

A problem is defined by:

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

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

Problem Solving by Searching

Problem solving is the systematic search through a range of possible states in order to reach Some predefined goal. A problem solving has following steps.

Four general steps in problem solving:

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

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 search problem is to find a path from a city J to a city G

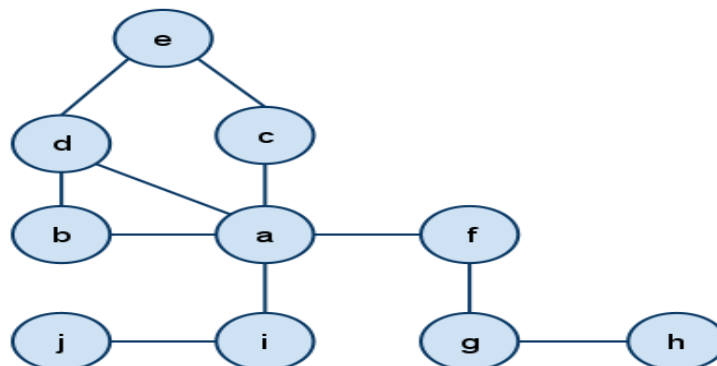


Fig. graph representation of a map

There are two broad classes of search methods:

- **Uninformed (or blind) search methods**
- **Heuristic (informed) search methods**

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

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

Measuring problem Solving Performance

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

- **Completeness**

- An algorithm is said to be complete if it definitely finds solution to the problem, if exist.

- **Time Complexity**

- How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**

- **Space Complexity**

- How much space is used by the algorithm? Usually measured in terms of the **maximum number of nodes in memory at a time**

- **Optimality/Admissibility**

- If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

Uninformed search (Blind search)

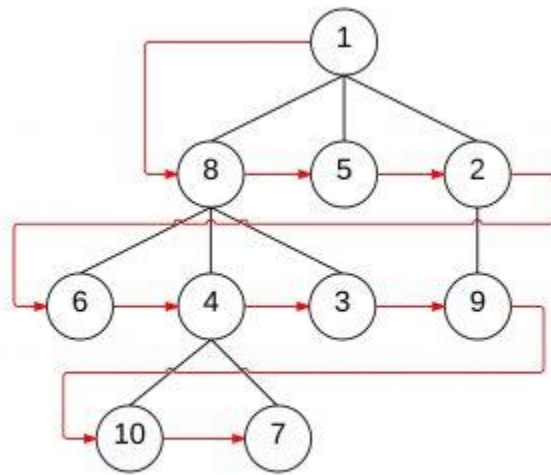
It has only goal state and initial state. No other additional information is provided like the cost of path and no domain specific information. It search the potential solution in some arbitrary direction.

Breadth First Search (BFS)

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

Fringe is implemented as a FIFO queue.

Note: Fringe is a generated but unexpanded node. Fringe is the border obtained by connecting all the nodes at the end of each path.



Order: 1, 8, 5, 2, 6, 4, 3, 9, 10, 7

BFS evaluation

Completeness:

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

Time complexity:

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

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

Space complexity:

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

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

Optimality (i.e., admissible):

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

BFS pseudocode

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

 remove the head u of Q

 mark and enqueue all (unvisited) neighbours of u

Two lessons:

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

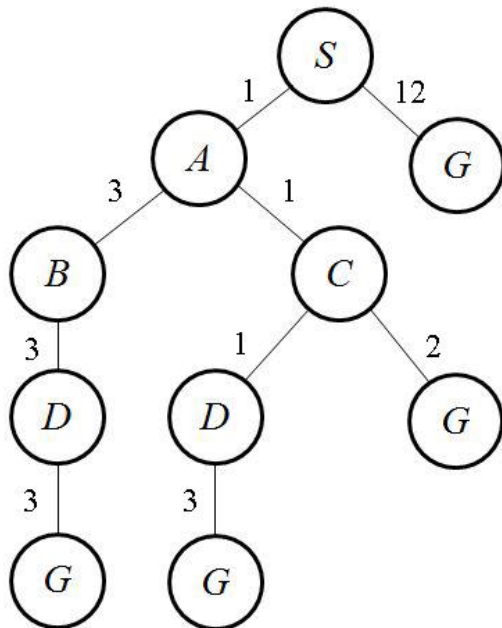
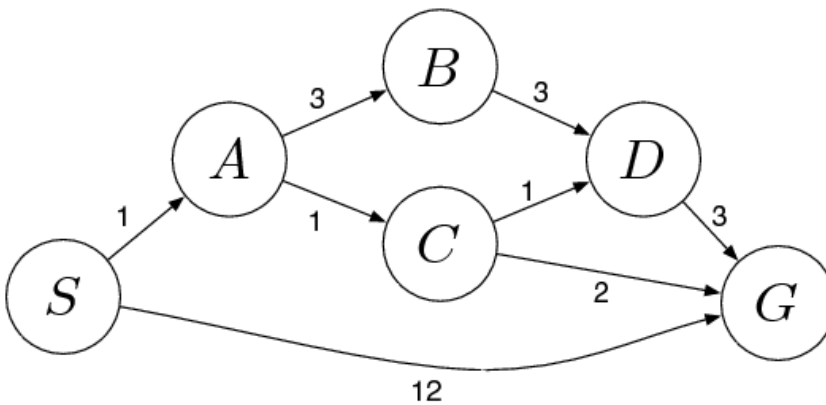
DEPTH	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	107	19 minutes	10 gigabytes
8	109	31 hours	1 terabyte
10	1011	129 days	101 terabytes
12	1013	35 years	10 petabytes
14	1015	3523 years	1 exabyte

Uniform Cost Search

Uniform Cost Search is the best algorithm for a search problem, which does not involve the use of heuristics. It can solve any general graph for optimal cost. Uniform Cost Search as it sounds searches in branches which are more or less the same in cost.

Uniform Cost Search again demands the use of a priority queue. Recall that Depth First Search used a priority queue with the depth upto a particular node being the priority and the path from the root to the node being the element stored. The priority queue used here is similar with the priority being the cumulative cost upto the node. Unlike Depth First Search where the maximum

depth had the maximum priority, Uniform Cost Search gives the minimum cumulative cost the maximum priority.



Depth First Search (DFS)

Looks for the goal node among all the children of the current node before using the sibling of this node

Expand deepest unexpanded node

Fringe is implemented as a LIFO queue (=stack)

Optimal (i.e., admissible):

