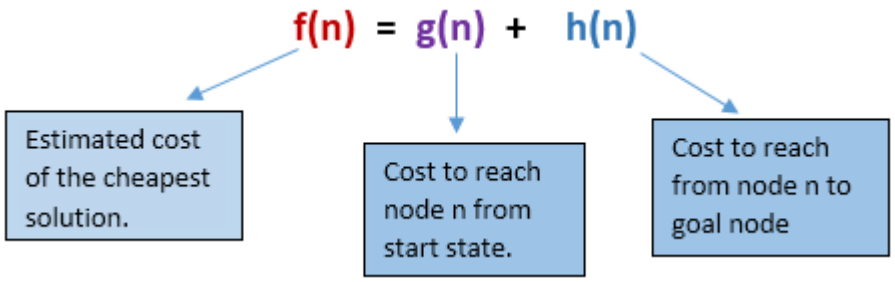



**Figure** Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic $h_{SLD}$. Nodes are labeled with their $h$-values.

## 3. A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.




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.

For example: From the graph of cities in earlier example, starting at Arad city, problem is to reach Bucharest:

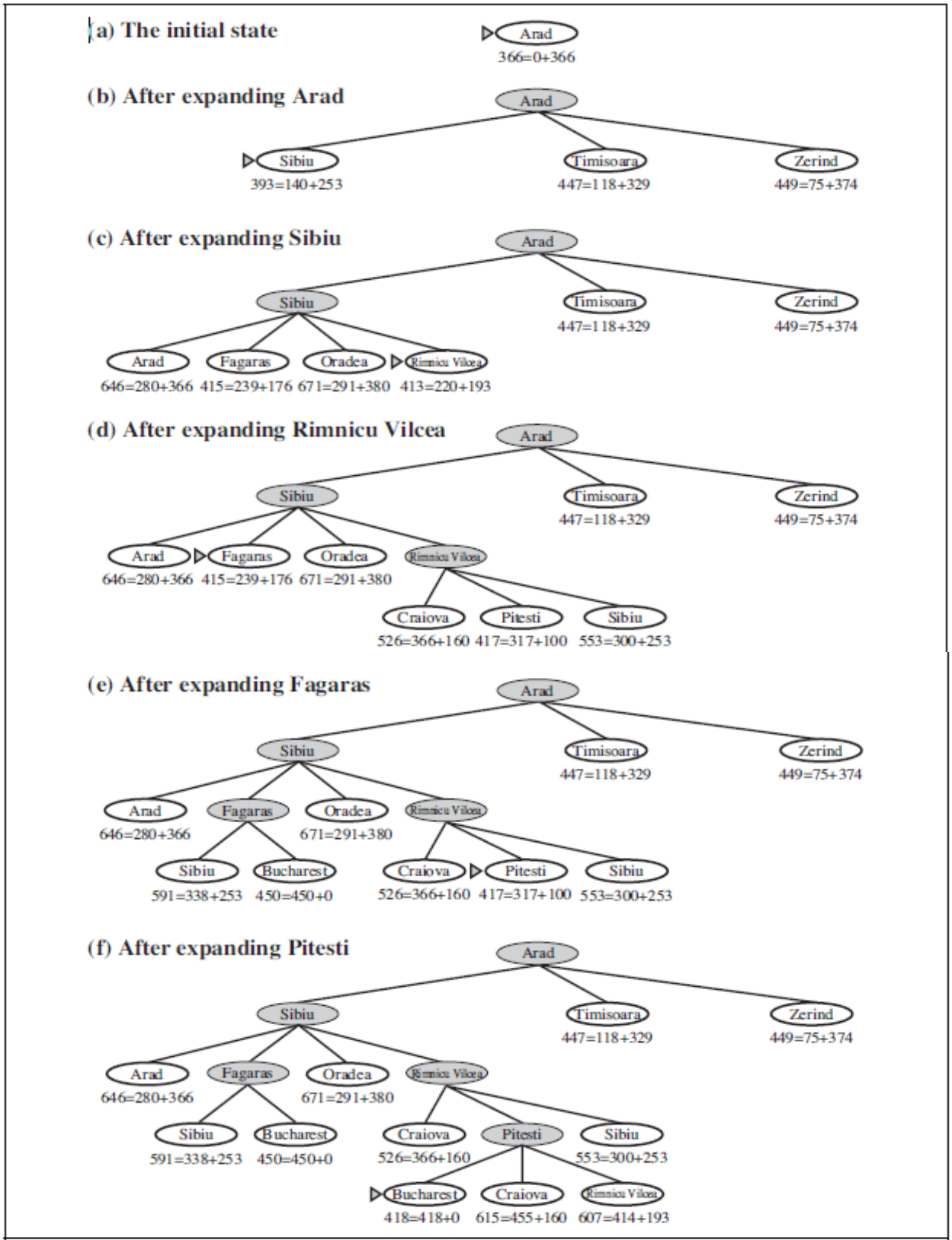



**Figure** Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The $h$ values are the straight-line distances to Bucharest

# Means End Analysis

Means End Analysis (MEA) is a problem-solving technique that has been used since the fifties of the last century to stimulate creativity. Means End Analysis is also a way of looking at the organisational planning, and helps in achieving the end-goals. In means-ends analysis, the problem solver begins by envisioning the end, or ultimate goal, and then determines the best strategy for attaining the goal in his current situation. If, for example, one wished to drive from Kalanki to Minbhawan in the minimum time possible, then, at any given point during the drive, one would choose the route that minimized the time it would take to cover the remaining distance, given traffic conditions, weather conditions, and so on.

With Means End Analysis, it is possible to control the entire process of problem solving. It starts from a predetermined goal, in which actions are chosen that lead to that goal.

Each action that is executed leads to the next action; everything is connected together in order to reach the end-goal. With the help of Means End Analysis, both forward and backward research can be done to determine where the inaction is occurring. This enables the larger parts of a problem to be solved first, to subsequently return to the smaller problems afterwards.

