VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering (SCOPE)

MCSE601L- ARTIFICIAL INTELLIGENCE
DIGITAL ASSIGNMENT-1

Course Name : ARTIFICIAL INTELLIGENCE
Course Code : MCSE601L
Slot: (L53+L54)
Course instructor: SIVA SANKARI S

Name : NIDHI SINGH
Registration no. : 22MAI0015

Outline

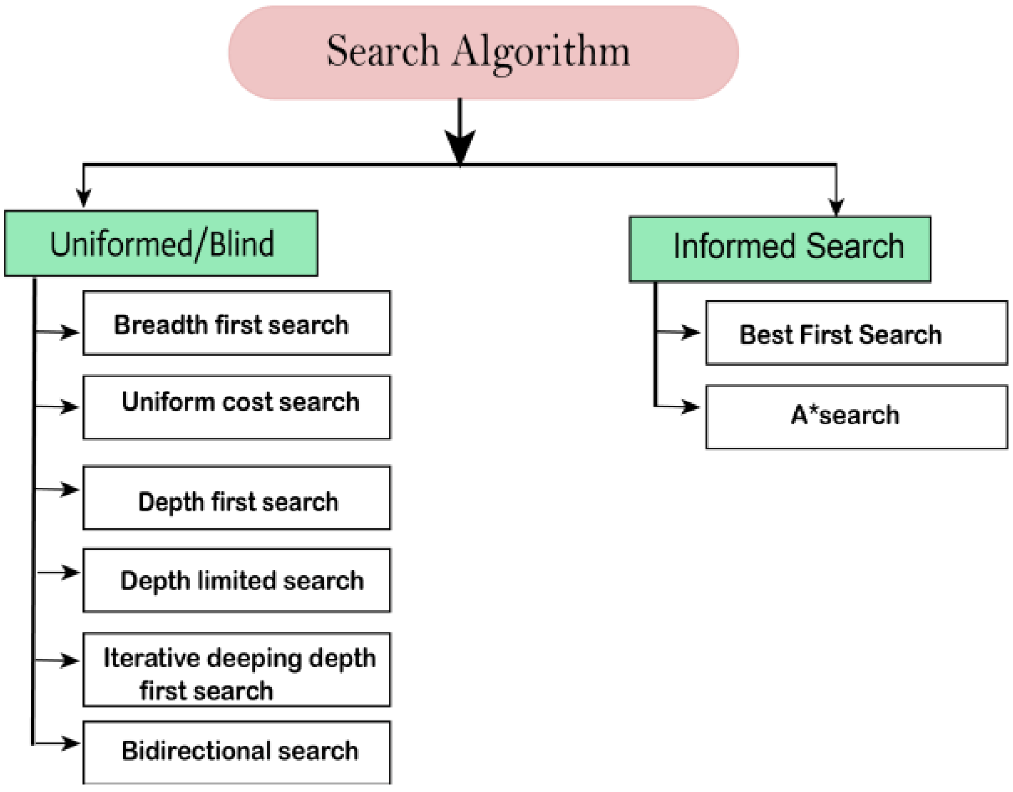
Search Algorithms in AI

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

- A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.
- This plan is achieved through search algorithms.
- Search algorithms are used in artificial intelligence to address a range of issues. There are two types of search algorithms: blind search, also known as uninformed search, and heuristic search, also known as informed search.

Types of search algorithms:

There are far too many effective search algorithms available to cover them all in one post. Instead, six essential search algorithms will be covered in this article, which are separated into two groups as shown below.



1. Informed search

1.1. Introduction :-

An intelligent search algorithm takes into account a wide range of information, including the distance to the target, the cost of the path, the best way to get there, etc. Agents can find the objective node more quickly and efficiently by using this knowledge to explore less of the search space. Finding answers can be done more quickly using a method that goes beyond just the definition of the problem. For vast search spaces, the informed search algorithm is more beneficial. Heuristic search is another name for informed search, which also uses the heuristic concept. The term "informed search" refers to search algorithms that assist in navigating large databases with specific information about the intended outcome of the search. These algorithms are most frequently used in large databases because uninformed search algorithms struggle to curate accurate results in smaller databases.

Algorithm 1

```
General Tree Search Paradigm of Innformed Search
function TreeSearch(RootNode)
Input: Search Tree
Output: solution on success, failure otherwise
1. frontier ← Successors(RootNode)
2. while(notempty(frontier))
3. node ← RemoveFirst(frontier) /* smallest f value */
4. state ← state(node)
5. if(GoalTest(state))
      return solution(node)
6. frontier ← InsertAll(Successors(RootNode))
7. return failure
end TreeSearch
```

1.1.1 Evaluation Function and Heuristic Functions

Heuristics are a type of function that are used in informed search to determine the most promising route. It generates an estimate of how far away the agent is from the goal using the agent's current condition as input. The heuristic approach, however, ensures that a good solution will be found in a fair amount of time, even though it may not always provide the greatest option. The proximity of a state to the goal is estimated via a heuristic function. It computes the price of the best route between the two states and is denoted by h(n). Heuristic function values are consistently positive.

We use three functions: f(n), g(n), and h(n). Evaluation function,

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

where n is the current node.
Purpose of these functions are as follows:
g(n) = cost incurred so far to reach n,
h(n) = estimated cost from n to goal,
f(n) = estimated total cost of path through n to goal.
f() is the evaluation function and h() = is the heuristic function

Admissibility of the heuristic function is given as:

$$\boxed{h(n) <= h^*(n)}$$

h(n) = estimated cost of the cheapest pathfrom node n to a goal node.

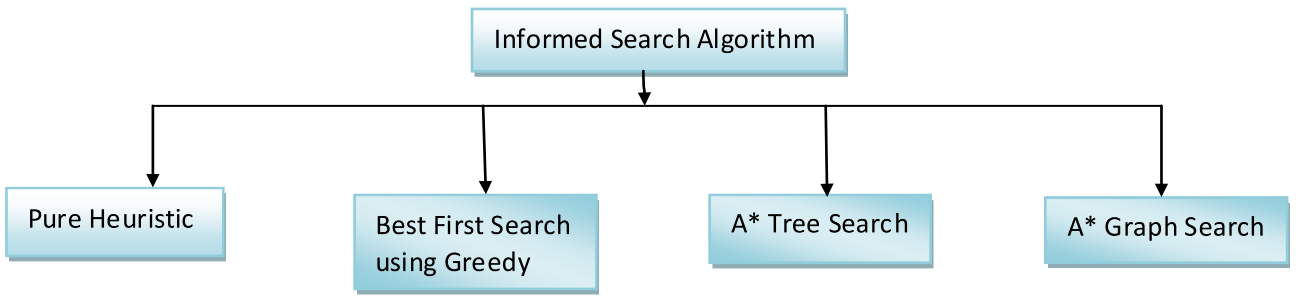
Example:
In route planning the estimate ofthe cost of the cheapest path might be thestraight line distance between two cities
Constraint:
if n is a goal node, then h(n)=0
Here, heuristic cost is denoted by h(n), while estimated cost is denoted by h*(n). Heuristic costs ought to be lower than or equal to predicted costs as a result.

Significance

The importance of the Informed Search Algorithm among the different elements that enable agents to successfully accomplish the goal and give solutions is exceptionally high. The use of certain prior knowledge or hints as heuristic functions distinguishes the informed search approach. The heuristic enhances the algorithm's effectiveness and potency.It guarantees that agents efficiently seek or move through the search space without compromising the cost, time, or speed of the search. In summary, the Informed Search Algorithm helps agents get above the limitations of Uninformed Search and enhances their ability to find the right answer. The following four criteria can be used to compare the uninformed and informed search strategies that are taken into account in this master's thesis: time complexity, space complexity, completeness, and optimality.

Completeness: This property aims to determine whether the method promises to locate the issue's resolution, if one exists.
Optimality: This characteristic looks at whether the tactic offers the best possible outcome.
Time complexity: This characteristic measures the length of time needed to identify the issue's solution.
Space complexity: This characteristic assesses the memory and physical space needed to execute the search algorithm.