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

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
    For each u ∈ G
        DFS(G, u)
}
```

Depth-limited search

It makes arbitrary depth limit l and makes DFS upto that limit. It means nodes at depth l are treated as if they have no successor and after this limit no nodes are visited.

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

If $l < d$ then incompleteness results.

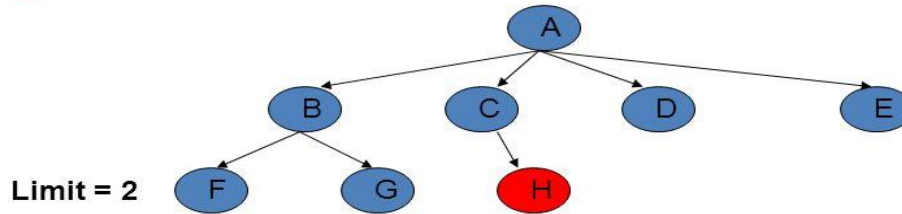
If $l > d$ then not optimal.

Time complexity: $O(b^l)$

Space complexity: $O(b^l)$

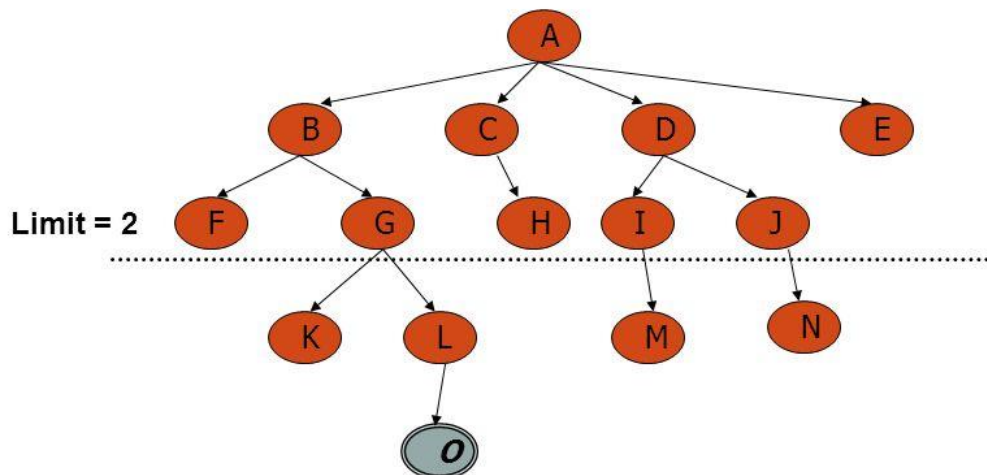
Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,



Depth-Limited Search (DLS)

- DLS algorithm returns **Failure (no solution)**
- The reason is that the goal is beyond the limit (Limit =2): the goal depth is (d=4)



Iterative deepening search

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

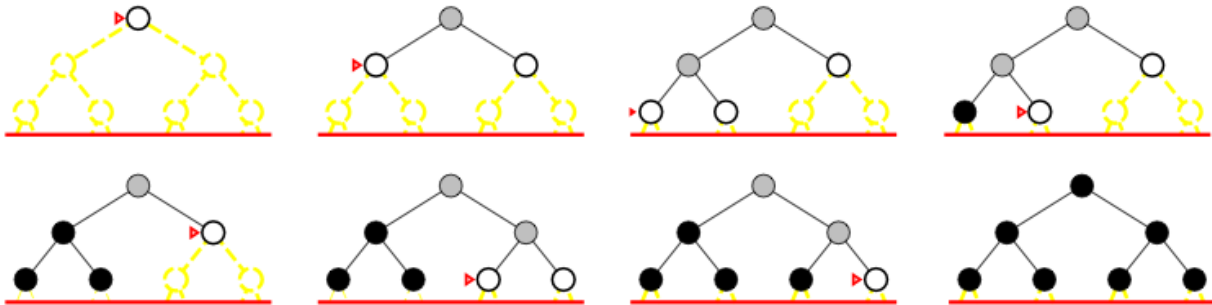
Limit = 0



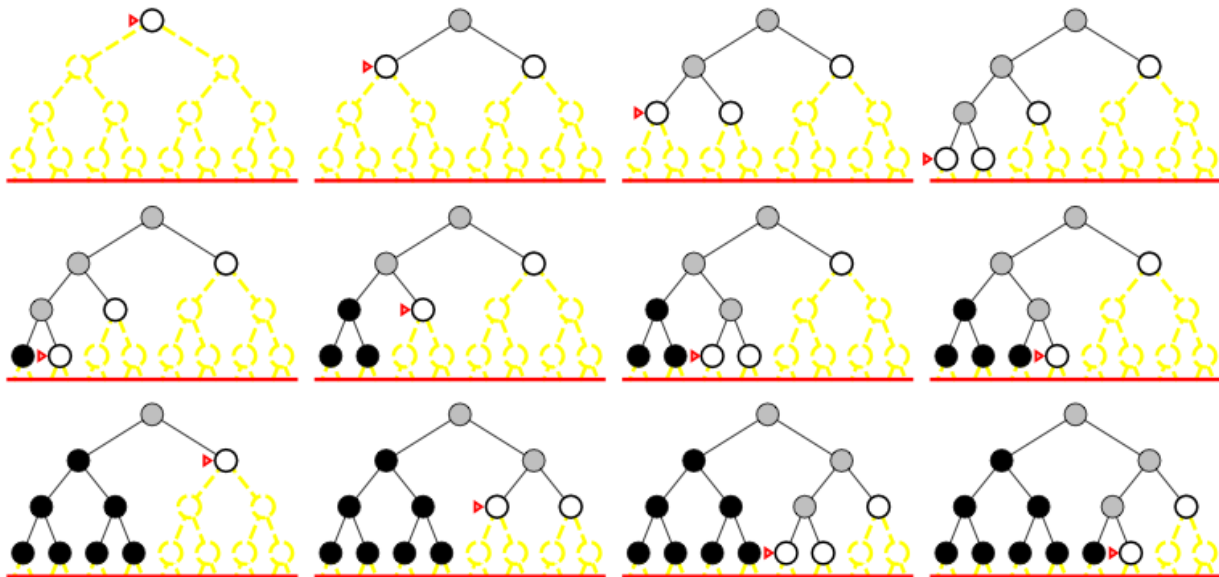
Limit = 1



Limit = 2



Limit = 3



ID search evaluation

Completeness:

- YES (no infinite paths)

Time complexity:

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

– Node generation:

level d: once

level d-1: 2

level d-2: 3

...

level 2: d-1

level 1: d

– Total no. of nodes generated:

$$d \cdot b + (d-1) \cdot b^2 + (d-2) \cdot b^3 + \dots + 1 \cdot b^d = O(b^d)$$

Space complexity:

- It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.

– Total no. of nodes in memory:

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

Optimality:

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

Note:-

Notice that BFS generates some nodes at depth $d+1$, whereas IDS does not. The result is that IDS is actually faster than BFS, despite the repeated generation of node.

Example:

Num. of nodes generated for $b=10$ and $d=5$ solution at far right

$$N(\text{IDS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

Drawbacks of uniformed search :

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

Informed Search (Heuristic Search)

In addition to initial state and goal state, Heuristic Search uses domain-dependent (heuristic) information like estimated path cost from initial state to goal state in order to search the space more efficiently. Heuristic function is the estimated cost that defines the goodness of a node to be expanded next. This allows to search in best path. It does this by deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order. It can discard or prune certain nodes from search space also.

Ways of using heuristic information:

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

Best-First Search

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

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

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

There is a whole family of best-first search strategies, each with a different evaluation function. Typically, strategies use estimates of the cost of reaching the goal and try to minimize it.

Special cases: based on the evaluation function.

- **Greedy best-first search**
- **A*search**

Greedy Best First Search

Greedy best first search expands the node that seems to be closest to the goal node. Heuristic function is used to estimate which node is closest to the goal node. It does not always give optimal solution. Optimality of the solution depends upon how good our heuristic is. Once the greedy best first search takes a node, it never looks backward.

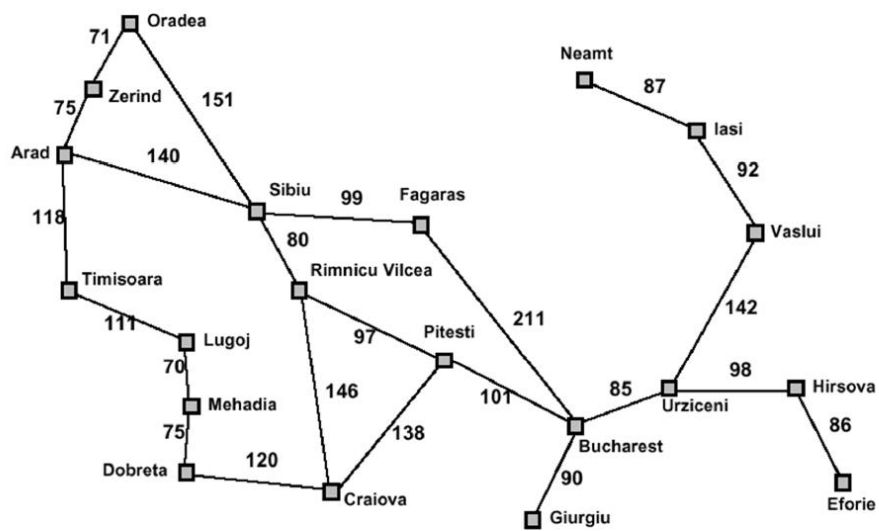
Heuristic used by Greedy Best First Search is:

$$f(n) = h(n)$$

Where $h(n)$ is the estimated cost of the path from node n to the goal node.

For example consider the following graph:

□ Romania with step costs in km

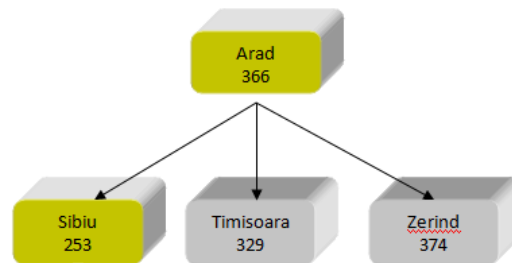


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

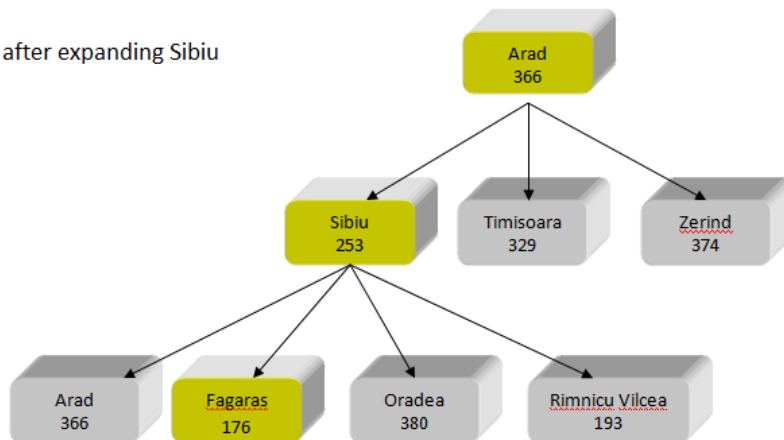
(a) the initial state



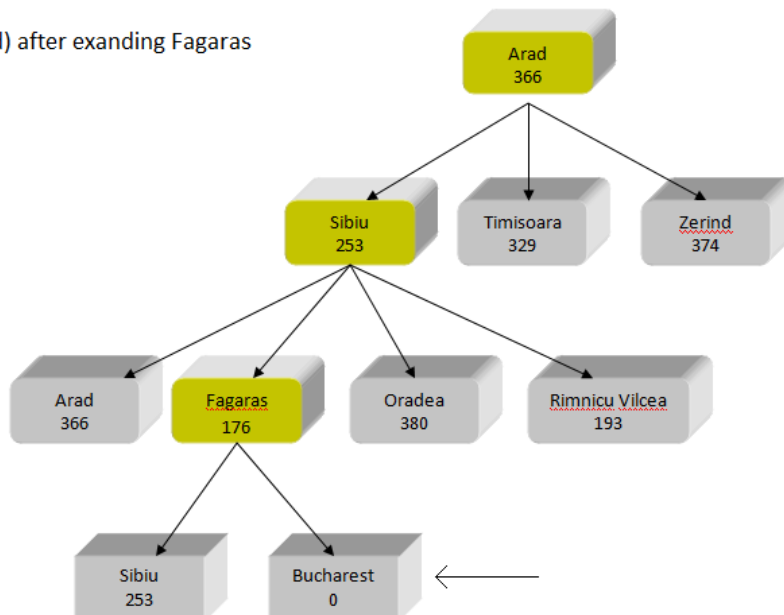
(b) after expanding Arad



(c) after expanding Sibiu



