CHANDIGARH ENGINEERING COLLEGE----------B.TECH.(CSE 6th Sem.)

Notes

Subject: Artificial Intelligence

Subject Code: ((BTCS 602-18))

Unit 2 Search Algorithms

Uninformed Search Algorithms

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

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

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

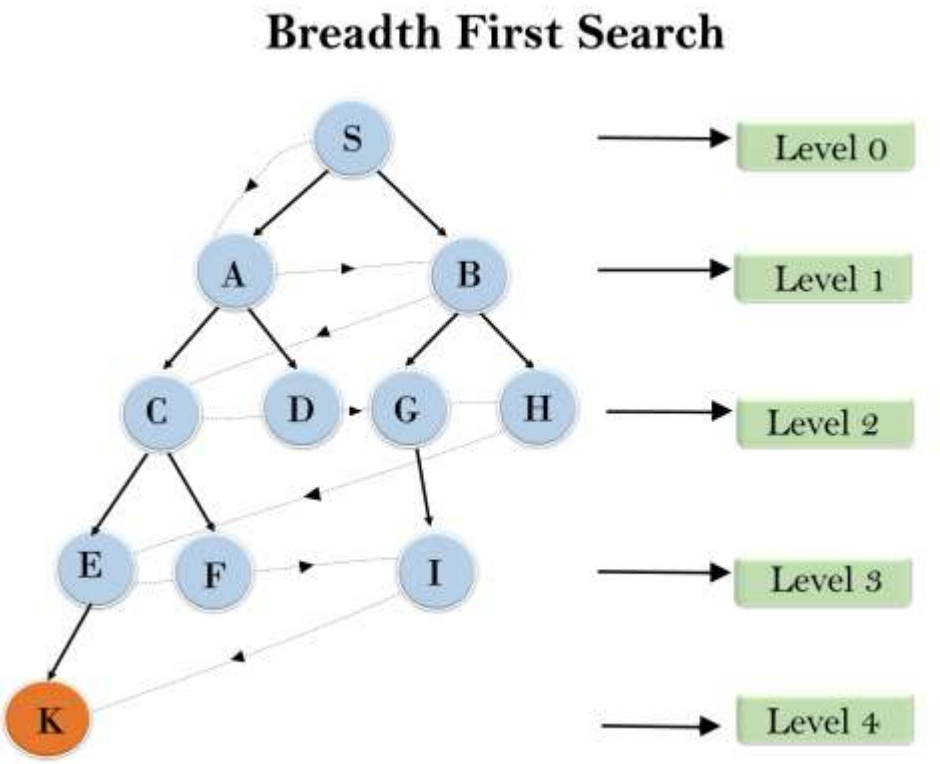
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

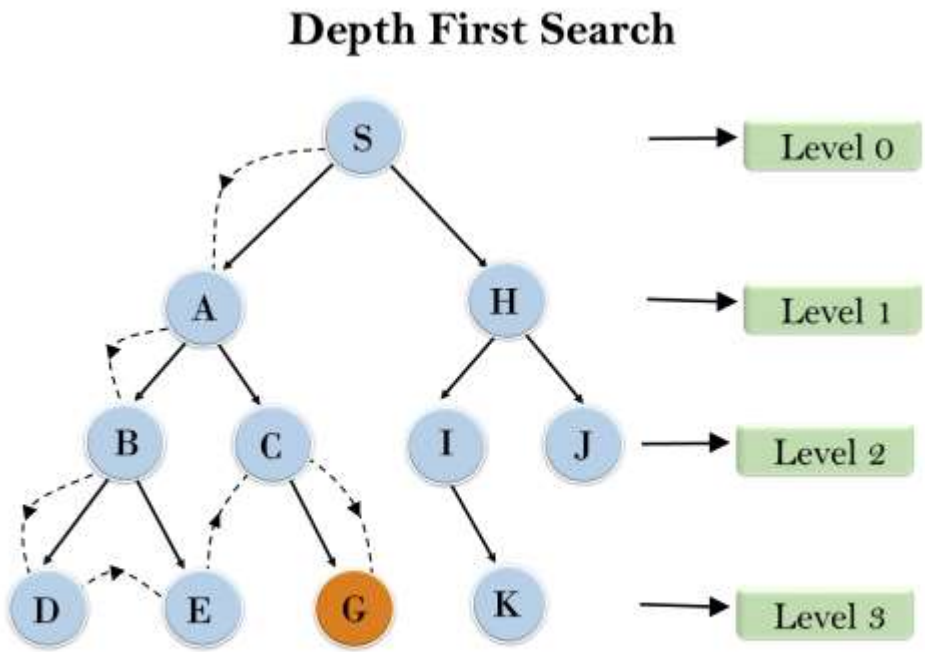
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. 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.

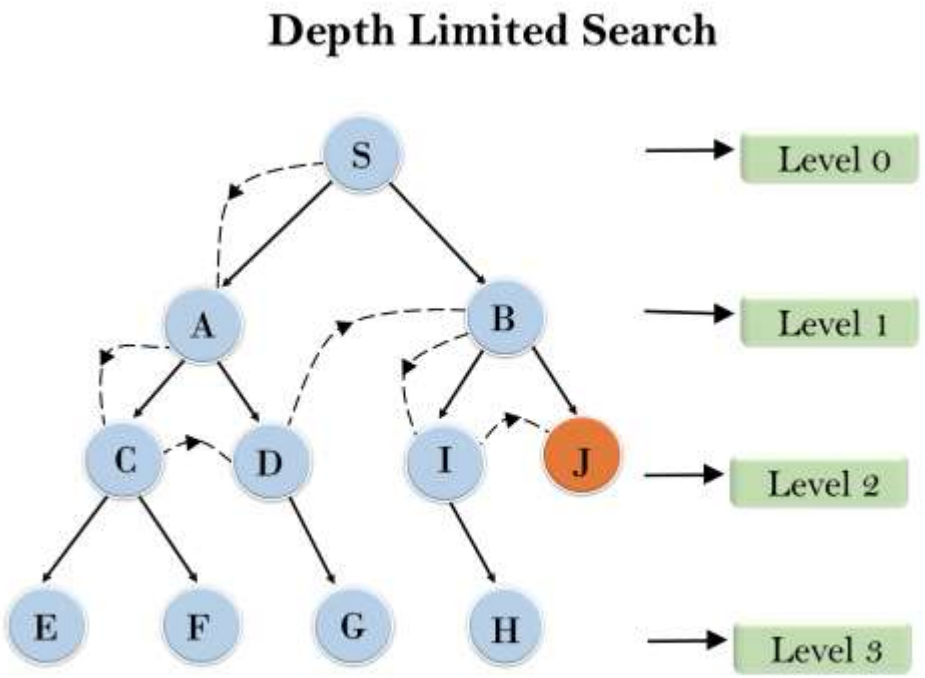
Advantages:

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.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **O(b^{ℓ})**.

Space Complexity: Space complexity of DLS algorithm is O**(b×ℓ)**.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

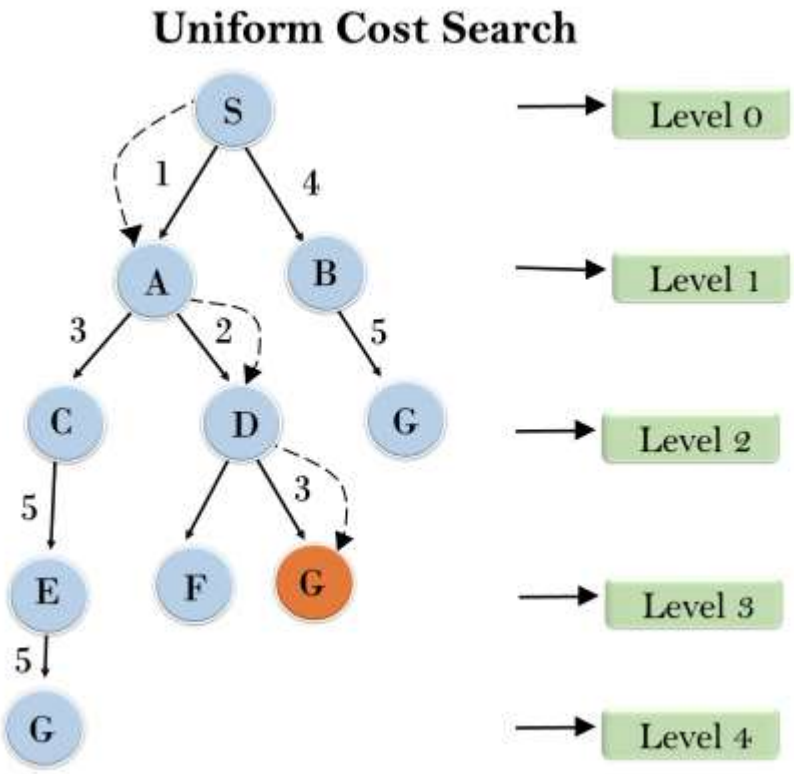
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is$\mathbf{O(b^{1 + \lceil C^*/\varepsilon \rceil})}$/.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + \lceil C^*/\varepsilon \rceil})}$.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