Types of Informed Search Algorithms



Pure Heuristic Search:
The simplest heuristic search algorithm is called pure heuristic search. Nodes are expanded based on their heuristic value, H. (n). It maintains the OPEN list and the CLOSED list. The nodes that have already expanded are listed in the CLOSED list, whereas the nodes that haven't yet expanded are listed in the OPEN list.Every time an iteration occurs, the node with the lowest heuristic value, n, is enlarged to produce all of its descendants, and n is added to the closed list. Until a goal state is found, the algorithm keeps running. A pure heuristic search algorithm is a straightforward search that relies on the heuristic value y h(n) to a node. In a heuristic search, there are two lost created, one open for new but unexpanded nodes and

the other closed for expanded nodes. For each iteration, the node with the smallest heuristic value is enlarged, and all of its "child" nodes are expanded and added to close it. The node with the shortest path is then preserved, and the other nodes are distributed, after that closed list has been subjected to a heuristic function.

1. Best First or ‘Greedy’ Search

We must first grasp the idea of DFS and BFS in order to comprehend the Greedy search. Both methods for determining the shortest path rely on vertex information. BFS employs a queue data structure, whereas DFS uses a stack data structure, to determine the shortest path.By integrating breadth-first and depth-first search methods, best-first search is formed. This enables the best-first search to benefit from both breadth-first and depth-first search advantages. The best-first search discovers the path that has no loops because it does not take into account all of the potential paths in the graph. In best-first search, each node's desirability is determined using the evaluation function f (n), so the most promising and desirable unexpanded node gets expanded. The node that will be selected should therefore have the lowest value of the heuristic function f (n).The greedy best-first search algorithm always chooses the path that appears to be the most appealing at the time. It's the result of combining depth-first and breadth-first search algorithms. It makes use of the heuristic function as well as search. We can combine the benefits of both methods with best-first search. At each step, we can use best-first search to select the most promising node. We expand the node that is closest to the goal node in the best first search method, and the closest cost is determined using a heuristic function,. The priority queue implements the greedy best first algorithm. The most crucial feature of best-first search is the ability to consider visiting additional nodes if necessary after the most promising successor node has been chosen and moved towards. It should be noted that the heuristic function occasionally takes the expensive path as well.

$f(n)= g(n)$

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

Best first search algorithm:

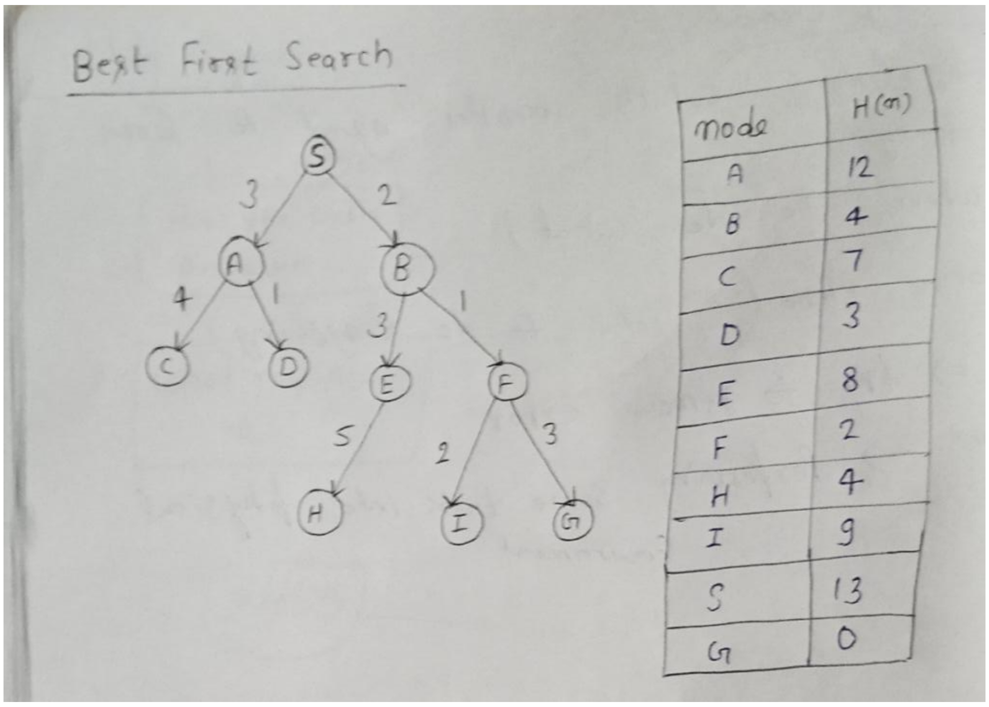
Step 1: Place the initial node in the OPEN list first.
Step 2: Stop and return failure if the OPEN list is empty.
Step 3: Move the node n from the OPEN list to the CLOSED list, as it has the lowest value of h(n).
Step 4: Extend node n and construct node n's descendants.
Step 5: Examine each successor of node n to see whether any of them is a goal node. Return success and end the search if any successor node is a goal node; otherwise, proceed to Step 6.
Step 6: The method checks for the evaluation function f(n) for each successor node, then determines if the node has been in the OPEN or CLOSED list. Add the node to the OPEN l if it isn't already on both lists.
Step 7: return to the step 2

The best-first search has the following properties:

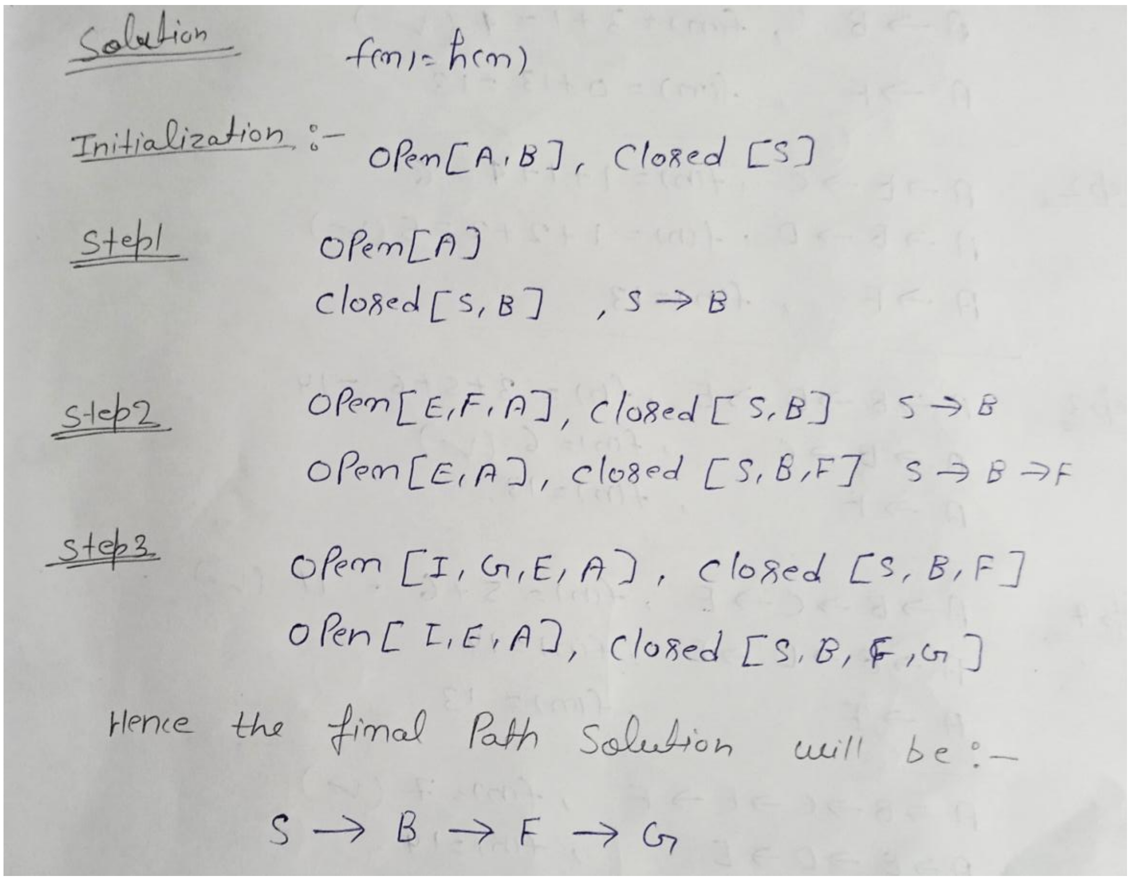
Complete: No; **:** Even if the given state space is finite, greedy best-first search is still imperfect.
Optimal: No.
Time Complexity: O(b^m); where m is the maximum depth of the tree and b is the branching factor.
Space Complexity: The Greedy best first search's worst case space complexity is O(b^m)

Advantages:
This approach outperforms the BFS and DFS algorithms in terms of efficiency by combining the advantages of both algorithms, allowing best first search to transition between BFS and DFS.
Disadvantages:
- This method isn't ideal.
- In the worst situation, it can behave like an unguided depth-first search.
- Similar to DFS, it's possible to get caught in a loop.