(d) after expanding Fagaras



Evaluation Greedy Search

Completeness:

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

Optimality:

– Greedy search is not optimal. Same as DFS.

Time complexity:

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

Space Complexity:

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

A * Search

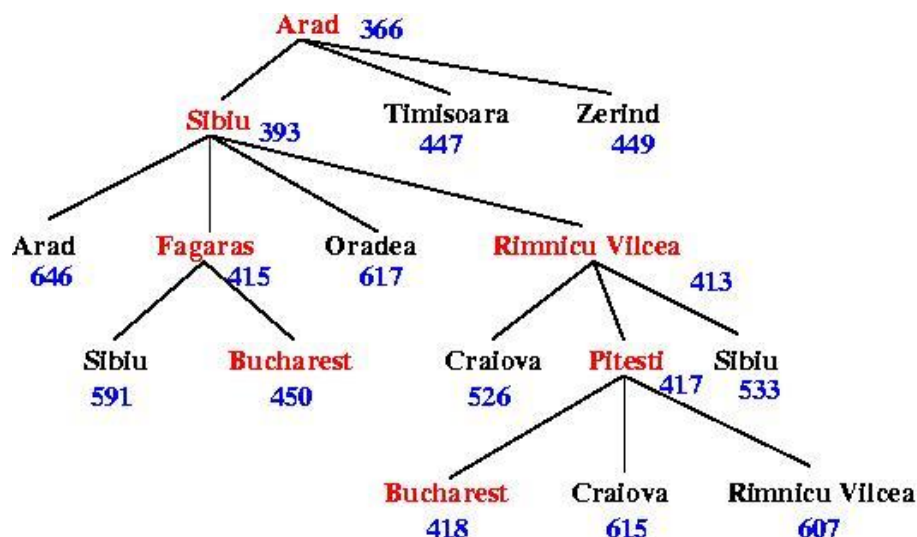
Like all informed search algorithms, it first searches the routes that appear to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account (the $g(n)$ part of the heuristic is the cost from the start, and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

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

Where:

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

For example consider the following graph



Evaluating A* Search:

Completeness:

- Yes A* search always gives us solution

Optimality:

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

Time complexity:

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

Space complexity:

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

Admissible heuristic

A heuristic function is said to be admissible heuristic if it never overestimates the cost to reach to the goal.

i.e

$$h(n) \leq h^*(n)$$

Where

$h(n)$ = Estimated cost to reach to the goal node from node n

$h^*(n)$ = Actual cost to reach to the goal node from node n

Formulating admissible heuristics:

n is a node

h is a heuristic

$h(n)$ is cost indicated by h to reach a goal from n

$C(n)$ is the actual cost to reach a goal from n

h is admissible if

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

For Example: 8-puzzle

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

$h1(n)$ = number of misplaced tiles

$h2(n)$ = sum of the distance of the tiles from their goal position (not diagonal).

$h1(S) = ?$ 8

$h2(S) = ?$ $3+1+2+2+2+3+3+2 = 18$

$hn(S) = \max\{h1(S), h2(S)\} = 18$

Initial
configuration

1	2	3
5	6	
7	8	4

Final
configuration

1	2	3
5	8	6
	7	4

Consistency (Monotonicity)

A heuristic is said to be consistent if for any node N and any successor N' of N , estimated cost to reach to the goal from node N to goal node is less than the sum of step cost from N to N' and estimated cost from node N' to goal node.

i.e

$$h(n) \leq d(n, n') + h(n')$$

Where

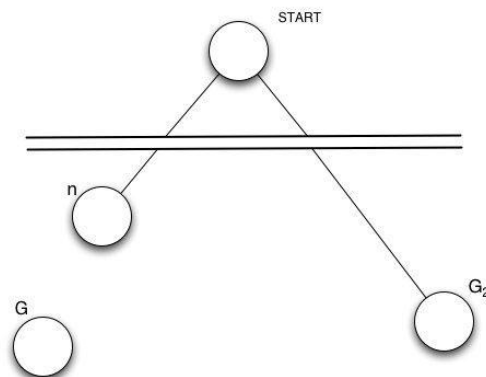
$h(n)$ = Estimated cost to reach to the goal node from node n

$d(n, n')$ = actual cost from n to n'

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

Suppose suboptimal goal G_2 in the queue.

Let n be an unexpanded node on a shortest to optimal goal G and C^* be the cost of optimal goal node.



$$f(G_2) = h(G_2) + g(G_2)$$

$$f(G_2) = g(G_2) \text{ since } h(G_2) = 0$$

$$f(G_2) > C^* \dots\dots\dots(1)$$

Again

Since $h(n)$ is admissible, It doesn't not overestimates the cost of completing the solution path.

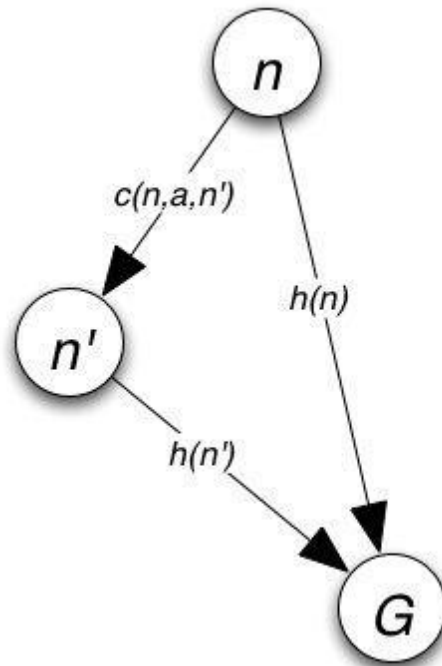
$$\Rightarrow f(n) = g(n) + h(n) \leq C^* \dots\dots\dots(2)$$

Now from (1) and (2)

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

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

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



Suppose n' is successor of n , then

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

we know that,

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

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

A heuristic is consistent if

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

Now from (1) and (2)

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

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

$\square f(n)$ is nondecreasing along any path.

Hill climbing

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

In this search method, we start with a partial solution and try to improve it for better solution

iteratively. For eg we start N queen problem with certain configuration and try to reduce conflicts by moving the queen iteratively so that conflicts are reduced and finally no conflicts. The iteration goes on until we get worse solution than the previous solution.