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:

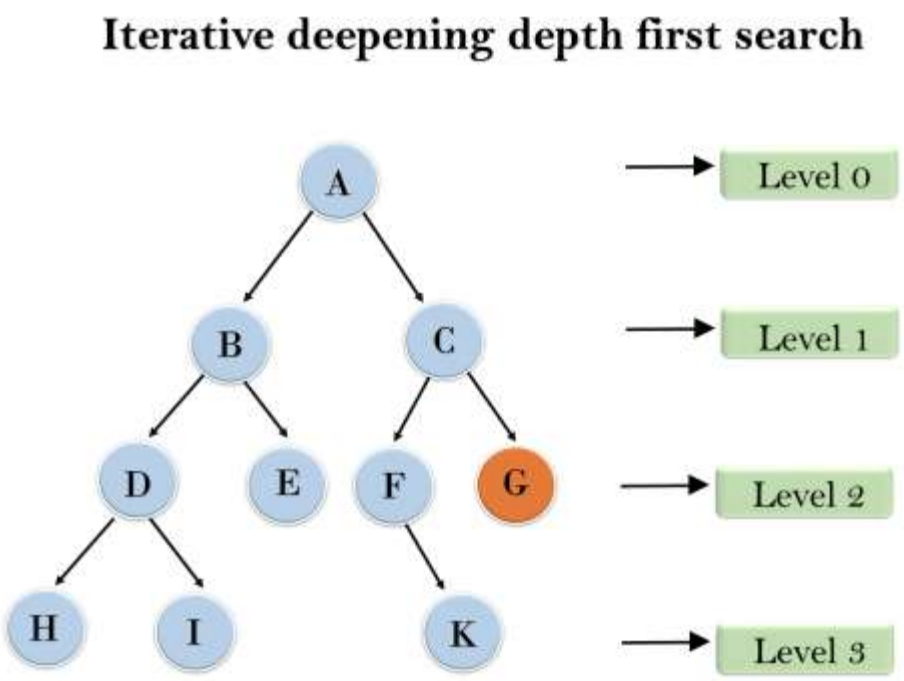
- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

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

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

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

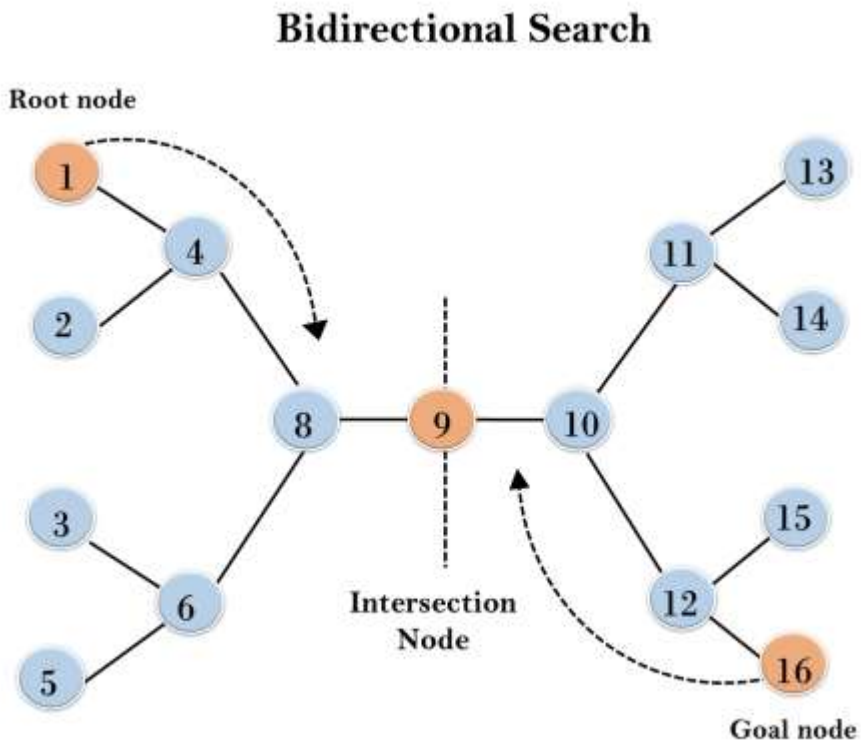
Disadvantages:

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

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is **$O(b^d)$**.

Space Complexity: Space complexity of bidirectional search is **$O(b^d)$**.

Optimal: Bidirectional search is Optimal.

Informed Search Algorithms

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

1. $h(n) <= h^*(n)$

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

1.) 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.

1. $f(n)= g(n)$.

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

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

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Advantages:

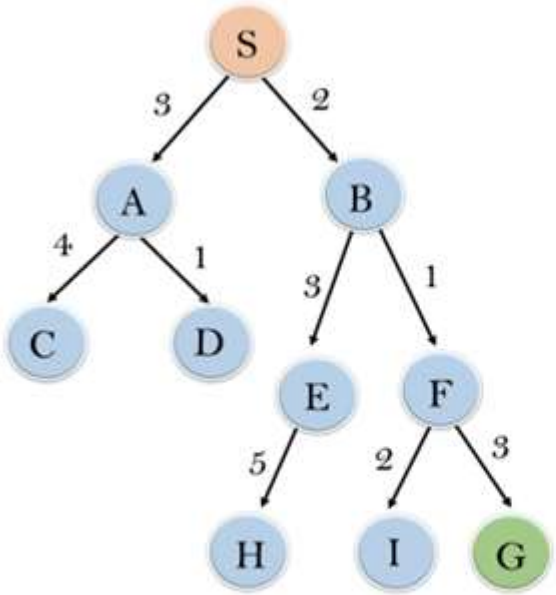
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

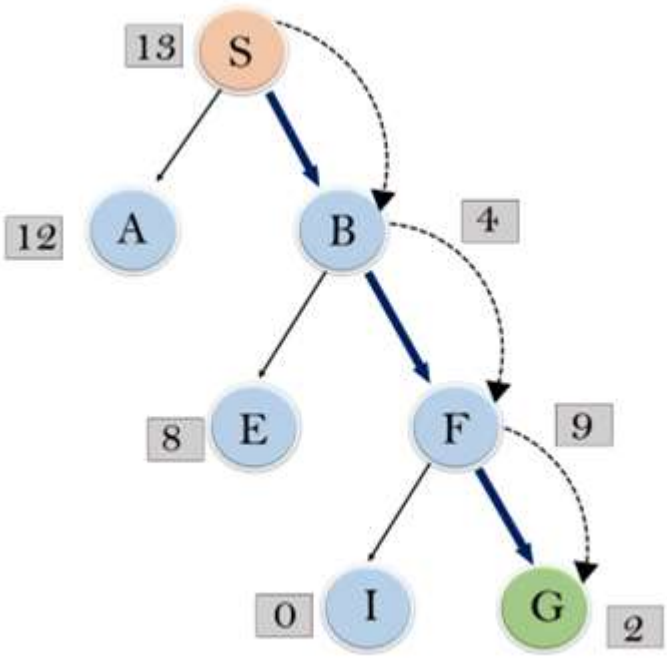
Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B] : Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F] : Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

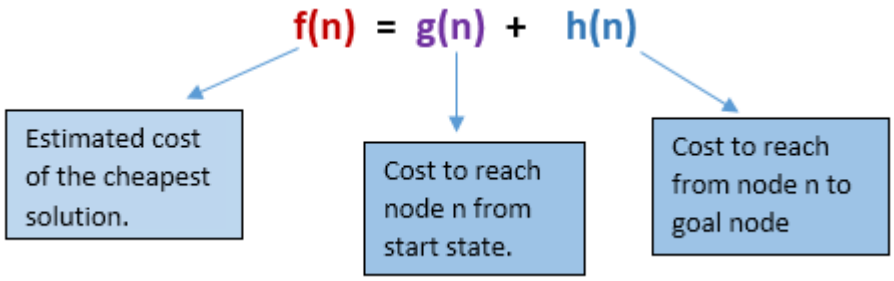
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

2.) 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**.