## Household Robot

Problem: Move a big desk with two small items on top of it from one room to another

| *Operator* | *Preconditions* | *Results* |
|---|---|---|
| PUSH(obj, loc) | at(robot, obj)^<br>large(obj)^<br>clear(obj)^<br>armempty | at(obj, loc)^<br>at(robot, loc) |
| CARRY(obj, loc) | at(robot, obj)^<br>small(obj) | at(obj, loc)^<br>at(robot, loc) |
| WALK(loc) | none | at(robot, loc) |
| PICKUP(obj) | at(robot, obj) | holding(obj) |
| PUTDOWN(obj) | holding(obj) | ¬holding(obj) |
| PLACE(obj1, obj2) | at(robot, obj2)^<br>holding(obj1) | on(obj1, obj2) |

**Fig.** *The Robot's Operators*

| | Push | Carry | Walk | Pickup | Putdown | Place |
|---|---|---|---|---|---|---|
| Move object | * | * | | | | |
| Move robot | | | * | | | |
| Clear object | | | | * | | |
| Get object on object | | | | | | * |
| Get arm empty | | | | | * | * |
| Be holding object | | | | * | | |

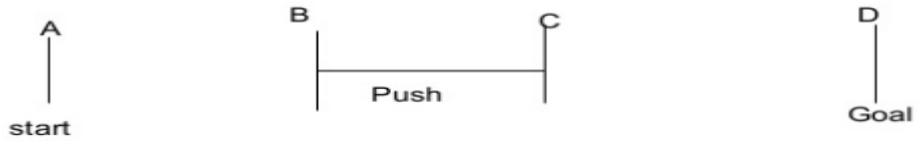
**Fig.** *A Difference Table*

Difference between start and end state: Location

**The ROBOT solves this problem as follows:**

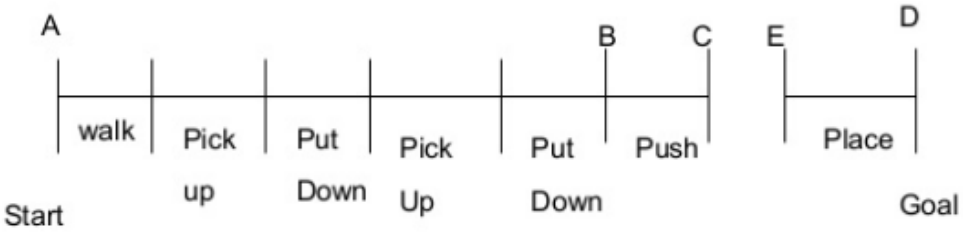
- The main difference between the initial state and the goal state would be the location of the desk.
- To reduce this difference, either PUSH or CARRY could be chosen.
- If CARRY is chosen first, its preconditions must be met. This results in two more differences that must be reduced:
  - The location of the robot and the size of the desk.
  - The location of the robot is handled by applying WALK, but there are no operators that can change the size of an object, So this path leads to a dead-end