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms.

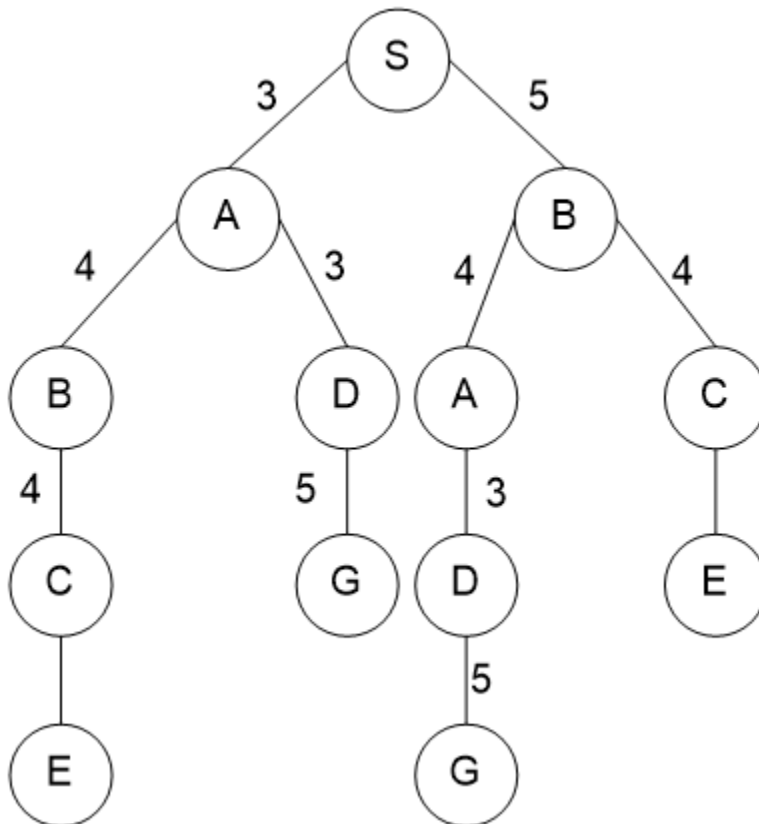
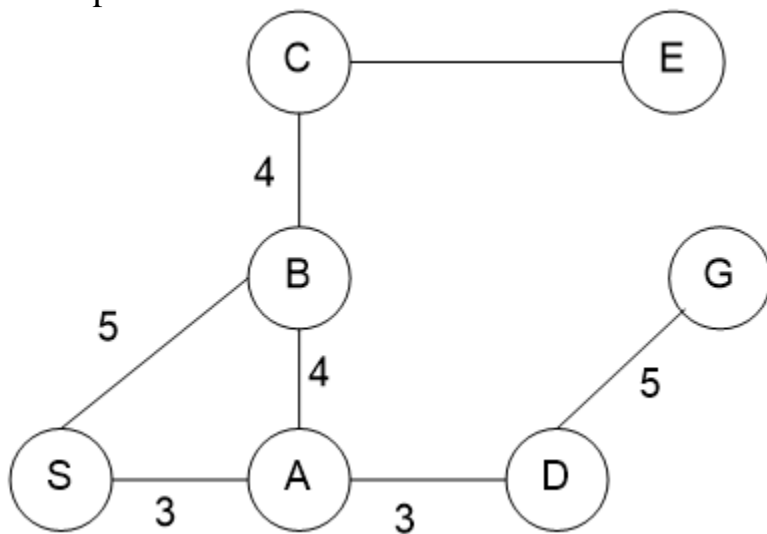
Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems, so long as a small number of increments typically converges on a good solution (the optimal solution or a close approximation).

Example: n -queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
-



Example:



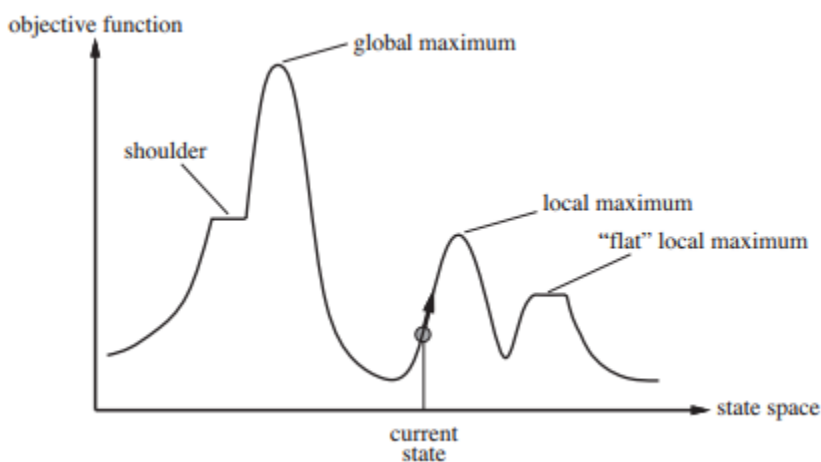
From the graph, S moves to B because it seems much closer to the goal than A (cost doesn't matter). A and C seems equi-distance, so goes to the A by heuristics. Then to D and finally G.

Note:

The difference between the hill climbing search method and the best first search method is the following one:

- the best first search method selects for expansion the most promising leaf node of the current search tree;
- the hill climbing search method selects for expansion the most promising successor of the node last expanded.

Problems with Hill Climbing



Hill climbing cannot reach the optimal/best state (global maximum) if it enters any of the following regions:

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

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: This is an area where the search space is flat so that all neighbors return the same evaluation. 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 current state. Chances are that we will land at a non-plateau region

Ridge: Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.

Ridges are a challenging problem for hill climbers that optimize in continuous spaces. Because hill climbers only adjust one element in the vector at a time, each step will move in an axis-aligned direction. If the target function creates a narrow ridge that ascends in a non-axis-aligned direction (or if the goal is to minimize, a narrow alley that descends in a non-axis-aligned direction), then the hill climber can only ascend the ridge (or descend the alley) by zig-zagging. If the sides of the ridge (or alley) are very steep, then the hill climber may be forced to take very tiny steps as it zig-zags toward a better position. Thus, it may take an unreasonable length of time for it to ascend the ridge (or descend the alley).

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

Means-ends analysis (MEA)

Means-ends analysis (MEA) is a problem solving technique used commonly in artificial intelligence (AI) for limiting search in AI programs

With MEA, 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. In the meantime however, problems may arise. It is often hard to determine where exactly the crux (point of difficulty) is. With the help of Means End Analysis, both forward and backward research can be done to determine where the stagnation (state without flow or movement) is occurring. This enables the larger parts of a problem to be solved first, to subsequently return to the smaller problems afterwards.

Monkey Banana Problem

The monkey and banana problems is a famous toy problem in artificial intelligence, particularly in logic programming and planning.

A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey?

There are many applications of this problem. One is as a toy problem for computer science.

Another possible purpose of the problem is to raise the question: Are monkeys intelligent?

Both humans and monkeys have the ability to use mental maps to remember things like where to go to find shelter, or how to avoid danger. They can also remember where to go to gather food and water, as well as how to communicate with each other.

Simulated Annealing

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

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

Where

c = the change in the evaluation function

t = the current value

r = a random number between 0 and 1

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

Game Search

Games are a form of *multi-agent environment*