At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

Advantages:

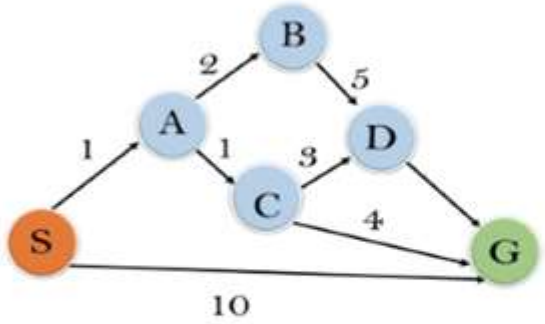
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

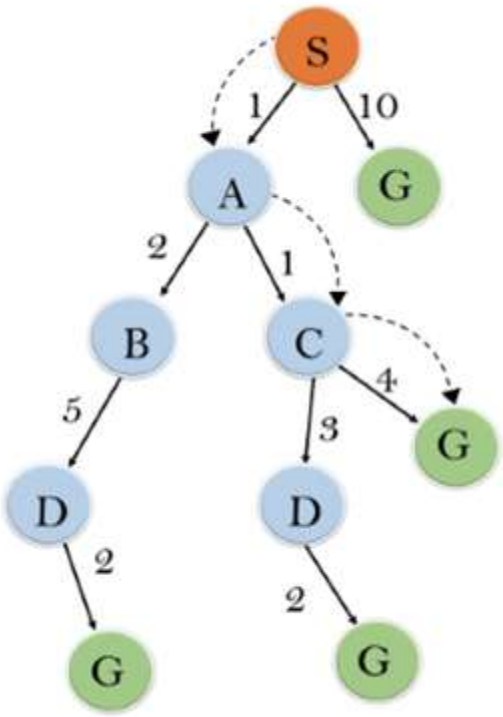
Example:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.
Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

Artificial Intelligence (AI) covers a range of techniques that appear as sentient behavior by the computer. For example, AI is used to recognize faces in photographs on your social media, beat the World's Champion in chess, and process your speech when you speak to Siri or Alexa on your phone.

In this course, we will explore some of the ideas that make AI possible:

1. **Search**

Finding a solution to a problem, like a navigator app that finds the best route from your origin to the destination, or like playing a game and figuring out the next move.

1. **Knowledge**

Representing information and drawing inferences from it.

2. **Uncertainty**

Dealing with uncertain events using probability.

3. **Optimization**

Finding not only a correct way to solve a problem, but a better—or the best—way to solve it.

4. **Learning**

Improving performance based on access to data and experience. For example, your email is able to distinguish spam from non-spam mail based on past experience.

5. **Neural Networks**

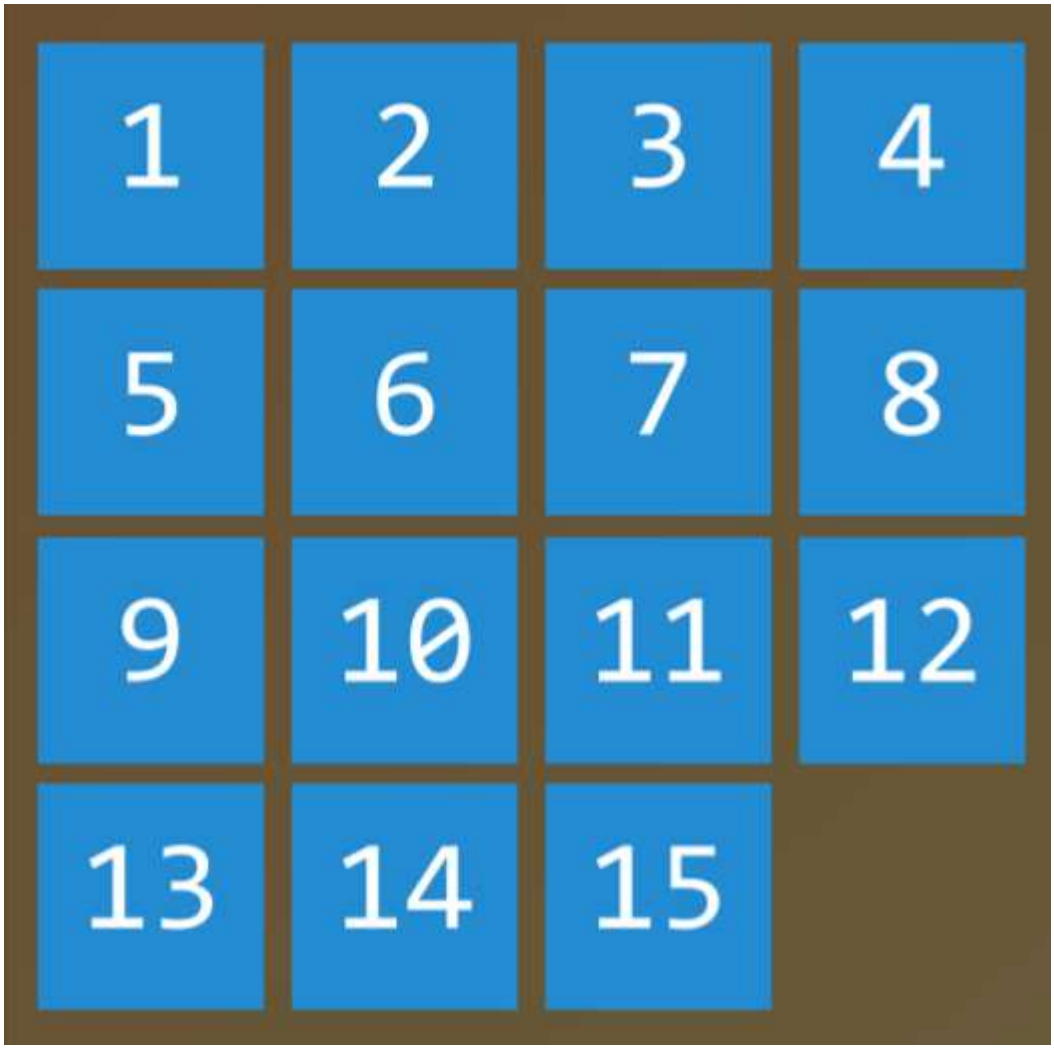
A program structure inspired by the human brain that is able to perform tasks effectively.

6. **Language**

Processing natural language, which is produced and understood by humans.

Search

Search problems involve an agent that is given an initial state and a goal state, and it returns a solution of how to get from the former to the latter. A navigator app uses a typical search process, where the agent (the thinking part of the program) receives as input your current location and your desired destination, and, based on a search algorithm, returns a suggested path. However, there are many other forms of search problems, like puzzles or mazes.



Finding a solution to a 15 puzzle would require the use of a search algorithm.

- **Agent**

  An entity that perceives its environment and acts upon that environment. In a navigator app, for example, the agent would be a representation of a car that needs to decide on which actions to take to arrive at the destination.

- **State**

  A configuration of an agent in its environment. For example, in a 15 puzzle, a state is any one way that all the numbers are arranged on the board.

  - **Initial State**

    The state from which the search algorithm starts. In a navigator app, that would be the current location.

- **Actions**

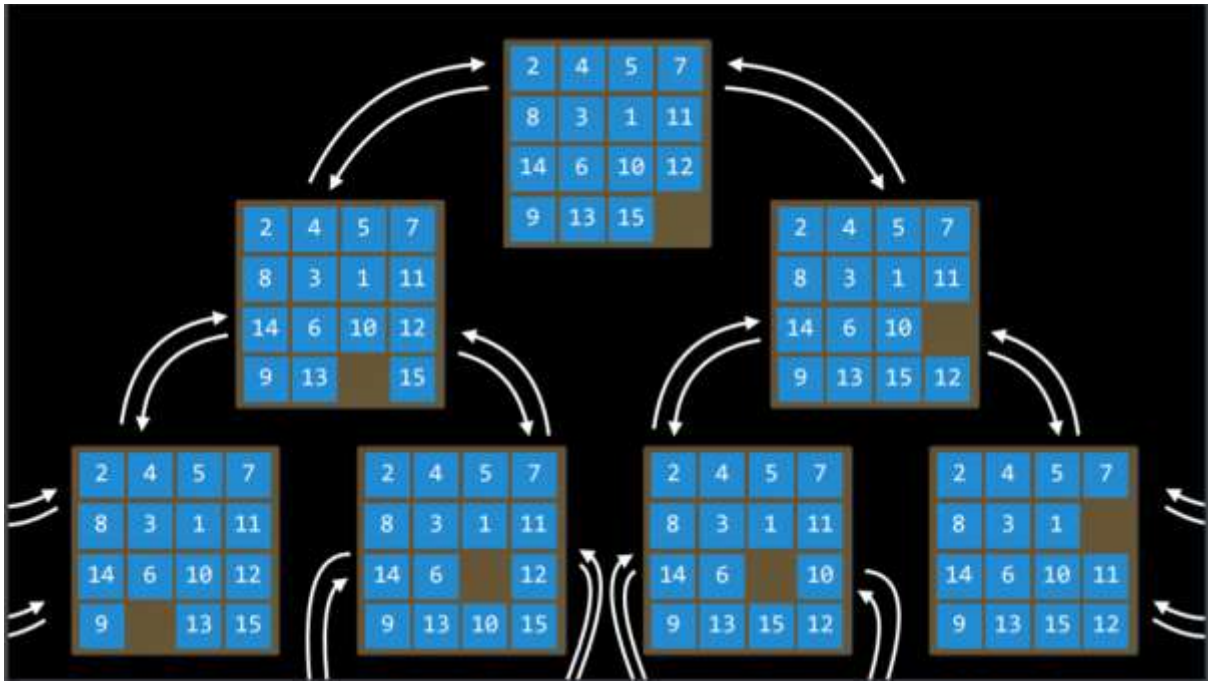
  Choices that can be made in a state. More precisely, actions can be defined as a function. Upon receiving state s as input, Actions(s) returns as output the set of actions that can be executed in state s. For example, in a *15 puzzle*, the actions of a given state are the ways you can slide squares in the current configuration (4 if the empty square is in the middle, 3 if next to a side, 2 if in the corner).