Example:
In the following tree, we start traversal from vertex S



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



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

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

Step1 Opem[A]

closed[S, B] , $S \rightarrow B$

Step2 Opem[E, F, A], Closed[S, B] $S \rightarrow B$

Opem[E, A], closed [S, B, F] $S \rightarrow B \rightarrow F$

Step3. Opem [I, G, E, A], Closed [S, B, F]

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

Hence the final Path Solution will be :-

$S \rightarrow B \rightarrow F \rightarrow G$

Application :

The optimum arrangement of these home appliances is chosen using the Best First Search algorithm and the suitable heuristic function. Application of the algorithm keeps demand below the PV system's generation capacity and reduces customer reliance on the grid.

2. A* Tree Search

A* is yet another intelligent search tactic. The most used bestfirst is really A* search. Researchers and scientists have devised a search methodology.Although A* search is a best-first search algorithm, it is not a greedy best-first algorithm. The evaluation function f (n) in the A* search strategy is denoted by f (n) =g(n)+ h(n),
where n is the path's last node and g(n) denotes the cost of the path from the starting point node to the node n, and the heuristic h(n) is utilised to calculate the cost from the node n to the target node. The estimation of f (n), which is actually the result of combining g(n) and h(n). To arrive at a conclusion, the best cost function was applied.Simply put, A* search combines elements and strengths from a greedy search and a Uniform cost search. The heuristic of A*search is the summation of Heuristic cost in greedy search. The most well-known type of best-first search is the A* search. It employs the heuristic function h(n) and the cost of getting from state g to node n. (n). It solves the problem efficiently by combining

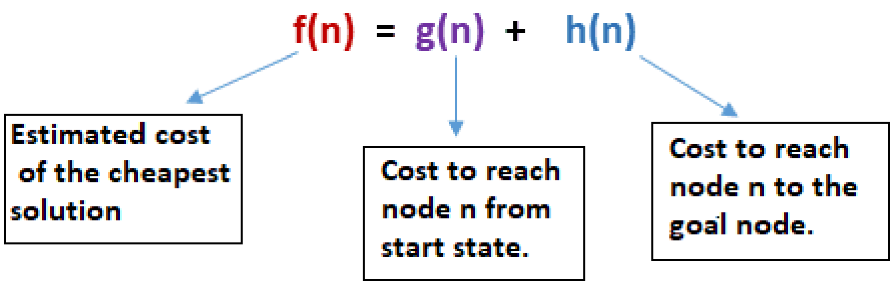
UCS and greedy best-first search features. Using the heuristic function, the A* search algorithm finds the shortest path through the search space. This search algorithm uses a smaller search tree and delivers the best results faster. The A* method is similar to UCS, but instead of g(n)+h(n), it uses

$$F(x)= h(x)+g(x)$$

Where,

- **h(x):** Forward cost referring cost of a node from the current state to the goal state.
- **g(x):** Backward cost which is the cost of a node from the root state or initial state.

The goal is to choose a node with the smallest f in this instance, as shown (x). The fact that the A* search algorithm is a comprehensive search algorithm, as opposed to a simple heuristic method, which just provides the shortest pathways, is one of its main advantages. As a result, it is the most used search algorithm and can solve complex functions in complex search spaces. As was previously mentioned, it has resource limitations and cannot be utilised for really large-scale activities because it stores all node expansion and generation in memory. As part of the A* search method, we use a search heuristic as well as the travel time to the node. This allows us to include both. *A_* is optimally efficient for any given consistent heuristic. In other words,no other optimal search algorithm will expand fewer nodes than *A_*. Complexity of *A_* often makes it impractical for adoption in many large scale applications.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution
Cost to reach node n from start state.
Cost to reach node n to the goal node.

Algorithm of A* search:

Step 1: Place the beginning node in the OPEN list as the first step.
Step 2: Determine whether or not the OPEN list is empty; if it is, return failure and stop.
Step 3: Choose the node with the shortest value of the evaluation function (g+h) from the OPEN list; if node n is the goal node, return success and stop; otherwise, return failure and continue.
Step 4: Expand node n and create all of its descendants, then place n in the closed list. Check whether n' is already in the OPEN or CLOSED list for each successor n'; if not, compute the evaluation function for n' and insert it in the Open list.
Step 5: If node n' is already in the OPEN and CLOSED states, it should be attached to the back pointer, which represents the lowest g(n') value.

A* search has the following properties:

Complete: Yes; The A* algorithm is complete if and only if:

- The number of branching factors is limited.
- Every action has a fixed cost.

Optimal: Yes; A* search method is optimal if it meets the following two criteria:

- **Admissible:** The first requirement for optimality is that h(n) is an admissible A* tree search heuristic. An acceptable heuristic is one that is optimistic.
- **Consistency:** It is the second necessary requirement for only A* graph-search.

Time Complexity: O(b^d) ; where *b* is branching factor, and *d* is depth of the solution;

Space Complexity: O(b^d)

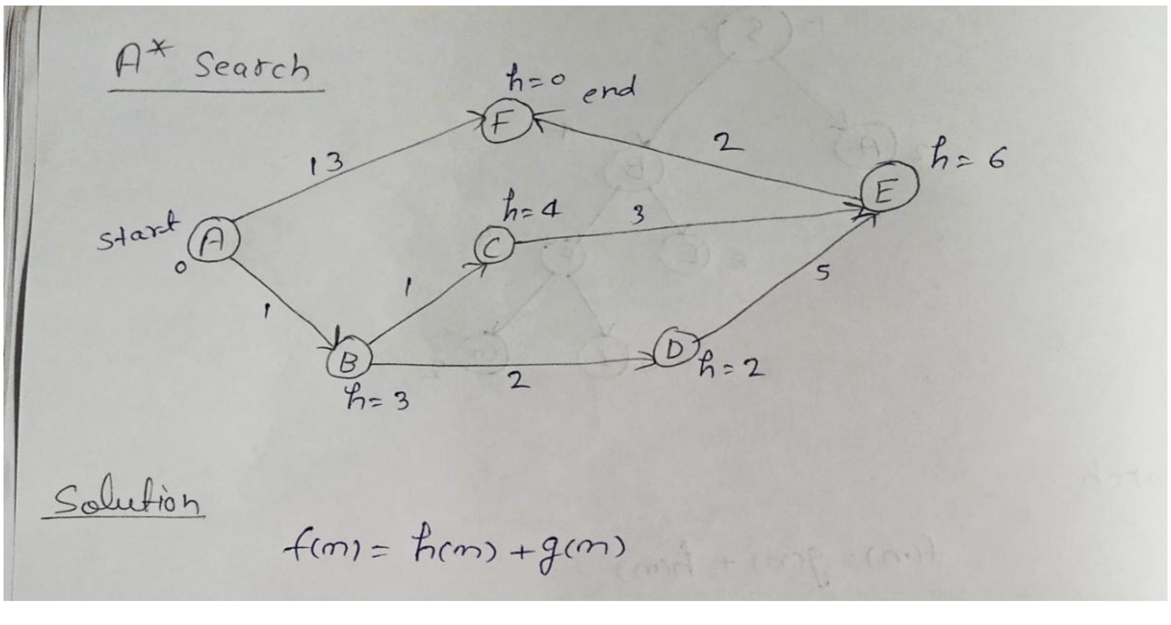
Advantages:

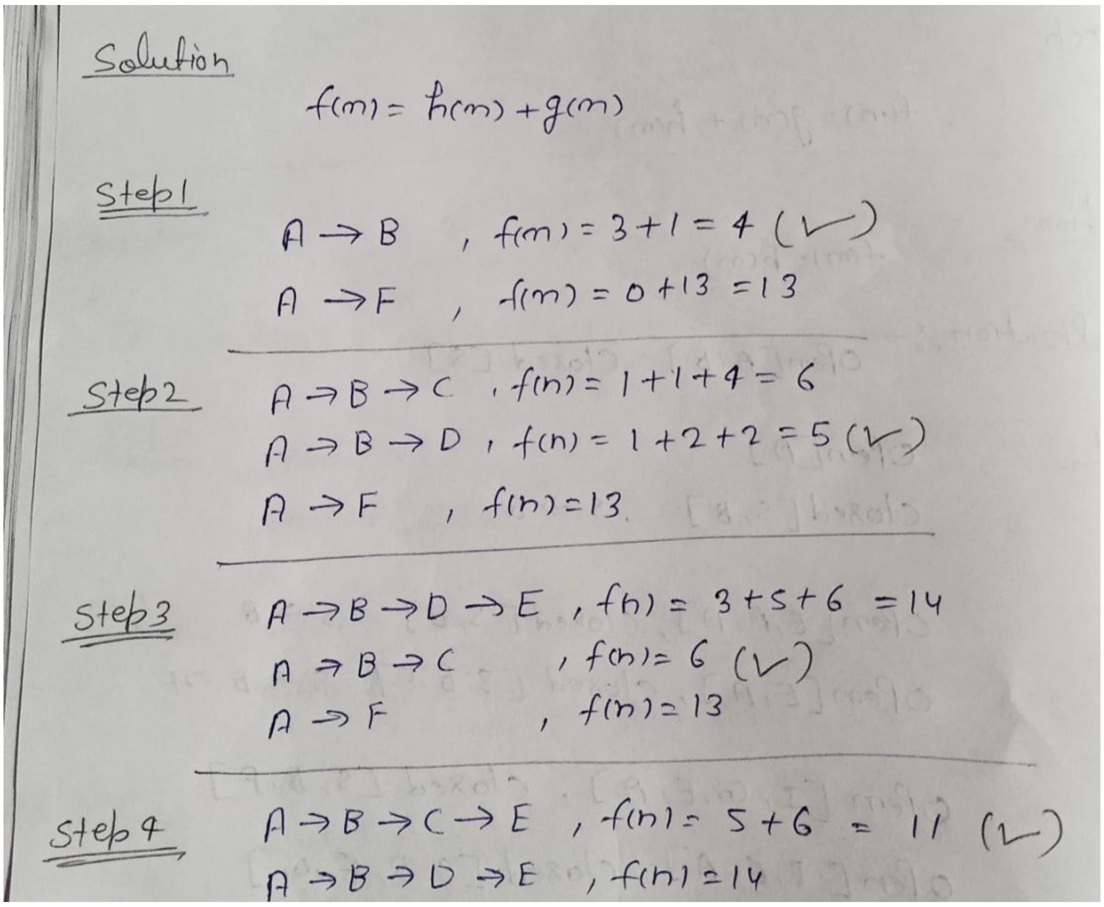
- The A* search algorithm is superior than other search algorithms.
- The A* search algorithm is perfect and thorough.
- Very complex problems can be solved with this technique.

Disadvantages:

- Because it relies on heuristics and approximation, it occasionally does not produce the shortest path.
- There are several complexity issues with the A* search algorithm.
- A*'s primary drawback is that it uses a lot of memory because it keeps track of every node it creates in memory, making it unsuitable for a number of large-scale problems.

Example:





Solution

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

Step1

A → B , $f(n) = 3+1 = 4$ (✓)

A → F , $f(n) = 0+13 = 13$

Step2

A → B → C , $f(n) = 1+1+4 = 6$

A → B → D , $f(n) = 1+2+2 = 5$ (✓)

A → F , $f(n) = 13$

Step3

A → B → D → E , $f(n) = 3+5+6 = 14$

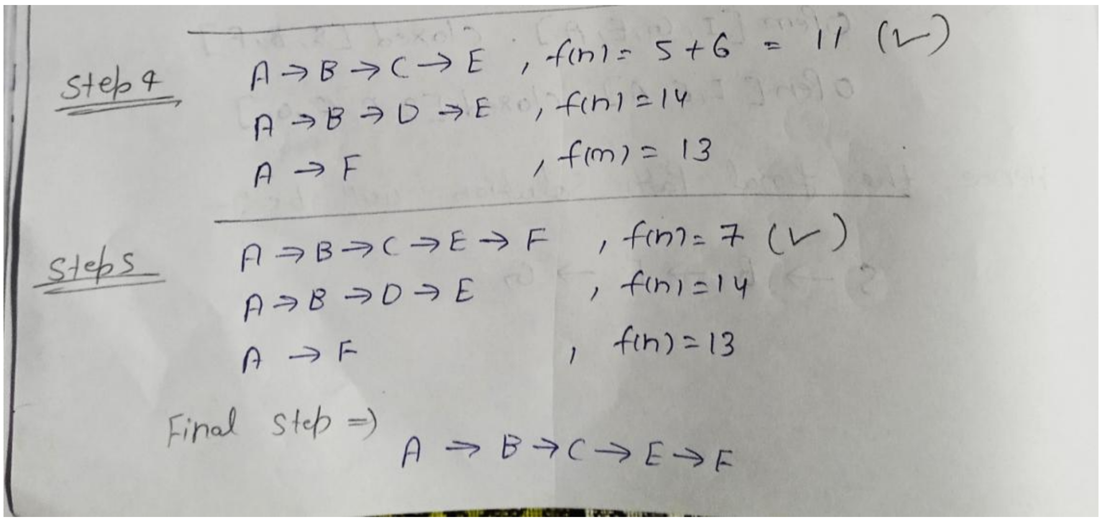
A → B → C , $f(n) = 6$ (✓)

A → F , $f(n) = 13$

Step 4

A → B → C → E , $f(n) = 5+6 = 11$ (✓)

A → B → D → E , $f(n) = 14$



Step 4

A → B → C → E , $f(n) = 5+6 = 11$ (✓)

A → B → D → E , $f(n) = 14$

A → F , $f(n) = 13$

Step5

A → B → C → E → F , $f(n) = 7$ (✓)

A → B → D → E , $f(n) = 14$

A → F , $f(n) = 13$

Final step ⇒ A → B → C → E → F

Final solution :-

A→B→C→E→F

Real world applications of A * Search :

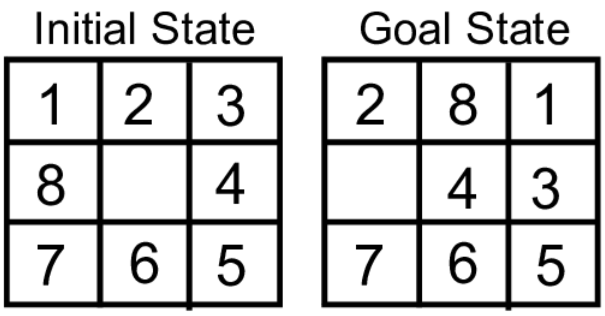
- It can be applied as a path-finding method for applications that employ maps.
- By figuring out the objective state, apps for string searching can also use this.
- This is used by NLP to check for parsing mistakes.
- This algorithm is used by several games for their positioning system.
- It is frequently used in online games and maps to locate the quickest path with the maximum level of effectiveness.
- A* is utilised in a variety of AI applications, including search engines.
- It is included into various algorithms, including the shortest path algorithm by Bellman and Ford.
- To determine the optimum path between two nodes, network routing protocols like RIP, OSPF, and BGP use the A* algorithm.

Pathfinding

The shortest route between two points is plotted by a computer programme in a process known as pathfinding or pathing. It is a more useful variation on navigating mazes. The Dijkstra algorithm for locating the shortest path on a weighted graph forms the foundation of a large portion of this field of study.The shortest path issue in graph theory, which looks at how to find the route that best satisfies certain criteria (shortest, cheapest, fastest, etc.) between two sites in a big network, is closely connected to pathfinding.A* is a Dijkstra's algorithm version that is frequently employed in games. Each open node is given a weight by A* that is equal to the weight of the edge attached to it plus the nearest distance from that node to the finish. This rough distance can be calculated usingA* uses this heuristic to improve on the behavior relative to Dijkstra's algorithm. A* is equivalent to Dijkstra's algorithm when the heuristic evaluates to zero. A* keeps finding the best paths but moves more quickly as the heuristic estimate grows and gets closer to the actual distance (by virtue of examining fewer nodes). A* looks at the fewest nodes when the value of the heuristic is exactly the true distance. (However, it is usually impractical to design a heuristic function that constantly computes the correct distance, as the same comparison result may frequently be reached using simpler calculations - for example, utilising Manhattan distance over Euclidean distance in two-dimensional space.) A* considers fewer nodes as the heuristic's value rises, but it no longer ensures an ideal path. This is acceptable and even desirable in many **applications (like video games).**

Solving 8-Puzzle using A* Algorithm

- A common puzzle made of N tiles is called the N-Puzzle or sliding puzzle.
- A common puzzle called the N-Puzzle or sliding puzzle has N tiles, where N can be any number from 8 to 24. N equals 8 in our example. There are sqrt(N+1) rows and sqrt(N+1) columns in the puzzle. An 8-Puzzle, for instance, will have 3 rows and 3 columns and a 15-Puzzle, 4 rows and 4 columns.
- N tiles make up the problem, and there is one empty area into which the tiles can be placed. The puzzle has Start and Goal configurations, also known as states. In order to achieve the Goal configuration, the puzzle can be solved by placing each tile in the one empty space one at a time.