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

Games – adversary

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

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

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

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

An exemplary game: Tic-tac-toe

There are two players denoted by X and O. They are alternatively writing their letter in one of the 9 cells of a 3 by 3 board. The winner is the one who succeeds in writing three letters in line. The game begins with an empty board. It ends in a win for one player and a loss for the other, or possibly in a draw.

A complete tree is a representation of all the possible plays of the game. The root node is the initial state, in which it is the first player's turn to move (the player X).

The successors of the initial state are the states the player can reach in one move, their successors are the states resulting from the other player's possible replies, and so on.

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

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

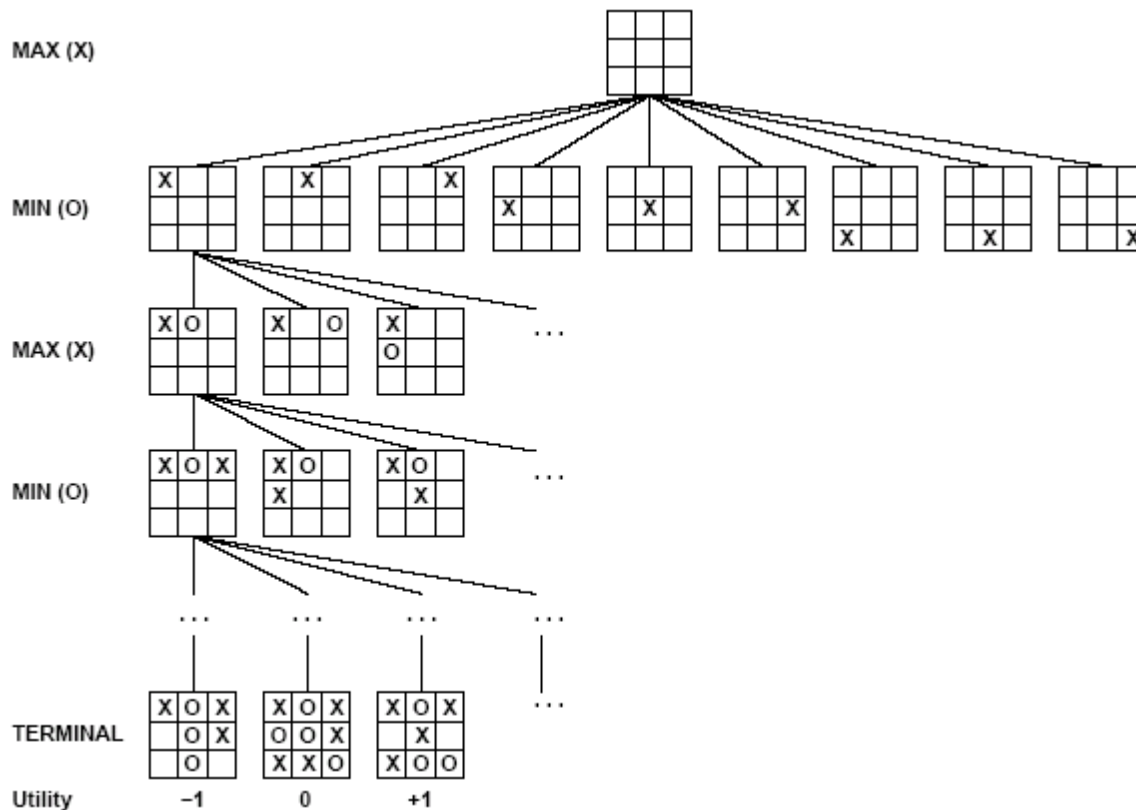
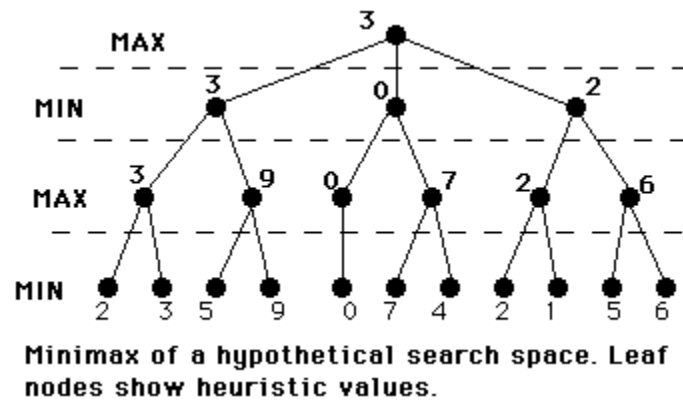


Figure: Partial game tree for Tic-Tac-Toe

The min-max Algorithm

Min-max algorithm allows us to determine the best move that maximizes the winning situation of a player. This algorithm assumes the zero sum game. If a player A is trying to maximize the cost then other player tries to minimize the cost. The game must be adversary. It means no player makes wrong move. In min-max algorithm we look ahead for some states and construct the game tree. Each node in the game tree represents the various situations in the game. There may be two types of nodes namely max and min node. Let max node represent the turn of player A and min node represents the turn of Player B. Max node always backs off with the maximum value of its children and min node backs off minimum value of its children. At the terminal of game tree, some heuristics is assigned. The heuristics is goodness for the player A for winning the game. After this we traverse the tree in DFS manner and values are backed off by min and max nodes according to their purpose up to the root node. The Player A then takes the move which maximizes the its winning situation.



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:

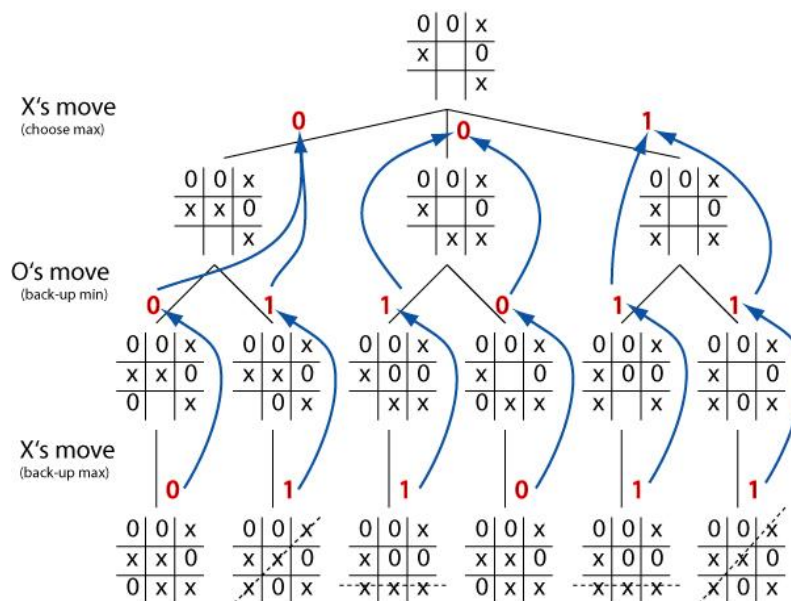
- the value of a node where it is the turn of player X to move is the maximum of the values of its successors (because X tries to maximize its outcome);

- 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



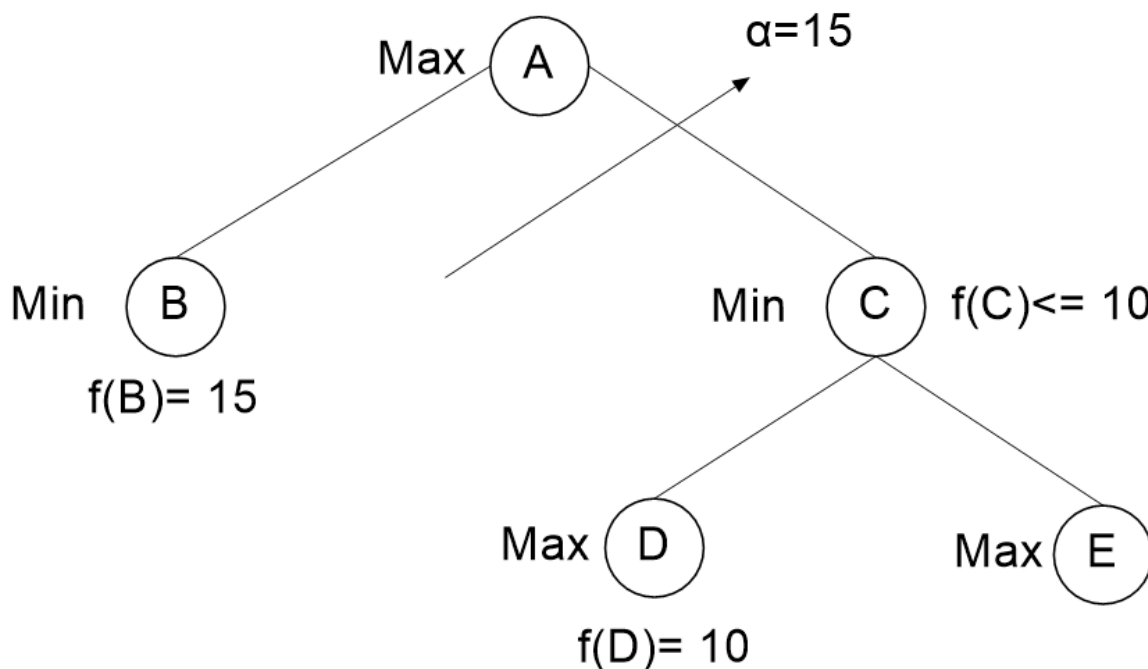
Alpha-beta pruning

We can increase the performance of max-min algorithm by using alpha beta pruning. Alpha-beta pruning is a technique for evaluating nodes of a game tree that eliminates unnecessary evaluations. The maximum value backed off by max node from all its children, is taken as alpha value and minimum value backed off by min node from all its children is taken as beta. Alpha bound of a min node J is the maximum value of all max ancestors nodes of J. Beta bound of a max node K is the min value of all ancestors of K. We keep on updating alpha and beta bounds for min and max nodes as different values are backed off when we make DFS search by min and max nodes. The exploration of max node or min node is stopped when $\alpha \geq \beta$.

An alpha cutoff

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

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



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

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

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

A beta cutoff

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

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

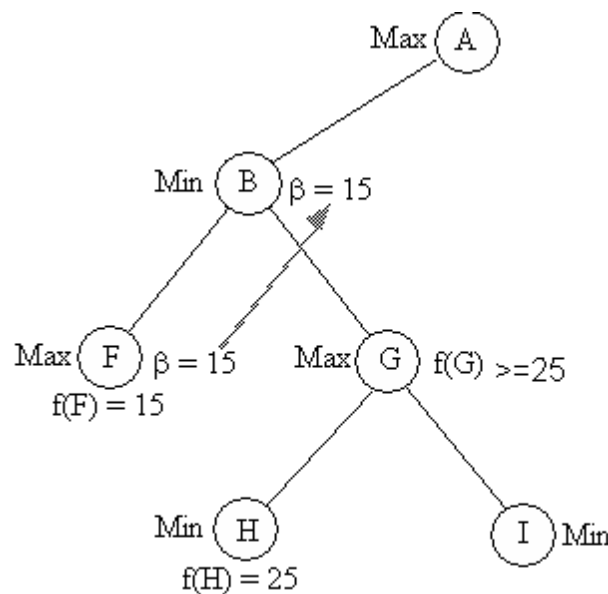


Figure: Illustration of the beta cut-off.

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

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