- **Transition Model**

  A description of what state results from performing any applicable action in any state. More precisely, the transition model can be defined as a function. Upon receiving state s and action a as input, Results(s, a) returns the state resulting from performing action a in state s. For example, given a certain configuration of a *15 puzzle* (state s), moving a square in any direction (action a) will bring to a new configuration of the puzzle (the new state).

- **State Space**

  The set of all states reachable from the initial state by any sequence of actions. For example, in a 15 puzzle, the state space consists of all the 16!/2 configurations on the board that can be reached from any initial state. The state space can be visualized as a directed graph with states, represented as nodes, and actions, represented as arrows between nodes.



- **Goal Test**

  The condition that determines whether a given state is a goal state. For example, in a navigator app, the goal test would be whether the current location of the agent (the representation of the car) is at the destination. If it is — problem solved. If it's not — we continue searching.

- **Path Cost**

  A numerical cost associated with a given path. For example, a navigator app does not simply bring you to your goal; it does so while minimizing the path cost, finding the fastest way possible for you to get to your goal state.

Solving Search Problems

- **Solution**

  A sequence of actions that leads from the initial state to the goal state.

  - **Optimal Solution**

A solution that has the lowest path cost among all solutions.

In a search process, data is often stored in a ***node***, a data structure that contains the following data:

- A *state*
- Its *parent node*, through which the current node was generated
- The *action* that was applied to the state of the parent to get to the current node
- The *path cost* from the initial state to this node

Nodes contain information that makes them very useful for the purposes of search algorithms. They contain a *state*, which can be checked using the *goal test* to see if it is the final state. If it is, the node's *path cost* can be compared to other nodes' *path costs*, which allows choosing the *optimal solution*. Once the node is chosen, by virtue of storing the *parent node* and the *action* that led from the *parent* to the current node, it is possible to trace back every step of the way from the *initial state* to this node, and this sequence of actions is the *solution*.

However, *nodes* are simply a data structure — they don't search, they hold information. To actually search, we use the **frontier**, the mechanism that "manages" the *nodes*. The *frontier* starts by containing an initial state and an empty set of explored items, and then repeats the following actions until a solution is reached:

Repeat:

1. If the frontier is empty,
    - *Stop.* There is no solution to the problem.
2. Remove a node from the frontier. This is the node that will be considered.
3. If the node contains the goal state,
    - Return the solution. *Stop*.

    Else,

    * Expand the node (find all the new nodes that could be reached from this node), and add resulting nodes to the frontier.
    * Add the current node to the explored set.

Depth-First Search

In the previous description of the *frontier*, one thing went unmentioned. At stage 1 in the pseudocode above, which node should be removed? This choice has implications on the quality of the solution and how fast it is achieved. There are multiple ways to go about the question of which nodes should be considered first, two of which can be represented by the data structures of **stack** (in *depth-first* search) and **queue** (in *breadth-first search*; and here is a cute cartoon demonstration of the difference between the two).

We start with the *depth-first* search (*DFS*) approach.

A *depth-first* search algorithm exhausts each one direction before trying another direction. In these cases, the frontier is managed as a *stack* data structure. The catchphrase you need to remember here is "*last-in first-out*." After nodes are being added to the frontier, the first node to remove and consider is the last one to be added. This results in a search algorithm that goes as deep as possible in the first direction that gets in its way while leaving all other directions for later.

(An example from outside lecture: Take a situation where you are looking for your keys. In a *depth-first* search approach, if you choose to start with searching in your pants, you'd first go through every single pocket, emptying each pocket and going through the contents carefully. You will stop searching in your pants and start searching elsewhere only once you will have completely exhausted the search in every single pocket of your pants.)

- Pros:
    - At best, this algorithm is the fastest. If it "lucks out" and always chooses the right path to the solution (by chance), then *depth-first* search takes the least possible time to get to a solution.
- Cons:
    - It is possible that the found solution is not optimal.
    - At worst, this algorithm will explore every possible path before finding the solution, thus taking the longest possible time before reaching the solution.

Code example:

```
  # Define the function that removes a node from the frontier and returns it.
  def remove(self):
        # Terminate the search if the frontier is empty, because this means that there is no solution.
    if self.empty():
      raise Exception("empty frontier")
    else:
        # Save the last item in the list (which is the newest node added)
      node = self.frontier[-1]
      # Save all the items on the list besides the last node (i.e. removing the last node)
      self.frontier = self.frontier[:-1]
      return node
```

Breadth-First Search

The opposite of *depth-first* search would be *breadth-first* search (*BFS*).

A *breadth-first* search algorithm will follow multiple directions at the same time, taking one step in each possible direction before taking the second step in each direction. In this case, the frontier is managed as a *queue* data structure. The catchphrase you need to remember here is "*first-in first-out*." In this case, all the new nodes add up in line, and nodes are being considered based on which one was added first (first come first served!). This results in a search algorithm that takes one step in each possible direction before taking a second step in any one direction.

(An example from outside lecture: suppose you are in a situation where you are looking for your keys. In this case, if you start with your pants, you will look in your right pocket. After this, instead of looking at your left pocket, you will take a look in one drawer. Then on the table. And so on, in every location you can think of. Only after you will have exhausted all the locations will you go back to your pants and search in the next pocket.)

- Pros:
    - This algorithm is guaranteed to find the optimal solution.
- Cons:
    - This algorithm is almost guaranteed to take longer than the minimal time to run.
    - At worst, this algorithm takes the longest possible time to run.

Code example:

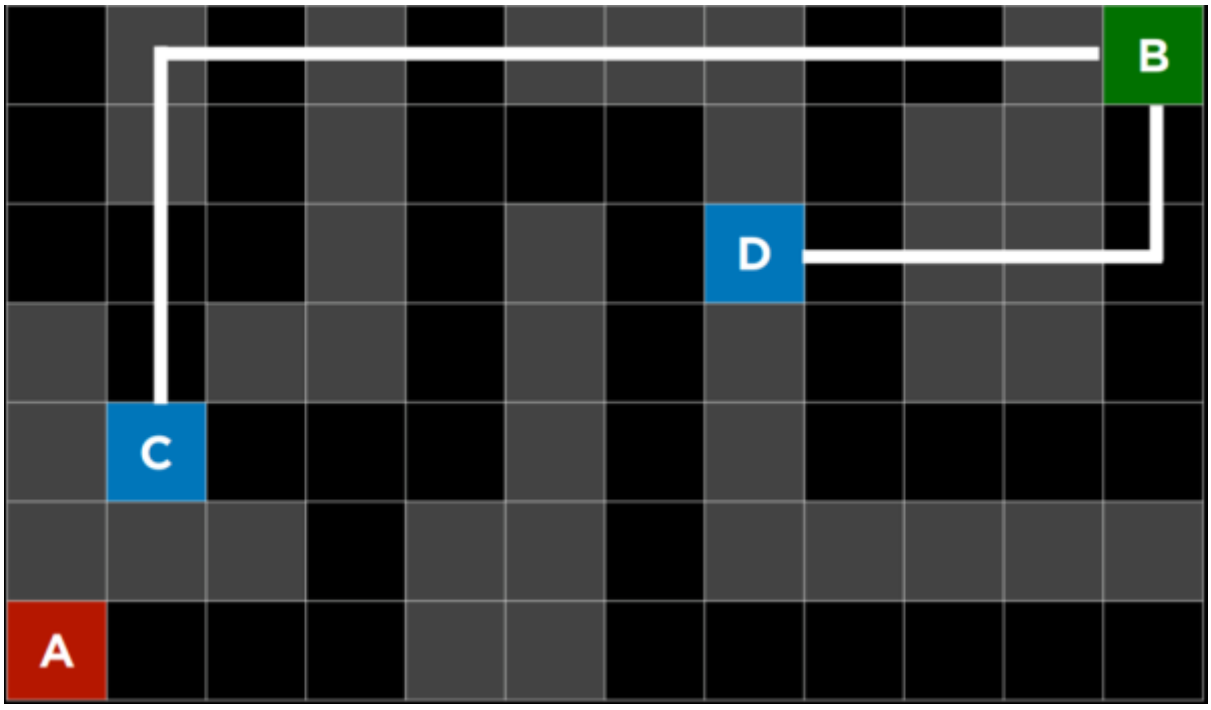
```
  # Define the function that removes a node from the frontier and returns it.
  def remove(self):
        # Terminate the search if the frontier is empty, because this means that there is no solution.
    if self.empty():
      raise Exception("empty frontier")
```

```
        else:
            # Save the oldest item on the list (which was the first one to be added)
            node = self.frontier[0]
            # Save all the items on the list besides the first one (i.e. removing the first node)
            self.frontier = self.frontier[1:]
            return node
```

Greedy Best-First Search

Breadth-first and depth-first are both **uninformed** search algorithms. That is, these algorithms do not utilize any knowledge about the problem that they did not acquire through their own exploration. However, most often is the case that some knowledge about the problem is, in fact, available. For example, when a human maze-solver enters a junction, the human can see which way goes in the general direction of the solution and which way does not. AI can do the same. A type of algorithm that considers additional knowledge to try to improve its performance is called an **informed** search algorithm.