Following the other branch, we attempt to apply PUSH. Figure below shows the robot's progress at this point




- Now the differences between A and B and Between C and D must be reduced.
- PUSH has preconditions:
  - the robot must be at the desk, and
  - the desk must be clear.
- The robot can be brought to the correct location by using WALK.
- And the surface of the desk can be cleared by two uses of the PICKUP.
- But after one PICKUP, An attempt to do the second results in another difference (the arm must be empty).
- PUTDOWN can be used to reduce the difference

- Once PUSH is performed, the problem state is closed to the goal state, but not quite. The object must be placed back on the desk. PLACE will put them there. The progress of the robot at this point is as shown in figure below:



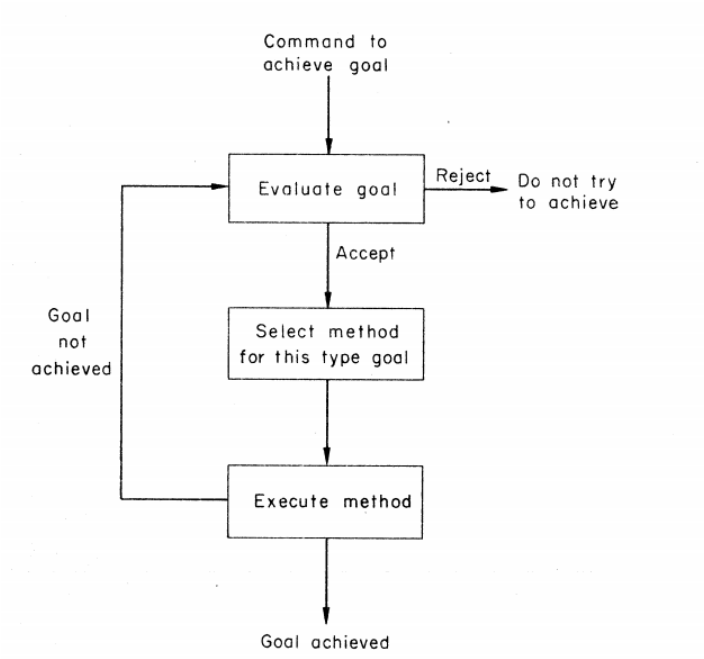

- The final difference between C and E can be reduced by using WALK to get the robot back to the object followed by PICKUP and CARRY

## General Problem Solving (GPS): Problem solving agents

- The General Problem Solver (GPS) was an AI program proposed by Herbert Simon, J.C. Shaw, and Allen Newell in 1959.
- As the name implies, it was intended to solve nearly any problem and intended to work as a universal problem solver machine.
- It was the first useful computer program that came into existence in the AI world.
  - The goal was to make it work as a universal problem-solving machine.
    - Of course there were many software programs that existed before, but these programs performed specific tasks.
    - GPS was the first program that was intended to solve any general problem.
- GPS was supposed to solve all the problems using the same base algorithm for every problem.
- Of course, at that time, programming GPS was difficult. The authors created a new language called Information Processing Language (IPL) in order to more efficiently program GPS.
  - GPS operates on problems that can be formulated in terms of *objects* and *operators*
- An **operator** is something that can be applied to certain objects to produce different objects (as a saw applied to logs to produces boards).
- The objects can be characterized by the *features* they possess, and by the *differences* that can be observed between pairs of objects.

- **Operators** may be restricted to apply certain kinds of objects; and there may be operators that are applied to several objects as inputs, producing one or more objects as output(as the operation of adding two numbers produces a third number, their sum)
- Constructing a computer program can also be described as a problem in same terms. Here, the objects are the computer memory; the operators are computer instructions that alters the memory contents. A program is a sequence of memory operators that transforms one state of memory to another; the programming problem is to find such sequence when certain features of the initial and terminal states are specified.