15. This value is transmitted upward to the node B and will be used for evaluating the other possible moves from B.

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

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

Game of chance

A game of chance is a game whose outcome is strongly influenced by some randomizing device, and upon which contestants may choose to wager money or anything of monetary value. Common devices used include dice, spinning tops, playing cards, roulette wheels, or numbered balls drawn from a container. A game of chance may have some skill element to it, however, chance generally plays a greater role in determining the outcome than skill. A game of skill, on the other hand, also may have elements of chance, but with skill playing a greater role in determining the outcome.

Examples: card playing, dice rolling, coin flipping, gambling games

Game Theory

Game theory is a framework for hypothetical social situations among competing players. 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 mathematicians John von Neumann and John Nash, as well as economist Oskar Morgenstern.

The focus of game theory is the game, which serves as a model of an interactive situation among rational players. The key to game theory is that one player's payoff is contingent on the strategy implemented by the other player. The game identifies the players' identities, preferences, and available strategies and how these strategies affect the outcome. Depending on the model, various other requirements or assumptions may be necessary.

Classic Game Theory Example

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 4 deals:

1. If both confess, they will each receive a 5-year prison sentence.
2. If prisoner 1 confesses but prisoner 2 does not, prisoner 1 will get 3 years and prisoner 2 will get 9 years.
3. If prisoner 2 confesses but prisoner 1 does not, prisoner 1 will get 10 years and prisoner 2 will get 2 years.
4. If neither confesses, each will serve 2 years in prison.

The most favorable strategy is to not confess. However, neither is aware of the other's strategy and without certainty that one will not confess, both will likely confess and receive a 5-year prison sentence.

Constraint Satisfaction Problem:

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

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

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

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

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

A CS problem is usually represented as an undirected graph, called *Constraint Graph* where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyper arcs. In the following we restrict our attention to the case of unary and binary constraints.

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a domain of values, and C is a set of constraints. Every constraint is in turn a pair $\langle t, R \rangle$, where t is a tuple of variables and R is a set of tuples of values; all these tuples having the same number of elements; as a result R is a relation. An evaluation of the variables is a function from variables to values, $v: X \rightarrow D$. Such an evaluation satisfies a constraint $\langle (x_1, x_2, \dots, x_n), R \rangle$ if $(v(x_1), \dots, v(x_n)) \in R$. A solution is an evaluation that satisfies all constraints.

Constraints

- A constraint is a relation between a **local** collection of variables.
- The constraint restricts the values that these variables can simultaneously have.
- For example, **all-diff(X1, X2, X3)**. This constraint says that X1, X2, and X3 must take on different values. Say that {1,2,3} is the set of values for each of these variables then:

X1=1, X2=2, X3=3 OK X1=1, X2=1, X3=3 NO

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

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

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

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

☐ Queens can attack each other,