Greedy best-first search expands the node that is the closest to the goal, as determined by a **heuristic function** *h(n)*. As its name suggests, the function estimates how close to the goal the next node is, but it can be mistaken. The efficiency of the *greedy best-first* algorithm depends on how good the heuristic function is. For example, in a maze, an algorithm can use a heuristic function that relies on the **Manhattan distance** between the possible nodes and the end of the maze. The *Manhattan distance* ignores walls and counts how many steps up, down, or to the sides it would take to get from one location to the goal location. This is an easy estimation that can be derived based on the (x, y) coordinates of the current location and the goal location.



Manhattan Distance

However, it is important to emphasize that, as with any heuristic, it can go wrong and lead the algorithm down a slower path than it would have gone otherwise. It is possible that an *uninformed* search algorithm will provide a better solution faster, but it is less likely to do so than an *informed* algorithm.

A* Search

A development of the *greedy best-first* algorithm, *A* search* considers not only *h(n),* the estimated cost from the current location to the goal, but also *g(n),* the cost that was accrued until the current location. By combining both these values, the algorithm has a more accurate way of determining the cost of the solution and optimizing its choices on the go. The algorithm keeps track of (*cost of path until now + estimated cost to the goal*), and once it exceeds the estimated cost of some previous option, the algorithm will ditch the current path and go back to the previous option, thus preventing itself from going down a long, inefficient path that *h(n)* erroneously marked as best.

Yet again, since this algorithm, too, relies on a heuristic, it is as good as the heuristic that it employs. It is possible that in some situations it will be less efficient than *greedy best-first* search or even the *uninformed* algorithms. For *A* search* to be optimal, the heuristic function, *h(n),* should be:

1. *Admissible,* or never *overestimating* the true cost, and
2. *Consistent,* which means that the estimated path cost to the goal of a new node in addition to the cost of transitioning to it from the previous node is greater or equal to the estimated path cost to the goal of the previous node. To put it in an equation form, *h(n)* is consistent if for every node *n* and successor node *n'* with step cost *c*, $h(n) \leq h(n') + c$.

Adversarial Search

Whereas, previously, we have discussed algorithms that need to find an answer to a question, in **adversarial search** the algorithm faces an opponent that tries to achieve the opposite goal. Often, AI that uses adversarial search is encountered in games, such as tic tac toe.

Minimax

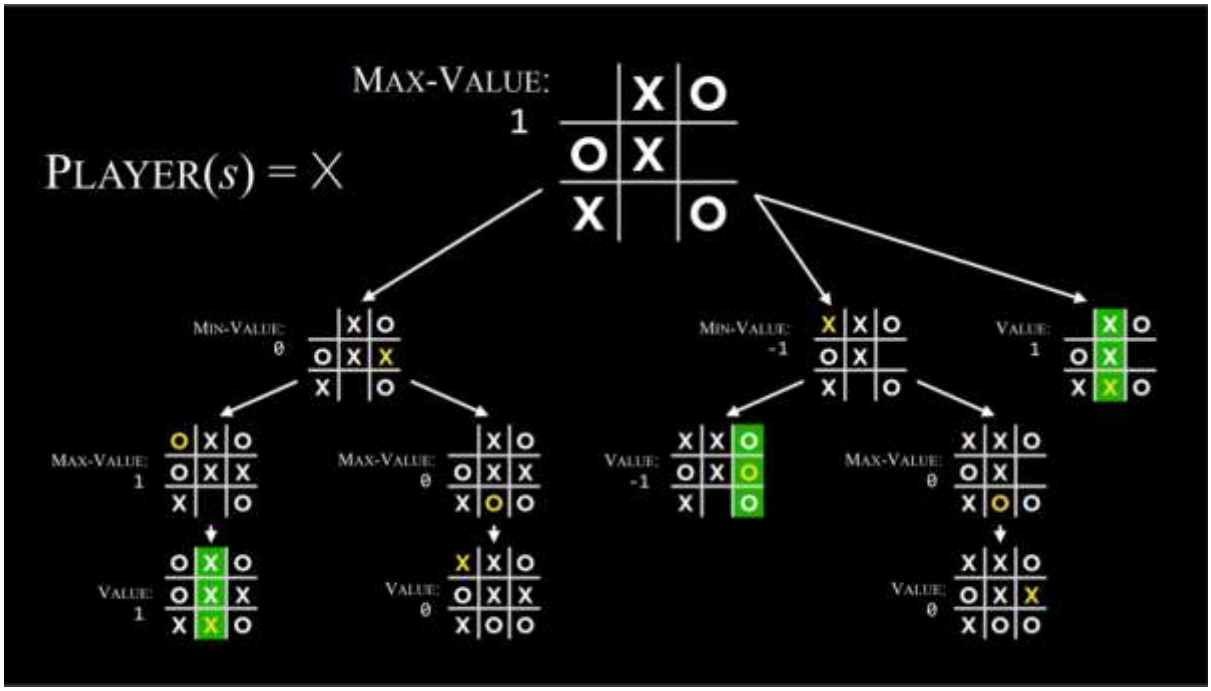
A type of algorithm in adversarial search, **Minimax** represents winning conditions as (-1) for one side and (+1) for the other side. Further actions will be driven by these conditions, with the minimizing side trying to get the lowest score, and the maximizer trying to get the highest score.

Representing a Tic-Tac-Toe AI:

- S_0: Initial state (in our case, an empty 3X3 board)
- *Players(s)*: a function that, given a state *s*, returns which player's turn it is (X or O).
- *Actions(s)*: a function that, given a state *s*, return all the legal moves in this state (what spots are free on the board).
- *Result(s, a)*: a function that, given a state *s* and action *a*, returns a new state. This is the board that resulted from performing the action *a* on state *s* (making a move in the game).
- *Terminal(s)*: a function that, given a state *s*, checks whether this is the last step in the game, i.e. if someone won or there is a tie. Returns *True* if the game has ended, *False* otherwise.
- *Utility(s)*: a function that, given a terminal state *s*, returns the utility value of the state: -1, 0, or 1.

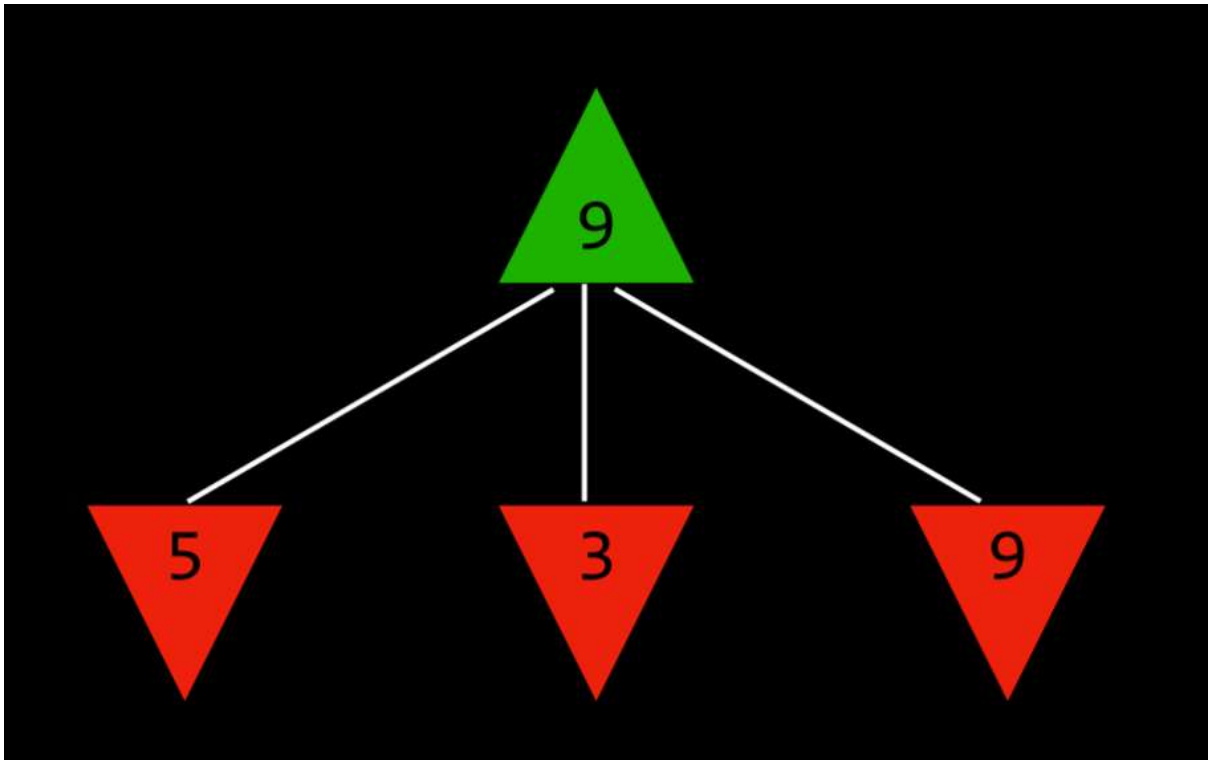
How the algorithm works:

Recursively, the algorithm simulates all possible games that can take place beginning at the current state and until a terminal state is reached. Each terminal state is valued as either (-1), 0, or (+1).