The executive organization of GPS is shown in the figure below. With each goal type is associated a set of methods related to achieving the goals of that type. When an attempt is made to achieve a goal, it is first evaluated to see whether it is worthwhile achieving and whether achievement seems likely. If so, one of the method is selected and executed. This either leads to success or to a repetition of the loop.



## Problem Reduction

We already know about the divide and conquer strategy, a solution to a problem can be obtained by decomposing it into smaller sub-problems. Each of this sub-problem can then be solved to get its sub solution. These sub solutions can then recombined to get a solution as a whole. That is called is **Problem Reduction**. This method generates arc which is called as **AND** 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.


# Constraint Satisfaction Problem

Constraint satisfaction problem (CSP) consists of

• a set of variables,

• a domain for each variable, and

• a set of constraints.

The aim is to choose a value for each variable so that the resulting possible world satisfies the constraints; we want a model of the constraints. A finite CSP has a finite set of variables and a finite domain for each variable. Many of the methods considered in this chapter only work for finite CSPs, although some are designed for infinite, even continuous, domains.

## CryptArithmetic Problem

CryptArithmetic or verbal arithmetic is a class of mathematical puzzles in which the digits are replaced by letters of the alphabet or other symbols. Usually it is required that each letter would be replaced by a unique digit. Each letter having different value from the other letters.

For Example: SEND + MORE = MONEY

SEND + MORE = MONEY is a classical ``crypto-arithmetic'' puzzle: the variables S, E, N, D, M, O, R, Y represent digits between 0 and 9, and the task is finding values for then such that the following arithmetic operation is correct:

| | | | | |
|---|---|---|---|---|
| | S | E | N | D |
| | M | O | R | E |
| M | O | N | E | Y |

Solution:

To begin, start in the 5th column. Since 9999 + 9999 < 20000, we must have M = 1.

Then go to the 4th column. Since 999 + 999 < 2000, we either have 1 + S + 1 = O + 10, or S + 1 = O + 10, meaning S = O + 8 or S = O + 9, and O = 0 or 1. Since S is a single digit, and M = 1, we must have O = 0.

In the 3rd column, since E cannot equal N, we cannot have E + 0 = N. Thus we must have 1 + E + 0 = N. Since N cannot be 0, we must also have E less than 9. So there cannot be carryover in this column, and the 2nd column must have carryover.

Returning to the 4th column (which has no carryover from the 3rd), we must have S + 1 = 10, which means S = 9.

Now we know 1 + E = N, and there must be carryover from the 2nd column. So we have two cases: N + R = E + 10, or N + R + 1 = E + 10. We can substitute 1 + E = N in both cases to get (1 + E) + R = E + 10 –> R = 9 (but 9 is already taken), or we have 1 + E + R + 1 = E + 10 –> R = 8. So we must have R = 8.

Now in the units column D + E = Y, and it must have carryover. Since Y cannot be 0 or 1, we need D + E ≥ 12. Since 9 and 8 are taken for S and R, we can have 5 + 7 = 12 or 6 + 7 = 13. So either D = 7 or E = 7.

If E = 7, then E + 1 = N so N = 8–which is not possible since R = 8. So we must have D = 7, meaning E is either 5 or 6.

If E = 6, then N = 7 which is not possible as D = 7. So we must have E = 5 and N = 6. This means D + E = 7 + 5 = 12, and thus Y = 2.

So we have solved for all the letters!

SEND + MORE = 9567 + 1085 = 10652.

# Game Playing and AI

## Game Playing

In Uninformed Searches and Informed Search, there are problems or puzzles that has a specified start state and goal state. Different operators are used to transform problem states and to eventually reach the goal. The only obstacle to the progress are the immensity of the associated state space.

Game playing introduces an additional challenge: an adversary who is trying to impede the advancement. Nearly all games include one or more opponents who are actively trying to defeat each other. In fact, much of the excitement in game playing—whether in a friendly card game or a tension filled night of poker— is derived from the risk of losing.