The goal state can be reached by placing the initial(start) state tiles in the vacant space in a specific order.

The puzzle's rules of solution:

We can imagine moving the empty space in the location of the tile, thus exchanging the tile with the empty space, rather than moving the tiles in the empty space. The void can only move in these four directions:

1. Up
2.Down
3. proper or
4. Left

The empty space can only move one step at a time and cannot move diagonally.

How A* solves the 8-Puzzle problem?

- The empty area is initially moved in all directions in the In order to determine the f-score for each state, we first move the empty space in the start state in all feasible directions.
- Expanding the current state is what it is known as. The newly created states are pushed into the open list after the expanded state has been pushed into the closed list. The state with the lowest f-score is chosen, then expanded one again.
- The desired state becomes the present state at the end of this operation. In essence, what we're doing is giving the algorithm a way to select its course of action.
- The algorithm selects the optimum course of action and follows it.
- Due to the procedure expanding the node with the lowest f-score, the problem of generating redundant child states is resolved.

The Heuristic Value (Cost Function) of an 8 Puzzle State:

An 8 puzzle state's heuristic value is a combination of two values. It's frequently known as the cost function f.

$$f = h + g$$

The goal node's distance is shown by h, and the number of nodes that were travelled to get to the current node is indicated by g. We'll use the Manhattan distance for h and the depth of the current node for g.

The total cost for our A* search will be calculated as the sum of the Manhattan distance and the node's current depth.

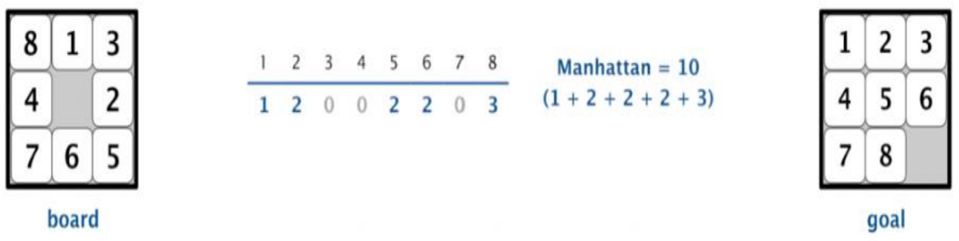
Manhattan distance

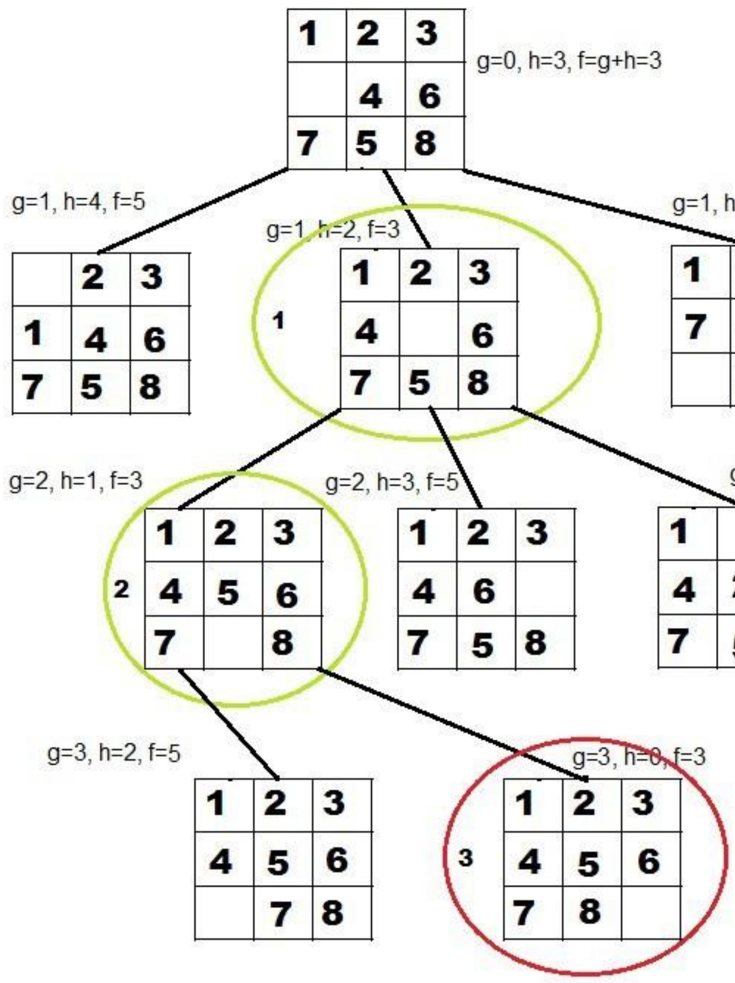
The Manhattan distance heuristic is employed due to its ease of use and capacity to calculate the approximate number of moves needed to transform a specific puzzle state into the solution state. The sum of each tile's separations from its proper location yields the Manhattan distance.

```
function manhattan_distance(node, goal) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return dx + dy
```

As an illustration, the distance in Manhattan between "213540678" and "123456780" is 9 and 21 respectively.

The Manhattan price for a particular tile configuration is depicted in the diagram below.





2. Uninformed Search:

An uninformed search strategy is characterised by the lack of any prior knowledge, specifications, or indications regarding how to resolve the problem. The Uninformed Search Algorithm creates the search tree without any prior knowledge of the domain or any further knowledge about the states than what is given in the problem descriptions. This class-purpose search technique, also known as Blind Search or Brute-Force Search, investigates every root node in the search tree until it reaches the objective state.

Requirements of uninformed search:

Even though uninformed search does not require lot of details but the basic requirements are:

- **State Overview**: states that can be reached from the initial state through any action sequence, in addition to specifics like where the search begins, ends, and the transition model, among other things.
- **A set of valid operations**: That aids in locating the most cost-effective solution.
- **Starting Point**: information regarding the initial stage at which the agent begins the search.
- **Description of the Goal State**: Details like the search tree's limit and the order of nodes from start to goal, among other things.

Significance

To determine the best solution to an issue, one typically looks at how well the proposed approach will work under realistic conditions. Evaluation of search strategies based on performance is prevalent in artificial intelligence. While the number of nodes raised and investigated was connected with memory utilisation and CPU activities, respectively. This suggests that, in terms of memory use, both best first search using mismatched tile and utilising Manhattan distance are much better than breadth-first search, depth-first search, and optimal search. This is due to the significantly lower average number of raised nodes. Additionally, a substantially lower average number of investigated nodes shows that best first search uses less processing resources than other algorithms, making it superior to them in this area. Although depth first search performs poorly in terms of memory use and processing power demand, uninformed search techniques produce a large number of elevated and investigated nodes. The best results in terms of memory usage and processing power are achieved by best first searches that combine the Manhattan distance and the mismatched tile heuristic functions. Heuristic-based informed search generally raises and explores fewer nodes than ignorant search algorithms. This demonstrates how much more effective the informed search algorithm with heuristic is than the ignorant search algorithm.

A lesser percentage of raised nodes must be investigated in order to solve the puzzle because the explored to raised node ratio in uninformed search is equally low. This demonstrates that, in terms of memory use and processing power needs, uninformed search generally outperforms informed search using heuristics while solving the 8-puzzle.

The following four criteria can be used to compare the uninformed and informed search strategies that are taken into account in this master's thesis: time complexity, space complexity, completeness, and optimality.

Completeness: This property aims to determine whether the method promises to locate the issue's resolution, if one exists.