Minimax Algorithm in Tic Tac Toe

Knowing based on the state whose turn it is, the algorithm can know whether the current player, when playing optimally, will pick the action that leads to a state with a lower or a higher value. This way, alternating between minimizing and maximizing, the algorithm creates values for the state that would result from each possible action. To give a more concrete example, we can imagine that the maximizing player asks at every turn: "if I take this action, a new state will result. If the minimizing player plays optimally, what action can that player take to bring to the lowest value?" However, to answer this question, the maximizing player has to ask: "To know what the minimizing player will do, I need to simulate the same process in the minimizer's mind: the minimizing player will try to ask: 'if I take this action, what action can the maximizing player take to bring to the highest value?'" This is a recursive process, and it could be hard to wrap your head around it; looking at the pseudo code below can help. Eventually, through this recursive reasoning process, the maximizing player generates values for each state that could result from all the possible actions at the current state. After having these values, the maximizing player chooses the highest one.



The Maximizer Considers the Possible Values of Future States.

To put it in pseudocode, the Minimax algorithm works the following way:

- Given a state *s*
    - The maximizing player picks action *a* in *Actions(s)* that produces the highest value of *Min-Value(Result(s, a))*.
    - The minimizing player picks action *a* in *Actions(s)* that produces the lowest value of *Max-Value(Result(s, a))*.
- Function *Max-Value(state)*
    - *v* = -∞
    - if *Terminal(state)*:

      return *Utility(state)*

    - for *action* in *Actions(state)*:

      v = Max(v, Min-Value(Result(state, action)))

      return *v*

- Function *Min-Value(state)*:
    - *v* = ∞
    - if *Terminal(state)*:

      return *Utility(state)*

    - for *action* in *Actions(state)*:

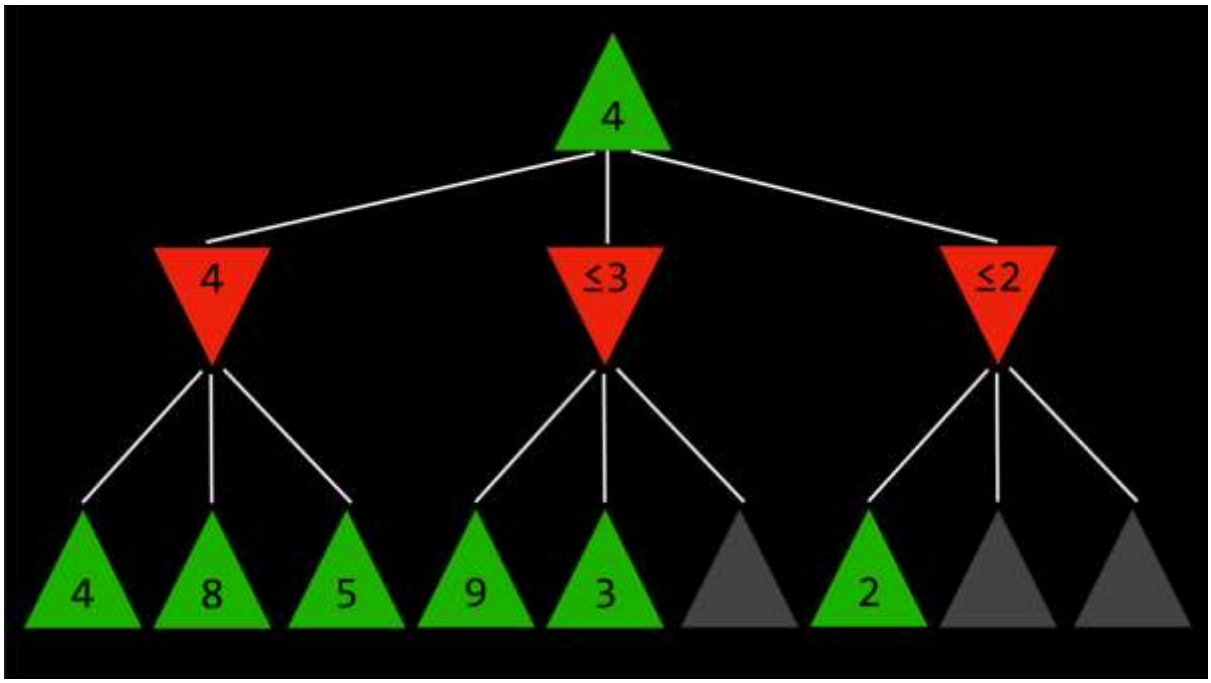
      v = Min(v, Max-Value(Result(state, action)))

      return *v*

Alpha-Beta Pruning

A way to optimize *Minimax*, **Alpha-Beta Pruning** skips some of the recursive computations that are decidedly unfavorable. After establishing the value of one action, if there is initial evidence that the following action can bring the opponent to get to a better score than the already established action, there is no need to further investigate this action because it will decidedly be less favorable than the previously established one.

This is most easily shown with an example: a maximizing player knows that, at the next step, the minimizing player will try to achieve the lowest score. Suppose the maximizing player has three possible actions, and the first one is valued at 4. Then the player starts generating the value for the next action. To do this, the player generates the values of the minimizer's actions if the current player makes this action, knowing that the minimizer will choose the lowest one. However, before finishing the computation for all the possible actions of the minimizer, the player sees that one of the options has a value of three. This means that there is no reason to keep on exploring the other possible actions for the minimizing player. The value of the not-yet-valued action doesn't matter, be it 10 or (-10). If the value is 10, the minimizer will choose the lowest option, 3, which is already worse than the preestablished 4. If the not-yet-valued action would turn out to be (-10), the minimizer will this option, (-10), which is even more unfavorable to the maximizer. Therefore, computing additional possible actions for the minimizer at this point is irrelevant to the maximizer, because the maximizing player already has an unequivocally better choice whose value is 4.



Depth-Limited Minimax

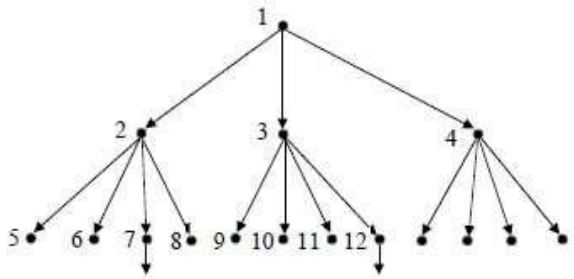
There is a total of 255,168 possible Tic Tac Toe games, and 10^{29000} possible games in Chess. The minimax algorithm, as presented so far, requires generating all hypothetical games from a certain point to the terminal condition. While computing all the Tic-Tac-Toe games doesn't pose a challenge for a modern computer, doing so with chess is currently impossible.

Depth-limited Minimax considers only a pre-defined number of moves before it stops, without ever getting to a terminal state. However, this doesn't allow for getting a precise value for each action, since the end of the hypothetical games has not been reached. To deal with this problem, *Depth-limited Minimax* relies on an **evaluation function** that estimates the expected utility of the game from a given state, or, in other words, assigns values to states. For example, in a chess game, a utility function would take as input a current configuration of the board, try to assess its expected utility (based on what pieces each player has and their locations on the board),

and then return a positive or a negative value that represents how favorable the board is for one player versus the other. These values can be used to decide on the right action, and the better the evaluation function, the better the Minimax algorithm that relies on it.

Uniformed or Blind search

- **Breadth First Search (BFS):** BFS expands the leaf node with the lowest path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple queue ("first in, firstout").



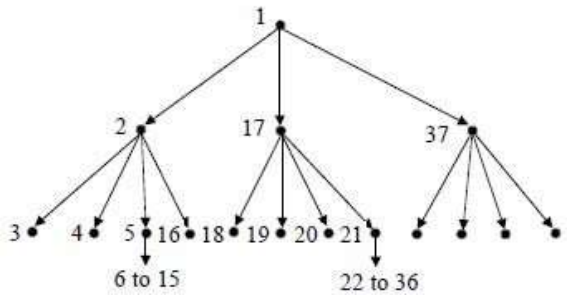
- This is guaranteed to find an optimal path to a goal state. It is memory intensive if the state space is large. If the typical branching factor is b, and the depth of the shallowest goal state is d – the space complexity is $O(b^d)$, and the time complexity is$O(b^d)$.
- BFS is an easy search technique to understand. The algorithm is presented below.

```
breadth_first_search()
{
    store initial state in queue Q
    set state in the front of the Q as current state ;
    while (goal state is reached OR Q is empty)
    {
        apply rule to generate a new state from the current state
        ;
        if (new state is goal state) quit ;
            else if (all states generated from current states are
            exhausted)
{
                delete the current state from the Q ;
                set front element of Q as the current state ;
}
    else continue ;
                }
}
```

- The algorithm is illustrated using the bridge components configuration problem. The initial state is PDFG, which is not a goal state; and hence set it as the current state. GenerateanotherstateDPFG(byswapping1stand2ndpositionvalues)andadditto

the list. That is not a goal state, hence; generate next successor state, which is FDPG (by swapping 1st and 3rd position values). This is also not a goal state; hence add it to the list and generate the next successor state GDFP.