## Game Tree and MiniMax Evaluation

To evaluate the effectiveness, or "goodness," of a move in a game, you can pursue that move and see where it leads. In other words, you can play "what if" to ask, "If I make this move, how will my opponent respond, and then how will I counter?" After charting the consequences of a move, you can evaluate the effectiveness of the original move. You are doing this to determine whether a move improves your chances of winning the game. You can use a structure called a game tree for this evaluation process. In a game tree, nodes represent game states and branches represent moves between these states. A game tree for tic-tac-toe is shown in following figure:

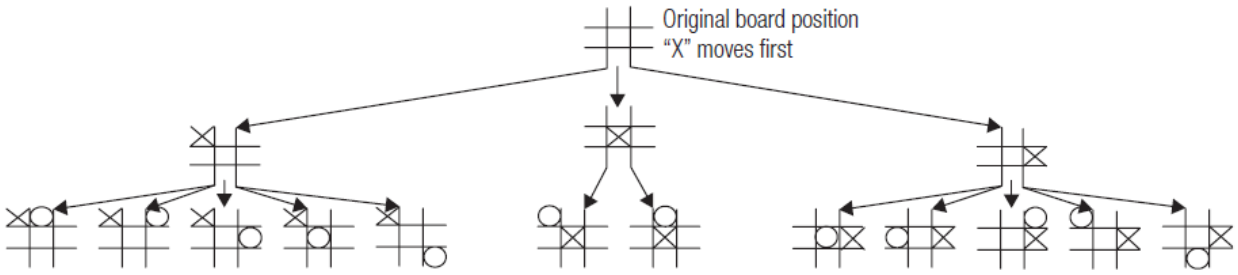



**Figure**
Game tree for tic-tac-toe.

## Heuristic Evaluation

When a game tree has been expanded to the end of a game, measuring the goodness of a move is trivial. If the move resulted in a win, then it was good; if a loss ensued, it was not as good. Combinatorial explosion, however, prevents a complete evaluation for all but the most basic games. For more complicated games, you need to use heuristic evaluation. Recall that a heuristic

is a set of guidelines that usually works to solve a problem. Heuristic evaluation is the process whereby a single number is attached to the state of a game; those states more likely to lead to a win are accorded larger numbers. You can use a heuristic evaluation to reduce the number of computations required to solve a problem complicated by combinatorial explosion.

You can use heuristic evaluation to solve the game of tic-tac-toe. Let N(X) equal the number of rows, columns, and diagonals that X could possibly complete, as shown in Figure alongside. N(O) is similarly defined for moves the O player can complete. When X is in the upper-left corner (with O in the adjacent space to the right), it can complete three possible moves: the leftmost column, and both diagonals. The heuristic evaluation of a game position E(X) is defined as N(X) – N(O). Hence, E(X) for the upper-left position illustrated in figure alongside is 3 – 1 = 2.

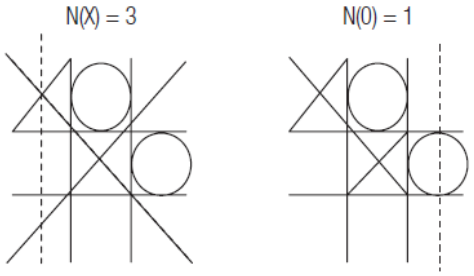



**Figure**
Heuristic evaluation in tic-tac-toe.

The exact number that a heuristic attaches to a game position is not that important. What does matter, however, is that more advantageous positions (better positions) are accorded higher heuristic values. Heuristics evaluation provides a strategy for dealing with combinational explosion.

Heuristic evaluation provides a tool to assign values to leaf nodes in the game. Figure below shows the game tree again where the heuristic evaluation function has been added. The X player would pursue those moves with the highest evaluation (that is 2 in this game) and avoid those game states that evaluate to 0 (or worse). Heuristic evaluation has permitted the X player to identify advantageous moves without exploring the entire game tree.