Optimality: This characteristic looks at whether the tactic offers the best possible outcome.

Time complexity: This characteristic measures the length of time needed to identify the issue's solution.

Space complexity: This characteristic assesses the memory and physical space needed to execute the search algorithm.

Breadth-First Search (BFS)

All nodes at level n will be extended first in the breadth-first search tracking method before accessing nodes at level n+1. Beginning at the root node, tracing proceeds at the first level from left to right before continuing at the next level in the same manner. The conditions in the goal state are compared with the conditions at each node that is visited. A solution has not been discovered if the conditions at the visited node differ from the intended circumstances. Then, the tracking procedure is performed on each node to a preset depth. The tracking is halted if, on the other hand, the state of the visited node matches the goal condition, indicating that a solution has been identified. In this search technique, the root node serves as the starting point for expansion. The successors of the root node are then enlarged. Up until the final successor is included, the procedure is repeated. The breadth-first search technique should be implemented using the First in First out (FIFO) data structure.
The performance of the method is assessed using the completeness criterion of the breadth-first search approach. Assume there are branching factors, and the depth is where the shallowest target node is.

To apply the Breadth-First Search method, the pseudo-code that is executed is as follows:

```
1. Give the starting node to the open list L
2. Loop: If the open list L is empty, then tracking is stopped
3. Put n at the beginning of the open list L
4. If n is a goal, then the tracking has been successful
5. Remove n from the open list L
6. Put n on the closed list C
7. Expand n. Give the tail an open list L of all child nodes that have not appeared in open list L or closed list C
and assign a pointer to n
8. Back to Loop
```

The breadth-first search has the following properties:

Complete: Yes
Optimal: Yes
Time Complexity: $O(b^d)$, where is branching factor and is depth of the solution.
Space complexity: $O(b^d)$

Where b- number of nodes ,d- depth of the tree formed.

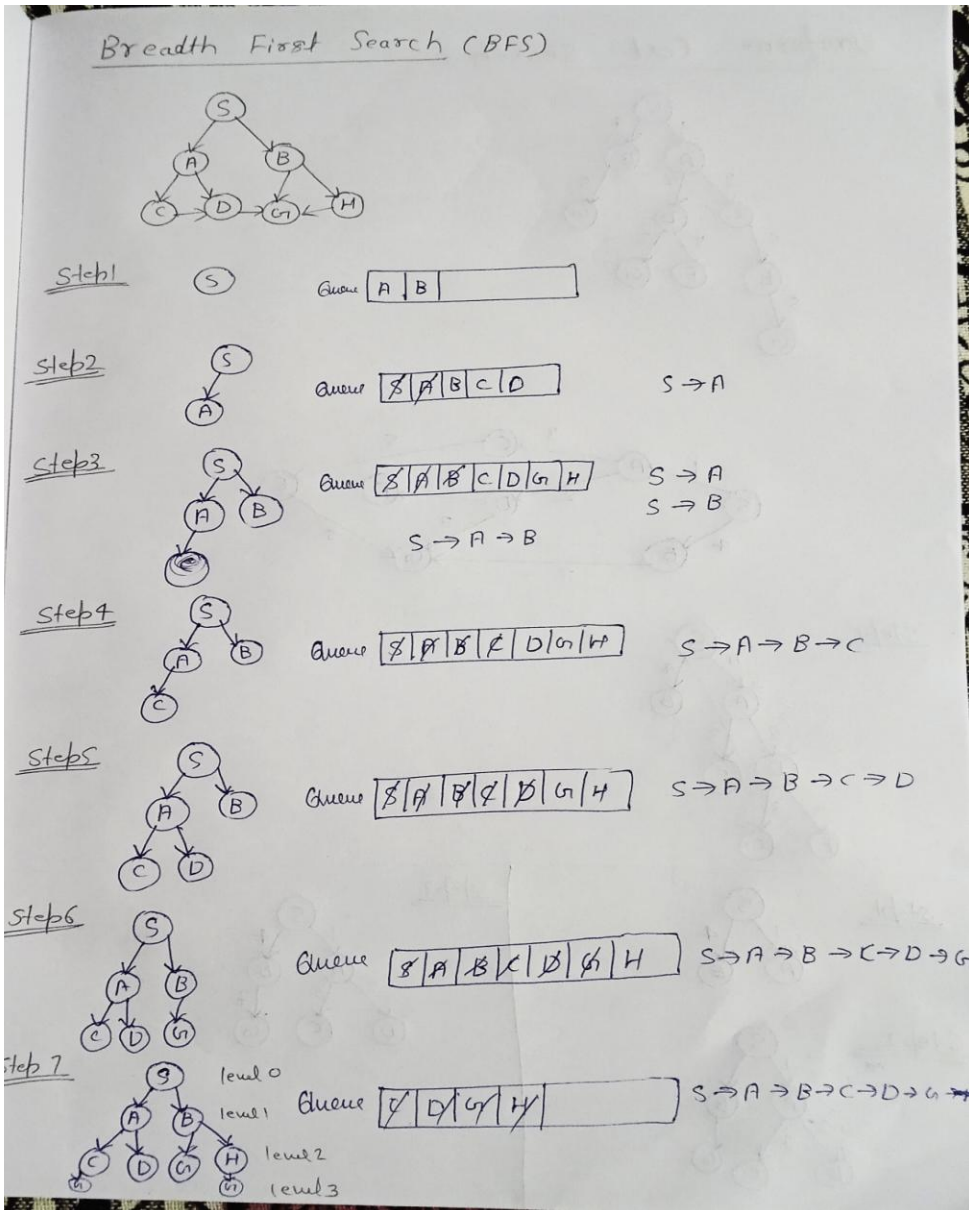
Advantage:

- Guaranteed to find the single solution at the shallowest depth level
- 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.

Disadvantage:

- Suitable for only smallest instances problem (i.e.) (number of levels to be minimum (or) branching factor to be minimum)
- 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 :



Applications of the BFS algorithm

Let's look at some real-world scenarios where a BFS algorithm implementation can be quite successful.

Unweighted Graphs: Using the BFS algorithm, it is simple to find the shortest path and the least spanning tree to accurately and quickly traverse the whole graph.

P2P Networks: BFS can be used to find every node that is close by or nearby in a peer-to-peer network. This will speed up the process of finding the needed information.

Web crawlers: By using BFS, search engines or web crawlers can quickly create many tiers of indexes. The web page serves as the starting point for BFS implementation, after which it visits each link on the page.

Navigational aids: BFS can assist in locating all nearby locations. Network Broadcasting: The BFS algorithm helps a packet that has been broadcast locate and connect to every node for which it has an address.

- BFS may be used to discover the locations nearest to a specific origin point.
- The BFS algorithm is used in peer-to-peer networks as a search technique to discover

all neighboring nodes. uTorrent, BitTorrent, and other similar torrent clients use this method to look for "peers" and "seeds" in the network.

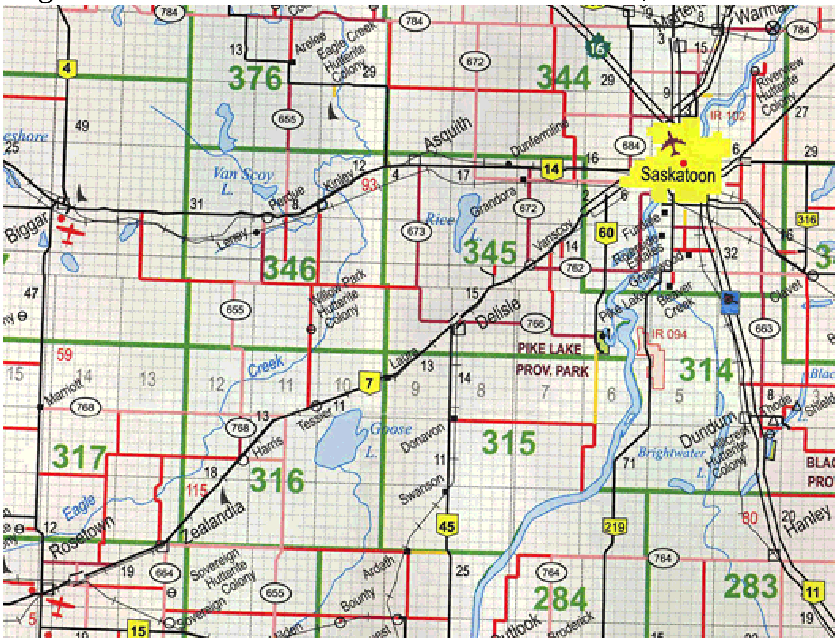
- BFS is used in web crawlers to build web page indexes. It starts at the source page and works its way through all of the links connected with it. Every web page is treated as a node in the network graph.