- Only three states can be generated from the initial state. Now the queue Q will have three elements in it, viz., DPFG, FDPG and GDFP. Now take DPFG (first state in the list) as the current state and continue the process, until all the states generated from this are evaluated. Continue this process, until the goal state DGPF isreached.
- The 14th evaluation gives the goal state. It may be noted that, all the states at one levelin the tree are evaluated before the states in the next level are taken up; i.e., the evaluations are carried out breadth-wise. Hence, the search strategy is called breadth- firstsearch.
- **Depth First Search (DFS):** DFS expands the leaf node with the highest path cost sofar, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple stack ("last in, firstout").

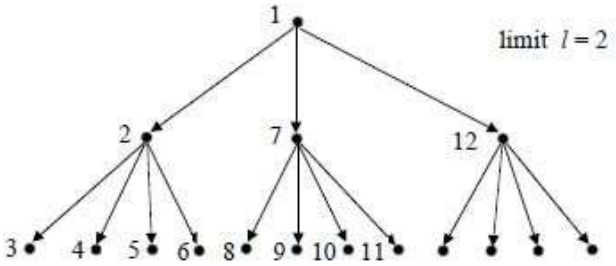


- This is not guaranteed to find any path to a goal state. It is memory efficient evenif the state space is large. If the typical branching factor is b, and the maximum depth of the tree is m – the space complexity is O(bm), and the time complexity is $O(b^m)$.
- In DFS, instead of generating all the states below the current level, only the first state below the current level is generated and evaluated recursively. The search continues till a further successor cannot begenerated.
- Then it goes back to the parent and explores the next successor. The algorithm isgivenbelow.

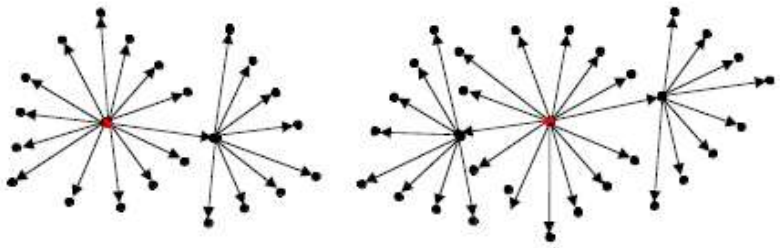
```
depth_first_search ()
{
        set initial state to current state ;
        if (initial state is current state) quit ;
```

```
        else
        {
            if (a successor for current state exists)
            {
                generate a successor of the current state and set
                it as current state ;
            }
            else return ;
            depth_first_search (current_state) ;
            if (goal state is achieved) return ;
            else continue ;
        }
    }
```

- Since DFS stores only the states in the current path, it uses much less memory during the search compared toBFS.
- The probability of arriving at goal state with a fewer number of evaluations is higher with DFS compared to BFS. This is because, in BFS, all the states in a level have to be evaluated before states in the lower level are considered. DFS is very efficient when more acceptable solutions exist, so that the search can be terminated once the first acceptable solution isobtained.
- BFS is advantageous in cases where the tree is verydeep.
- An ideal search mechanism is to combine the advantages of BFS and DFS.
- **Depth Limited Search (DLS):** DLS is a variation of DFS. If we put a limit l on how deep a depth first search can go, we can guarantee that the search will terminate (either in success orfailure).



- If there is at least one goal state at a depth less than l, this algorithm is guaranteed to find a goal state, but it is not guaranteed to find an optimal path. The space complexityis O(bl), and the time complexity is$O(b^l)$.
- **Depth First Iterative Deepening Search (DFIDS):** DFIDS is a variation of DLS. If the lowest depth of a goal state is not known, we can always find the best limit l for DLS by trying all possible depths l = 0, 1, 2, 3, … in turn, and stopping once we have achieved a goalstate.
- This appears wasteful because all the DLS for l less than the goal level are useless, and many states are expanded many times. However, in practice, most of the time is spent at the deepest part of the search tree, so the algorithm actually combines the benefits of DFS andBFS.
- Because all the nodes are expanded at each level, the algorithm is complete and optimal like BFS, but has the modest memory requirements of DFS. Exercise: if we had plenty of memory, could/should we avoid expanding the top level states many times?
- The space complexity is O(bd) as in DLS with l = d, which is better thanBFS.
- The time complexity is $O(b^d)$ as in BFS, which is better thanDFS.
- **Bi-Directional Search (BDS):** The idea behind bi-directional search is to search simultaneously both forward from the initial state and backwards from the goal state, and stop when the two BFS searches meet in themiddle.



- This is not always going to be possible, but is likely to be feasible if the state transitions are reversible. The algorithm is complete and optimal, and since thetwo

search depths are ~d/2, it has space complexity $O(b^{d/2})$, and time complexity $O(b^{d/2})$. However, if there is more than one possible goal state, this must be factored into the complexity.

- **Repeated States:** In the above discussion we have ignored an important complication that often arises in search processes – the possibility that we will waste time by expanding states that have already been expanded before somewhere else on the searchtree.
- For some problems this possibility can never arise, because each state can only be reached in oneway.
- For many problems, however, repeated states are unavoidable. This will include all problems where the transitions are reversible,e.g.

$$((1\,2\,3)) \rightarrow ((1)(2\,3)) \rightarrow ((1\,2\,3)) \rightarrow ((1)(2\,3)) \rightarrow ((1\,2\,3)) \rightarrow \ldots$$

- The search trees for these problems are infinite, but if we can prune out the repeated states, we can cut the search tree down to a finite size, We effectively only generate a portion of the search tree that matches the state spacegraph.
- **Avoiding Repeated States:** There are three principal approaches for dealing with repeatedstates:
  - Never return to the state you have just comefrom

    The node expansion function must be prevented from generating any node successor that is the same state as the node's parent.

  - Never create search paths with cycles inthem

    The node expansion function must be prevented from generating

    any node successor that is the same state as any of the node's ancestors.

  - Never generate states that have already been generatedbefore

    This requires that every state ever generated is remembered, potentially resulting in space complexity of $O(b^d)$.

- **Comparing the Uninformed Search Algorithms:** We can now summarize the properties of our five uninformed searchstrategies:

Strategy	Complete	Optimal	Time Complexity	Space Complexity
BFS	Yes	Yes	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
DLS	If $l \geq d$	No	$O(b^l)$	$O(bl)$
DFIDS	Yes	Yes	$O(b^d)$	$O(bd)$
BDS	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

- Simple BFS and BDS are complete and optimal but expensive with respect to space andtime.
- DFS requires much less memory if the maximum tree depth is limited, but has no guarantee of finding any solution, letalone an optimal one. DLS offers an improvement over DFS if we have some idea how deep the goalis.
- The best overall is DFID which is complete, optimal and has low memory requirements, but still exponential time.

Informed search

- Informed search uses some kind of evaluation function to tell us how far each expanded state is from a goal state, and/or some kind of heuristic function to help us decide which state is likely to be the best one to expandnext.
- The hard part is to come up with good evaluation and/or heuristic functions. Often there is a natural evaluation function, such as distance in miles or number objects in the wrongposition.
- Sometimes we can learn heuristic functions by analyzing what has worked well in similar previoussearches.
- The simplest idea, known as greedy best first search, is toexpand the node that is already closest to the goal, as that is most likely to lead quickly to a solution. This is like DFS in that it attempts to follow a single route to the goal, only attempting to try a different route when it reaches a dead end. As with DFS, it is not complete, not optimal, and has time and complexity of $O(b^m)$. However, with good heuristics, the time complexity can be reducedsubstantially.
- **Branch and Bound:** An enhancement ofbacktracking.
- Applicable to optimizationproblems.

- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution).
- Uses the boundfor:
  - Ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen sofar.
  - Guiding the search throughstate-space.
- The search path at the current node in a state-space tree can be terminated for any one of the following threereasons:
  - The value of the node's bound is not better than the value of the best solution seen sofar.
  - The node represents no feasible solutions because the constraints of the problem are alreadyviolated.
  - The subset of feasible solutions represented by the node consists of a single point and hence we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution isbetter.