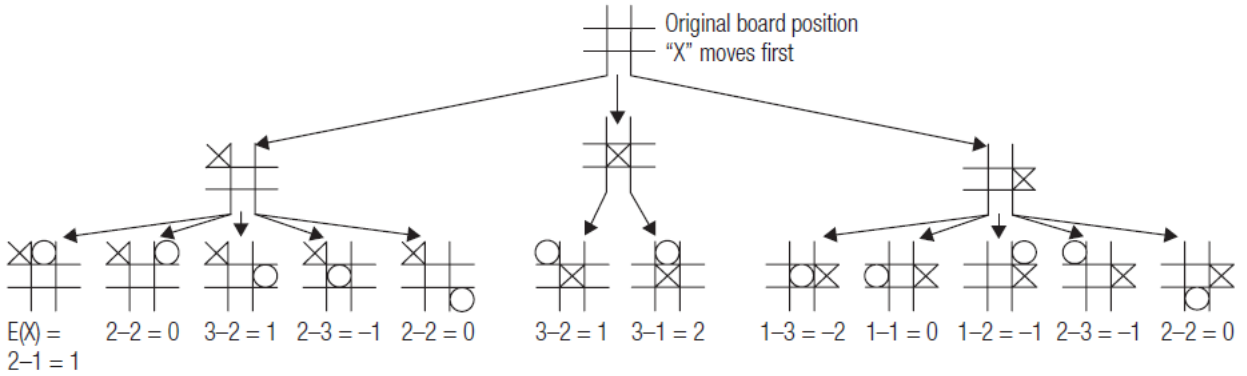



**Figure**
Heuristic evaluation applied to a game tree E(X) = N(X) – N(0).

We now require a technique so that these heuristic values can "percolate" upward so that all of this information can be used by the X player before she makes her first move. Minimax evaluation will provide just such a technique.

## Minimax Evaluation

In tic-tac-toe, the X player can use heuristic evaluation to find the most promising path to victory, the O player, however, can block that path in any move. A game between two experienced players will always end in a draw (unless one player makes a mistake). Instead of following the quickest path to victory, X needs to find the path that leads to victory even if O blocks it. Minimax evaluation is a technique that identifies such a path (when it exists), and is helpful in most two person games.

The two players in a two-person game are traditionally referred to as Max and Min, with Max representing the player who attempts to maximize the heuristic evaluation and Min representing the player seeking to minimize it. Players alternate moves, with Max generally moving first. Assume that heuristic values have been assigned to each of the moves possible for any player in a given position.

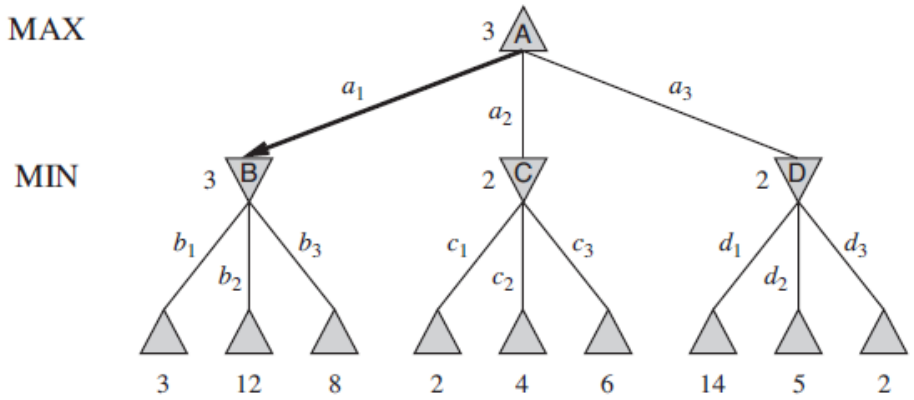



**Figure** A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is $a_1$, because it leads to the state with the highest minimax value, and MIN's best reply is $b_1$, because it leads to the state with the lowest minimax value.

The value of the Max node is the maximum of the values in either of its immediate successor nodes; hence, the value of the Max node shown in Figure above is 3. Keep in mind that Max and Min are opponents. A move that is good for Max is bad for Min.

Additionally, all values in a game tree are considered from Max's vantage point. The Min player always makes the move with the minimum value attached to it, because the Min player is trying to minimize the value that accrues to Max. So the value of the Min node shown in figure above is 3, 2 and 2 respectively from left to right, because this is the minimum of all values present for the successor nodes.

## Alpha beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:

    a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

    b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

- 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.