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

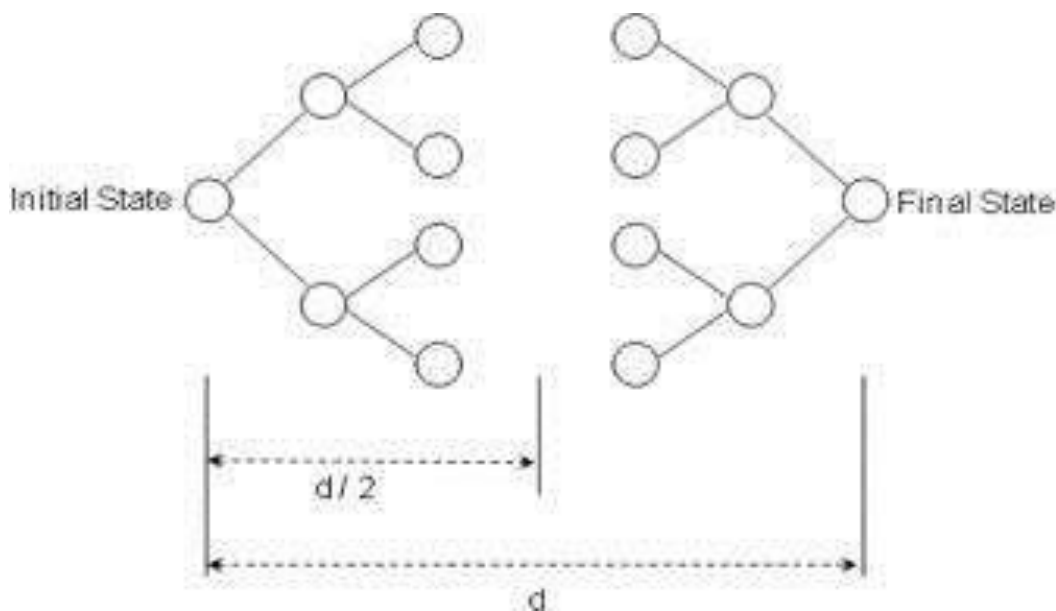
□ Representing the Constraints;

1. Between every pair of variables (Q_i, Q_j) ($i \neq j$), we have a constraint C_{ij} .
2. For each C_{ij} , an assignment of values to the variables $Q_i = A$ and $Q_j = B$, satisfies this constraint if and only if;
 $A \neq B$
 $|A - B| \neq |i - j|$

□ Solutions:

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

Bidirectional search



The idea behind bidirectional search is to run two simultaneous BFS one from initial state and other backward from goal state. This stops when the two searches meet with some common node that provides path from initial state to goal state. One or of the search can check if the node is in the fringe of another search tree before expanding it. If so, the path is found.

Time complexity is $2 O(b^{d/2})$ i.e $O(b^{d/2})$

Space complexity. At least one of the search tree must be kept in memory to check membership. Thus space complexity is $O(b^{d/2})$

AND/OR graphs (AO*)

It contains AND node and OR node. Each node represents problem. AND node represent actual decomposition. It means both sub problems must be solved. It takes sum of the cost i.e. it adds up all the cost of solving a problem and its sub problem and backs it up. Whereas OR node represents choice of decomposition. It means we can solve either one of problem. OR node takes minimum of cost i.e. OR node back up the best cost of successor. Game trees are similar with AO trees in the sense that both of them contains two different types of node. Game graph contains MAX and MIN nodes. A* graph is different from AO* in the sense that A* has only OR nodes.

--best algorithm for solving AND OR graphs.

--problem is divided into sub problems where sub problems can be solved separately.

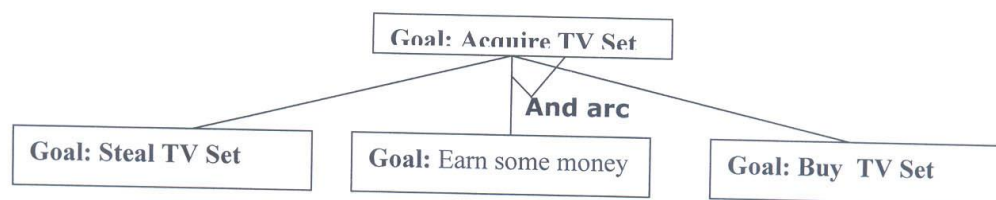
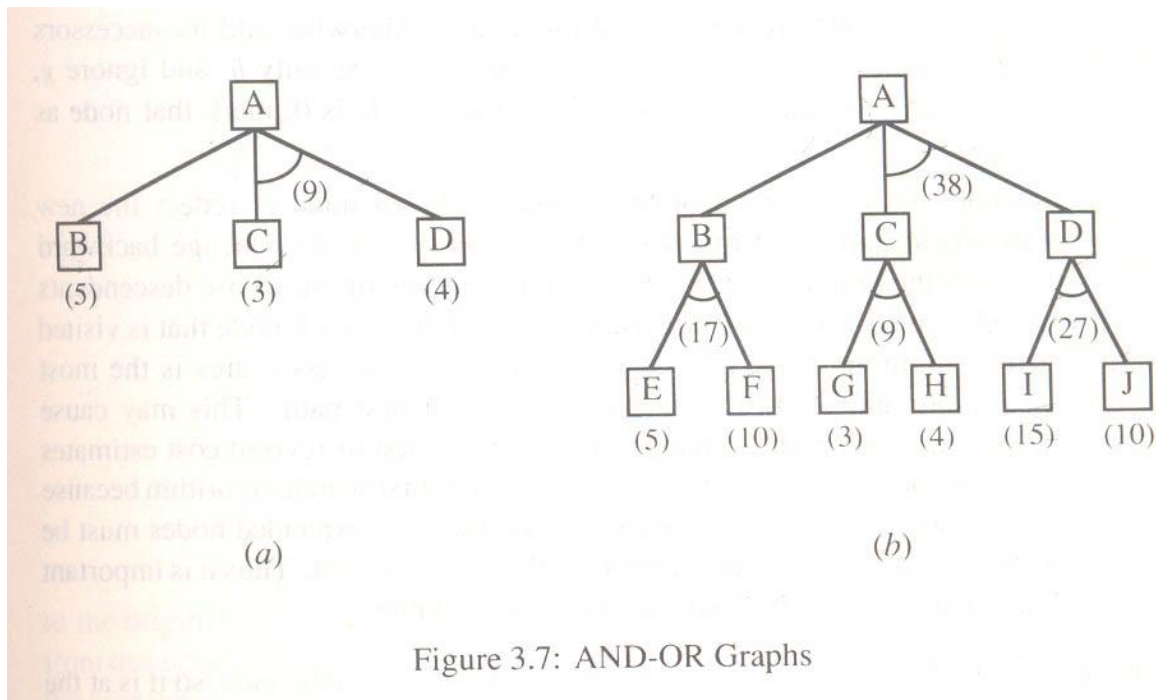


Figure shows AND - Or graph - an example.

Cryptarithmic Problems

Cryptarithmic problems are mathematical puzzles where numbers are replaced with alphabets or symbols. Usually it requires each letter would be replaced by unique digit. Each letter having different value from the other letters. In artificial intelligence, this problem is under category of Constraint Satisfaction Problem (CSP), or constraint programming.

By using standard arithmetic rules we need to decipher the alphabet.

General Rules:

1. Each alphabet takes only one number from 0 to 9 uniquely.
2. Two single digit numbers sum can be maximum 19 with carryover. So carry over in problems of two number addition is always 1.
3. Try to solve left most digit in the given problem.
4. If $a \times b = kb$, then the following are the possibilities
($3 \times 5 = 15$; $7 \times 5 = 35$; $9 \times 5 = 45$)

Solved Example 1:

The following questions are based on the following multiplication, where each digit has been replaced by an alphabet.

$$\begin{array}{r} J E \\ \times B B \\ \hline J E \\ J E A \\ \hline B A D E \end{array}$$

1. Find the value of J and A.
2. If $E = 4$, what is the value of D?

Explanation:

From the first two of multiplication, you can clearly say that $B = 1$, as $JE \times B = JE$. From the second row of multiplication, $A = 0$ as in the multiplication, second row should start from tenth's place. So $A = 0$. Now in the hundred's place, $J + \text{Something} = 10$. When you add something to the single digit number that results in 10. So $J = 9$. Now from the above table, we cannot determine values of E and D, but we can say, that E and D are consecutive. As it is given that $E = 4$, we can say $D = 3$.

$$\begin{array}{r} 9 E \\ \times 1 1 \\ \hline 9 E \\ 9 E 0 \\ \hline 1 0 3 E \end{array}$$

Solved Example 2:

From the multiplication below, What is the value of NAME?

$$\begin{array}{r}
 \text{H E} \\
 \times \text{E H} \\
 \hline
 \text{H E} \\
 \text{H H A} \\
 \hline
 \text{H N M E}
 \end{array}$$

Explanation:

From the first row of multiplication, $H=1$ is clear, As $HE \times H = HE$. Substitute $H = 1$ in all places. Now from the tenth's place, think about, the value of A. $1 + A = M$. If M is a single digit number, then $N = 1$, which is impossible (Already we have given $H = 1$). So $A = 9$, Then $M = 0$, and $N = 2$. Now $1E \times E = 119$. So by trial and error $E = 7$.

Therefore, NAME = 2907

$$\begin{array}{r}
 17 \\
 \times 17 \\
 \hline
 119 \\
 170 \\
 \hline
 2907
 \end{array}$$

Solved Example 3:

If $SEND + MORE = MONEY$ then find the respective values

Explanation:

Addition of two numbers with 'n' digits, results in a n+1 digits, then the left most place always = 1.

So $M=1$. Substitute this value.

Now 'o' cannot be 1 as M already 1. It may not be 2 either as $S+1 = 12$ or $1 + S + 1 = 12$ in the both cases S is a two digit number. So 'o' is nothing but zero. Put $o = 0$.

Now S can be either 8 or 9. If $S = 8$, then there must be a carry over.

$$E + 0 = 10 + N \text{ or } 1 + E + 0 = 10 + N$$

In the above two cases, $E - N = 10$ is not possible and $E - N = 9$ not possible as N cannot be zero.

So $E = 9$.

Now $E + 0 = N$ is not possible as $E = N$. So $1 + E = N$ possible.

$$\begin{array}{r}
 1 \\
 9 \text{ E N D} \\
 + 1 \text{ 0 R E} \\
 \hline
 1 \text{ 0 N E Y}
 \end{array}$$

The possible cases are, $N + R = 10 + E$ --- (1) or $1 + N + R = 10 + E$ --- (2)

Substituting $E = N - 1$ in the first equation, $N + R = 10 + N - 1$, we get $R = 9$ which is not possible.

Substituting $E = N - 1$ in the second equation, $1 + N + R = 10 + N - 1$, we get $R = 8$.

We know that N and E are consecutive and N is larger. Take $(N, E) = (7, 6)$ check and substitute,

you won't get any unique value for D.

Take $(N, E) = (6, 5)$, Now you get $D = 7, Y = 2$.

$$\begin{array}{r} 9 \quad 5 \quad 6 \quad 7 \\ 1 \quad 0 \quad 8 \quad 5 \\ \hline 1 \quad 0 \quad 6 \quad 5 \quad 2 \end{array}$$

Solved Example 4:

Find the values of all the alphabets if each alphabet represent a single digit from 0 - 9

$$\begin{array}{r} A \quad B \quad C \quad D \\ + E \quad B \quad C \quad B \\ \hline A \quad F \quad G \quad A \quad G \end{array}$$

Explanation:

Let us name the columns as below

$$\begin{array}{r} 1 \quad 2 \quad 3 \quad 4 \\ A \quad B \quad C \quad D \\ + E \quad B \quad C \quad B \\ \hline A \quad F \quad G \quad A \quad G \end{array}$$

We know that sum of two single digit alphabets should not cross 18, and maximum difference between two alphabets is 9.

If we add two maximum 4 digit numbers the sum is maximum 19998. So the digit in the 5th left is 1.

Now from the 1st column $1 + E = 1F$; if there is any carry over from the 2nd column $1 + 1 + E = 1F$

But $1F$ is a two digit number in alphanumeric is equal to $10 + F$

So $1+E=10+F \Rightarrow E-F=9$ $1+E=10+F \Rightarrow E-F=9$

From this relation we know that $E = 9, F = 0$

or $1+1+E=10+F \Rightarrow E-F=8$ $1+1+E=10+F \Rightarrow E-F=8$

$E = 9, F = 1$ or $E = 8, F = 0$

From the above we can infer that $F = 0$ but we don't know whether E is equal to either 8 or 9.

But surely F is not equal to 1 as we fixed already $A = 1$

Now from the 3rd column,

$2C = 1 \Rightarrow C = 1/2$

$1 + 2C = 1 \Rightarrow C = 0$

If the sum is a two digit number then

$2C = 11 \Rightarrow C = 11/2$

$$1 + 2C = 11 \Rightarrow C = 5$$

From the above $C = 1/2$ and $11/2$ are not possible nor is 0 possible as we fixed $F = 0$

If $C = 5$ then $A = 1$ and there is a carry over to the left column. and also there must be carry over from the first column, but we don't know $1 + 2B$ is a single digit or two digit number

From the second and fourth columns

$$1 + 2B = G \quad \text{--- (1) or } 1 + 2B = 10 + G \quad \text{--- (2)}$$

$$D + B = 10 + G \quad \text{--- (3)}$$

Solving (1) and (3) we get $D - B = 11$ which is not possible

But If we solve (2) and (3) then we get $D - B = 1$

So D and B are consecutive numbers and their sum is more than 10. So acceptable values are $D = 7$ and $B = 6$

This completes our problem so final table looks like the following

A	B	C	D		1	6	5	7	
+	E	B	C	B	+	8	6	5	6
A	F	G	A	G	1	0	3	1	3