- **Best-Firstbranch-and-bound:**
  - A variation ofbacktracking.
  - Among all the nonterminatedleaves, called as the live nodes, in the current tree, generate all the children of the most promising node, instead of generation a single child of the last promising node as itis done in backtracking.
  - Consider the node with the best bound as the most promisingnode.
- **A* Search:** Suppose that, for each node n in a search tree, an evaluation function f(n) is defined as the sum of the cost g(n) to reach that node from the start state, plus an estimated cost h(n) to get from that state to the goal state. That f(n) is then the estimated cost of the cheapest solution throughn.
- A* search, which is the most popular form of best-first search, repeatedly picks the node with the lowest f(n) to expand next. It turns out that if the heuristic function h(n) satisfies certain conditions, then this strategy is both complete andoptimal.
- In particular, if h(n) isan admissible heuristic, i.e. is always optimistic and never overestimates the cost to reach the goal, then A* isoptimal.

- .

Searching And-Or graphs

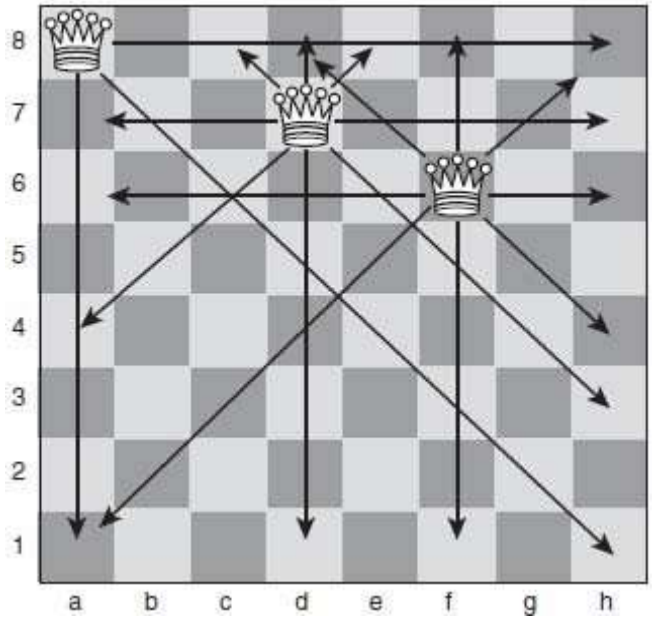
- The DFS and BFS strategies for OR trees and graphs can be adapted for And-Ortrees
- The main difference lies in the way termination conditions are determined, since all goals following an And node must be realized, whereas a single goal node following an Or node willdo
- A more general optimal strategy is AO* (O for ordered)algorithm
- As in the case of the A* algorithm, we use the open list to hold nodes that have been generated but not expanded and the closed list to hold nodes that have beenexpanded
- The algorithm is a variation of the original given byNilsson
- It requires that nodes traversed in the tree be labeled as solved or unsolved in the solution process to account for And node solutions which require solutions to all successorsnodes.
- A solution is found when the start node is labeled assolved

- The AO*algorithm
  - Step 1: Place the start node s onopen
  - Step 2: Using the search tree constructed thus far, compute the most promising solution treeT_0
  - Step 3:Select a node n that is both on open and a part of T_0. Remove n from open and place it onclosed
  - Step 4: If n iaterminal goal node, label n as solved. If the solution of n results in any of n's ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where $T_0$is the solution tree. Remove from open all nodes with a solvedancestor
  - Step 5: If n is not a solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. If any of n's ancestors become unsolvable because n is, label them unsolvable as well. Remove from open all nodes with unsolvableancestors
  - Otherwise, expand node n generating all of its successors. For each such successor node that contains more than one subproblem, generate their successors to give individual subproblems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h* for each newly generated node and place all such nodes thst do not yet have descendents on open. Next recomputed the values oh h* at n and each ancestors ofn
  - Step 7: Return to step2
- It can be shown that AO* will always find a minimum-cost solution tree if one exists, provided only that $h^*(n) \leq h(n)$, and all arc costs are positive. Like A*, the efficiency depends on how closely h* approximatesh

Constraint Satisfaction Search

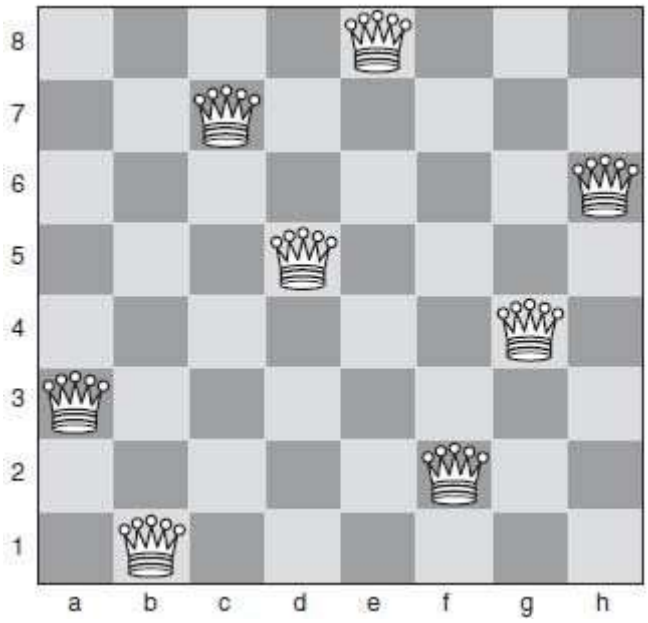
- Search can be used to solve problems that are limited by constraints, such as the eight-queens problem. Such problems are often known as Constraint Satisfaction Problems, or CSPs.I

- n this problem, eight queens must be placed on a chess board in such a way that no two queens are on the same diagonal, row, or column. If we use traditional chess board notation, we mark the columns with letters from a tog and the rows with numbers from 1 to 8. So, a square can be referred to by a letter and a number, such as a4 org7.
- This kind of problem is known as a constraint satisfaction problem (CSP) because a solution must be found that satisfies theconstraints.
- In the case of the eight-queens problem, a search tree can be built that represents the possible positions of queens on the board. One way to represent this is to have a tree that is 8-ply deep, with a branching factor of 64 for the first level, 63 for the next level, and so on, down to 57 for the eighthlevel.
- A goal node in this tree is one that satisfies the constraints that no two queens can be on the same diagonal, row, orcolumn.
- An extremely simplistic approach to solving this problem would be to analyze every possible configuration until one was found that matched theconstraints.
- A more suitable approach to solving the eight-queens problem would be to use depth-first search on a search tree that represents the problem in the followingmanner:
    - The first branch from the root node would represent the first choice of a square for a queen. The next branch from these nodes would represent choices of where to place the secondqueen.
    - The first level would have a branching factor of 64 because there are 64 possible squares on which to place the first queen. The next level would have a somewhat lower branching factor because once a queen has been placed, the constraints can be used to determine possible squares upon which the next queen can beplaced.
    - The branching factor will decrease as the algorithm searches down the tree. At some point, the tree will terminate because the path being followed will lead to a position where no more queens can be placed on legal squares on the board, and there are still some queensremaining.



In fact, because each row and each column must contain exactly one queen, the branching factor can be significantly reduced by assuming that the first queen must be placed in row 1, the second in row 2, and so on. In this way, the first level will have a branching factor of 8 (a choice of eight squares on which the first queen can be placed), the next 7, the next 6, and so on.

- The search tree can be further simplified as each queen placed on the board "uses up" a diagonal, meaning that the branching factor is only 5 or 6 after the first choice has been made, depending on whether the first queen is placed on an edge of the board (columns a or h) ornot.
- The next level will have a branching factor of about 4, and the next may have a branching factor of just 2, as shown in Fig6.1.
- The arrows in Fig 6.1 show the squares to which each queen canmove.
- Note that no queen can move to a square that is already occupied by anotherqueen.



- In Fig 6.1, the first queen was placed in column a of row 8, leaving six choices for the next row. The second queen was placed in column d of row 7, leaving four choices for row 6. The third queen was placed in column f in row 6, leaving just two choices (column c or column h) for row5.
- Using knowledge like this about the problem that is being solved can help to significantly reduce the size of the search tree and thus improve the efficiency of the searchsolution.
- A solution will be found when the algorithm reaches depth 8 and successfully places the final queen on a legal square on theboard.
- A goal node would be a path containing eight squares such that no two squares shared a diagonal, row, orcolumn.
- One solution to the eight-queens problem is shown in above Fig.
- Note that in this solution, if we start by placing queens on squares e8, c7, h6, and then d5, once the fourth queen has been placed, there are only two choices for placing the fifth queen (b4 or g4). If b4 is chosen, then this leaves no squares that could be chosen for the final three queens to satisfy the constraints. If g4 is chosen for the fifth queen, as has been done in Fig 6.2, only one square is available for the sixth queen (a3), and thefinaltwochoicesaresimilarly constrained.So,itcanbeseen thatby applyingthe

constraints appropriately, the search tree can be significantly reduced for this problem.

- Using chronological backtracking in solving the eight-queens problem might not be the most efficient way to identify a solution because it will backtrack over moves that did not necessarily directly lead to an error, as well as ones that did. In this case, nonchronological backtracking, or dependency-directed backtracking could be more useful because it could identify the steps earlier in the search tree that caused the problem further down thetree.

- oncept of “family” and the concept of“country.”