Algorithm:

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if alpha >= beta:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if alpha >= beta:
                break
        return bestVal
// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```

# Game Theory

Game theory is the study of mathematical models of strategic interaction among rational decision-makers. It has applications in all fields of social science, as well as in logic, systems science and computer science. In some respects, game theory is the science of strategy, or at least the optimal decision-making of independent and competing actors in a strategic setting.

The key pioneers of game theory were mathematician John von Neumann and economist Oskar Morgenstern in the 1940s. Mathematician John Nash is regarded by many as providing the first significant extension of the von Neumann and Morgenstern work. Using game theory, real-world scenarios for such situations as pricing competition and product releases (and many more) can be laid out and their outcomes predicted.

## Prisoner's Dilemma

This so-called Prisoner's Dilemma was first formulated in game-theoretical terms by Merrill Floyd and Melvin Dresher at the RAND Corporation in 1950. The Prisoner's Dilemma is the most well-known example of game theory. Consider the example of two criminals arrested for a crime. Prosecutors have no hard evidence to convict them. However, to gain a confession, officials remove the prisoners from their solitary cells and question each one in separate chambers. Neither prisoner has the means to communicate with each other. Officials present four deals, often displayed as a 2x2 box.

| | | Prisoner B | |
|---|---|---|---|
| | | Cooperate (remain silent) | Defect (betray partner) |
| Prisoner A | Cooperate (remain silent) | A: 1 year<br>B: 1 year | A: 10 years<br>B: 0 years |
| | Defect (betray partner) | A: 0 years<br>B: 10 years | A: 5 years<br>B: 5 years |

**Figure**
Payoff matrix for the Prisoner's Dilemma.

Assume that the two players (that is, the prisoners) in this game are rational and want to minimize their jail sentences. Each prisoner has two choices: cooperate with their partner in crime and remain silent, or defect by confessing to the police in return for a lesser sentence. You might notice that this game differs in an important aspect from the games discussed earlier in this chapter. To determine a course of action in other games, you need to know your opponent's course of action. For example, if you are the second person to move in a

game of tic-tac-toe, you need to know where the other player has placed the initial X. This is not the case in the Prisoner's Dilemma. Suppose you are the A player and that you choose to defect. However, the B player decides to remain loyal and chooses the cooperate-with-partner strategy. In this case, your decision results in no prison time as opposed to a one-year term you would have received if you had chosen instead to also cooperate. If your partner chooses to defect, your outcome is still superior if you choose to defect. In game-theoretic terms, defecting is a **dominant strategy**. Because you assume that your opponent in this game is rational, he will arrive at the same strategy.

The strategies {Betray, Betray} on the part of the two participants are referred to as a **Nash equilibrium**. This strategy is named after John F. Nash who won the Nobel Prize in Economics for his groundbreaking work in game theory. A change in strategy by either player results in a lesser return to them (i.e., more jail time).

As shown in Figure, if each player acts more on faith than rationality (faith that their partners would remain loyal) then the total payoff would exceed the total of 10 prison years accorded by the Nash equilibrium of {Defect, Defect}. This strategy of {Cooperate, Cooperate} yields the best possible outcome in terms of total payoff to the two players. This optimal strategy is referred to as a **Pareto Optimal**.

## Game of Chance

A game of chance is a game whose outcome is strongly influenced by some randomizing device. Common devices used include dice, spinning tops, playing cards, roulette wheels, or numbered balls drawn from a container. A game of chance may be played as gambling if players wage money or anything of monetary value.

Some games of chance may also involve a certain degree of skill. This is especially true where the player or players have decisions to make based upon previous or incomplete knowledge, such as blackjack. In other games like roulette the player may only choose the amount of bet and the thing he wants to bet on; the rest is up to chance, therefore these games are still considered games of chance with small amount of skills required.

**References**

Lucci, S., & Kopec, D. (2016). *ArtificiAl intelligence in the 21st century: A Living Introduction.* Virginia: Mercury Learning and Information.

Poole, D. L., & Mackworth, A. K. (2010). *Artificial Intelligence: Foundations of Computational Agents.* New York: CAMBRIDGE UNIVERSITY PRESS.

Rich, E., Knight, K., & Nair, S. B. (2009). *Artificial Intelligence.* Delhi: Tata McGraw-Hill.

Russell, S., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach.* New Jersey: Pearson Education, Inc.