Breadth First Search example (BFS) – How GPS navigation works

There are variations between the route I typically travel and the one that GPS indicates is the shortest, most likely as a result of the algorithms employed. I discovered that GPS navigation and digital maps are examples of (BFS) Breadth First Search from my graph theory data structure classes. I tried searching the web for information about the potential usage of algorithms (such as the Breadth First Search example or the A* application) in GPS navigation, but I didn't discover many specifics. So this is an example of how Breadth First Search is applied to a practical application like GPS.

Let's first understandworking of GPS navigation

In contrast to people, digital maps view streets as a collection of nodes. Central Park West is the name of the 2.6-mile route that runs from Cathedral Pkwy (110 st) to Columbus Circle station (59 St). This route is regarded as a single entity by us (humans) (You may divide it into few more segments based on metro stations or intersections, but not more than that).



Graph Traversal in Maps

So how does an algorithm determine which route is the quickest to take to get somewhere? Algorithms for traversing graphs

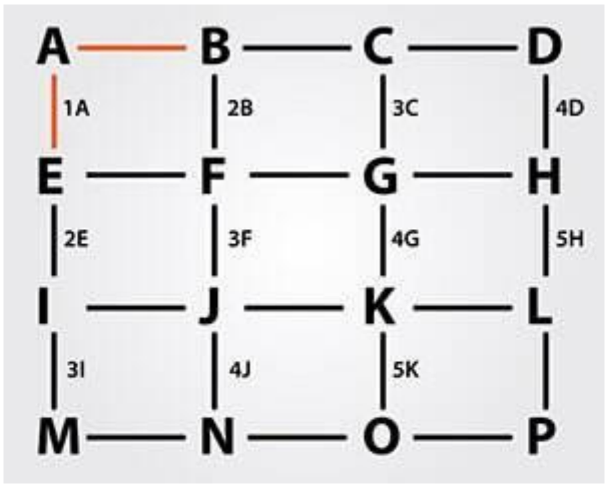
The Breadth First Search algorithm views the map in the same way that we do, but it is limited in how much it can take in. You draw a line connecting points A and B when you need to get from one place to another, and then you pick the route that is closest to that line. The same procedure is used repeatedly by algorithms to select the node closest to the intersection locations, ultimately determining the shortest path.

Take the above-mentioned straightforward GridWorld example, and let's try to solve it using a Breadth First Search search. Let's say you need to get from point A to point P.

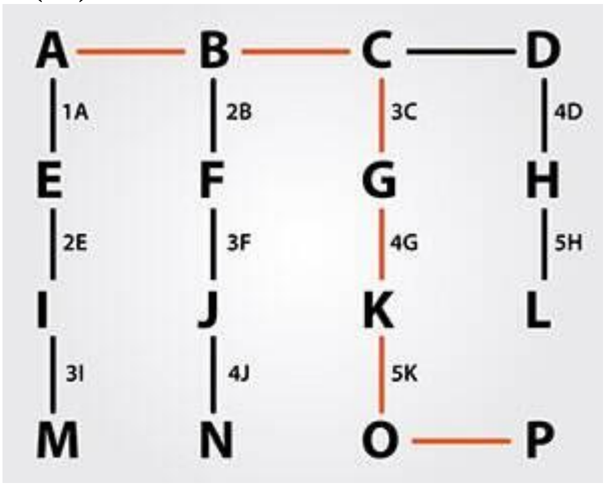
Every vertex in the image has a unique number assigned to it. Let’s understand it with the same GridWorld example, but in this case, the algorithm is not allowed to move or visit a node diagonally.

Breadth First Search for GridWorld

Step 1: Take node A into consideration as the source and go to all of A's neighbours, B(1A) and E. (1A). Mark A had come.

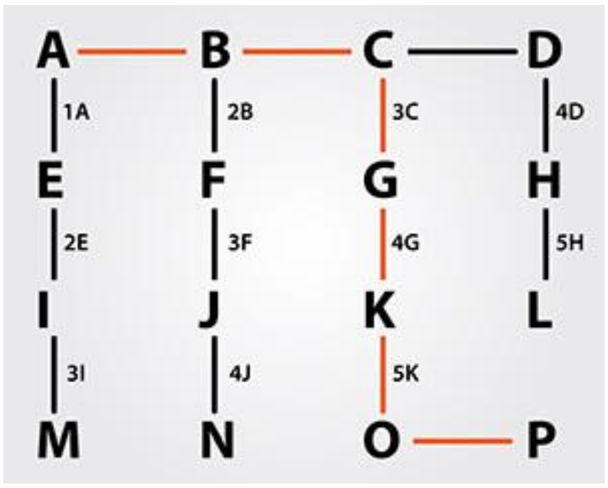


Step 2: Mark B as visited after visiting all of B's nearby nodes, including nodes C and F (2B).
Step 3: Visit nodes that are close to E because B has already been to F. Mark E as visited after visiting node I (2E).



Step 4: Carry out the previous steps for each node until each has been visited at least once. Identify the nodes you contemplated visiting.

Step 5: all disconnected or unneeded vertices are removed, and the graph is transformed into a minimal spanning tree with at least one connection between each node.
Highlight the nodes connecting source node A to node P, which has distance 6 and is the shortest path between two nodes.



Now that you know why they weren't equally far, it makes sense why GPS navigation didn't propose the routes A, E, I, M, N, O, or A, B, C, D, H, L, P.

After learning how GPS works, you'll wish the world were a straightforward Grid! But it isn't, much to the disappointment of a programmer. Therefore, while picking a route using a GPS, other factors such as elapsed time, the posted speed limit, real-time traffic updates, and the number of stop signs must also be taken into account. Because of this, your GPS might occasionally advise using curvy state roadways rather of the regular national ones.

2.2 . Uniform cost search

Uninformed search tactics include the uniform cost search (UCS). This search method is unique in that it always yields the best results when the function for calculating path length is defined. The advantage of the uniform cost search approach is that it may be applied to any generic or common search function. An algorithm for navigating a weighted tree or graph is known as uniform-cost search. When a separate cost is provided for each edge, this algorithm is used. Finding a path to the goal node with the lowest cumulative cost is the main objective of the uniform-cost search. According to their path costs, nodes are expanded using uniform-cost search from the root node. Any graph or tree can be solved with it if the optimal cost is required. The priority queue uses a uniform-cost search algorithm. The lowest cumulative cost is given the highest priority. If all edges have the same path cost, uniform cost search is comparable to BFS algorithm.

Advantages:

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

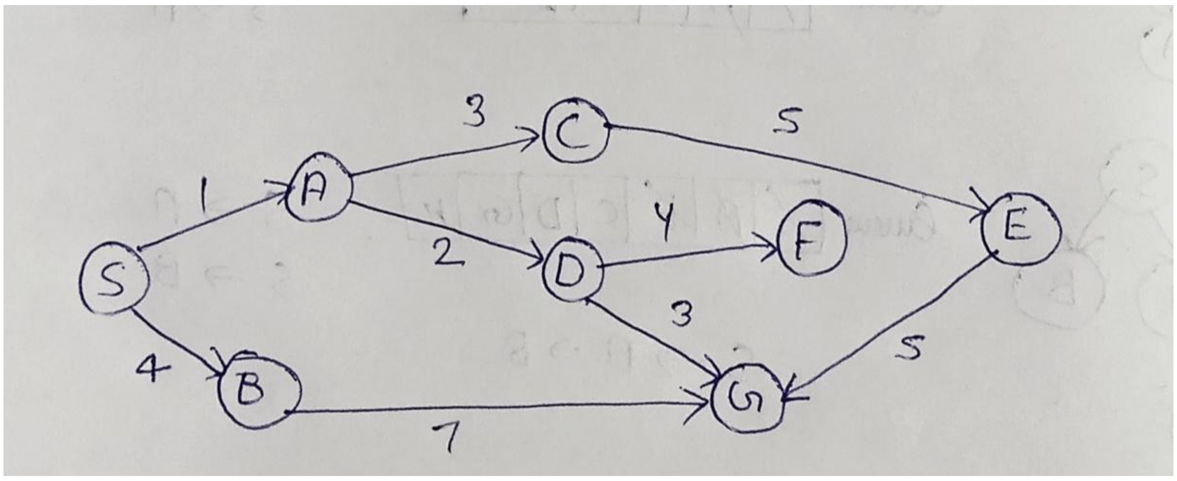
Disadvantages:

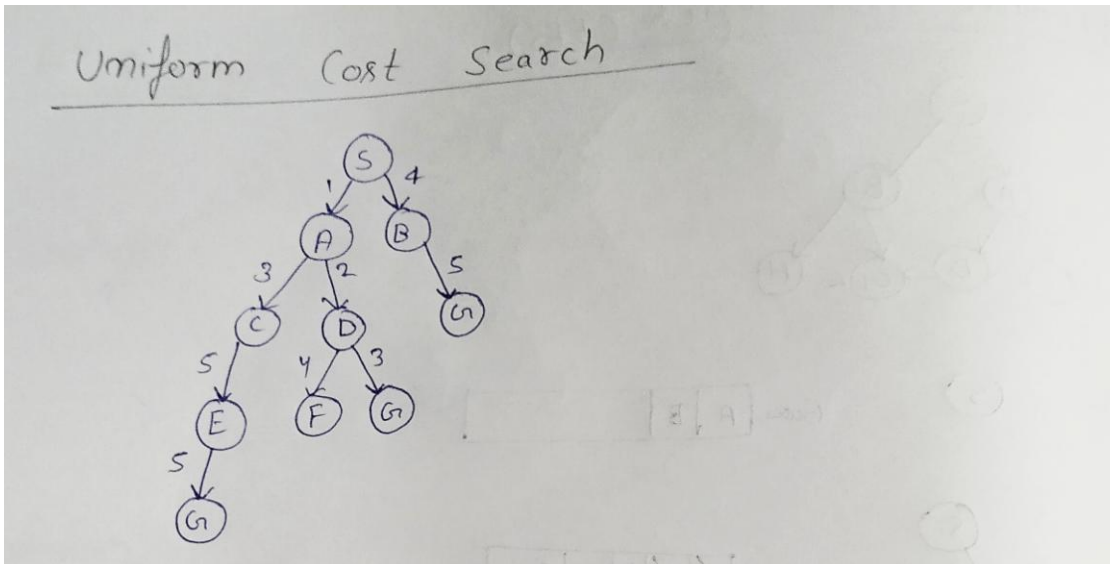
- o It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

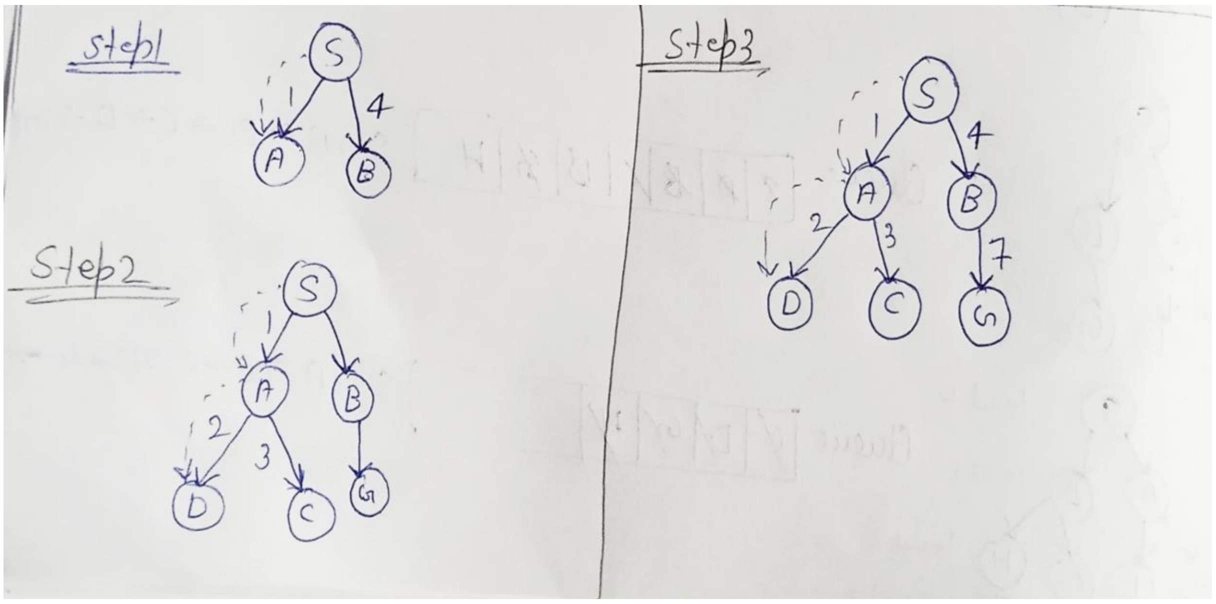
Completeness:
Uniform-cost search is complete, such as if there is a solution, UCS will find it.
Time Complexity:
Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
Hence, the worst-case time complexity of Uniform-cost search is**O(b**$^{1 + [C^*/\varepsilon]}$**)**/.

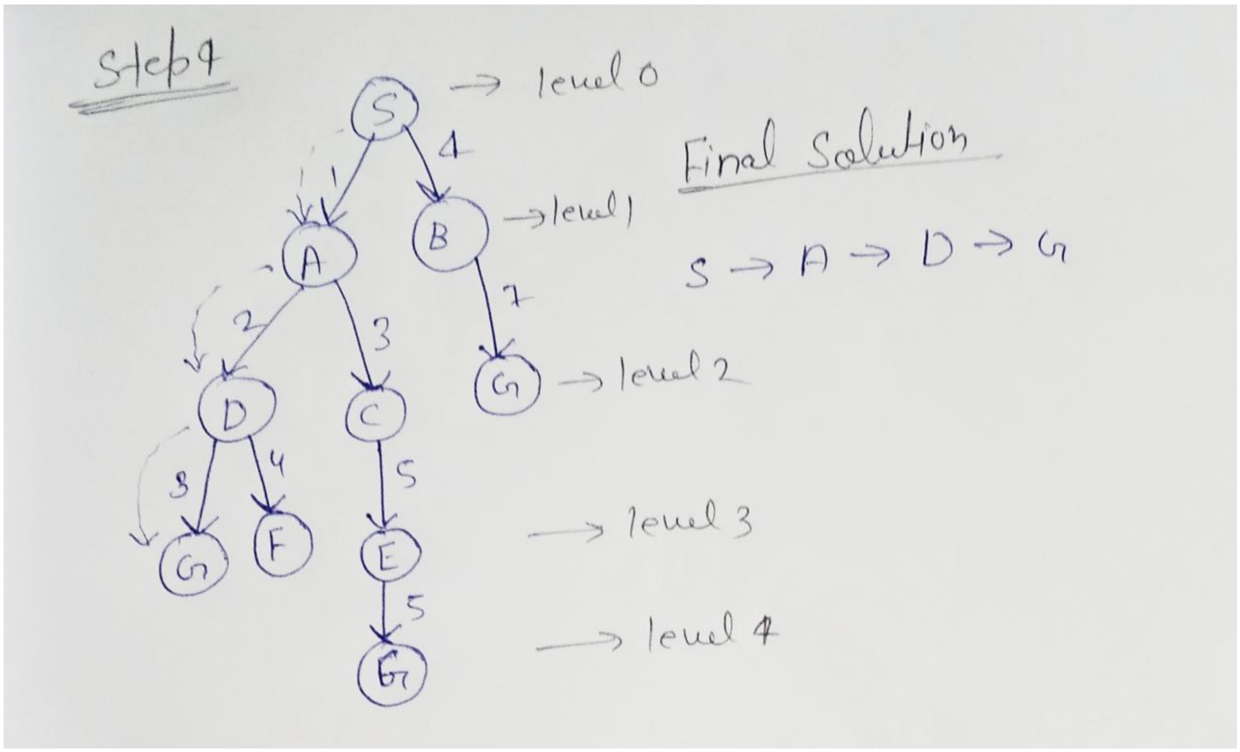
Space Complexity:
The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **O(b$^{1 + [C*/\varepsilon]}$)**.
Optimal:
Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Example









Final solution :- S →A→D →G