ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY
UNIVERSITY FOR INDUSTRY

AI Group Assignment

The nine Searching Algorithms pseudocode and flowchart

College: Electrical and Mechanical Engineering

Department: Software Engineering

Section: A

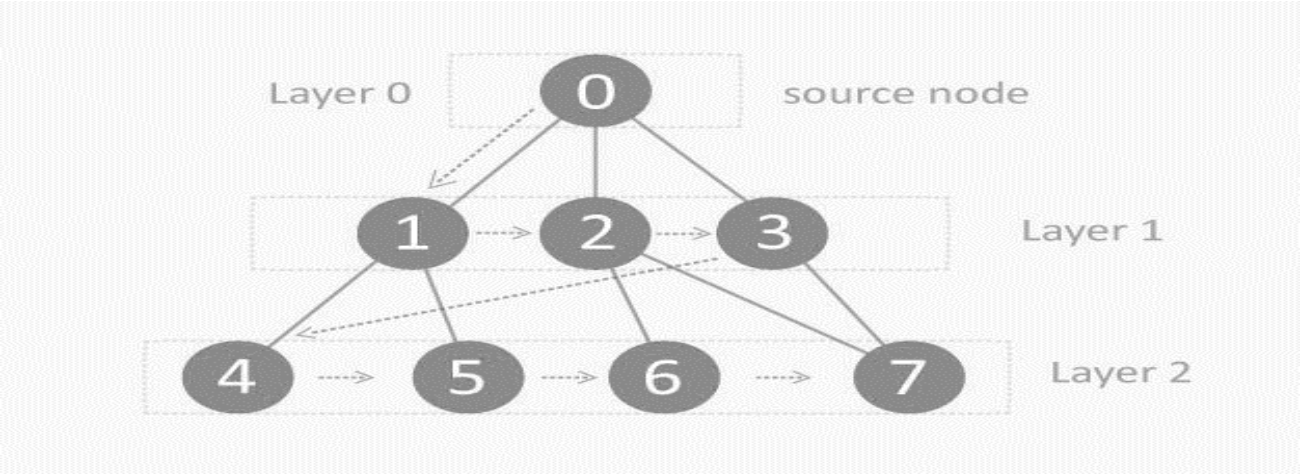
4th Year (1st Semester)

Group Members	Id No.
1. Adane Moges	Ets0079/13
2. Abdulmajid Awol	Ets0016/13
3. Abel Atkelet	Ets0020/13
4. Amanuel Mandefrow	Ets0122/13
5. Elias Balude	Ets0237/12

Contents

1. Breadth First Search (BFS)

BFS is a traversing algorithm where we should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). we must then move towards the next-level neighbor nodes.



steps involved in traversing a graph by using Breadth-First Search:

Step 1: Take an Empty Queue.

Step 2: Select a starting node (visiting a node) and insert it into the Queue.

Step 3: Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (exploring a node) into the Queue.

Step 4: Print the extracted node.

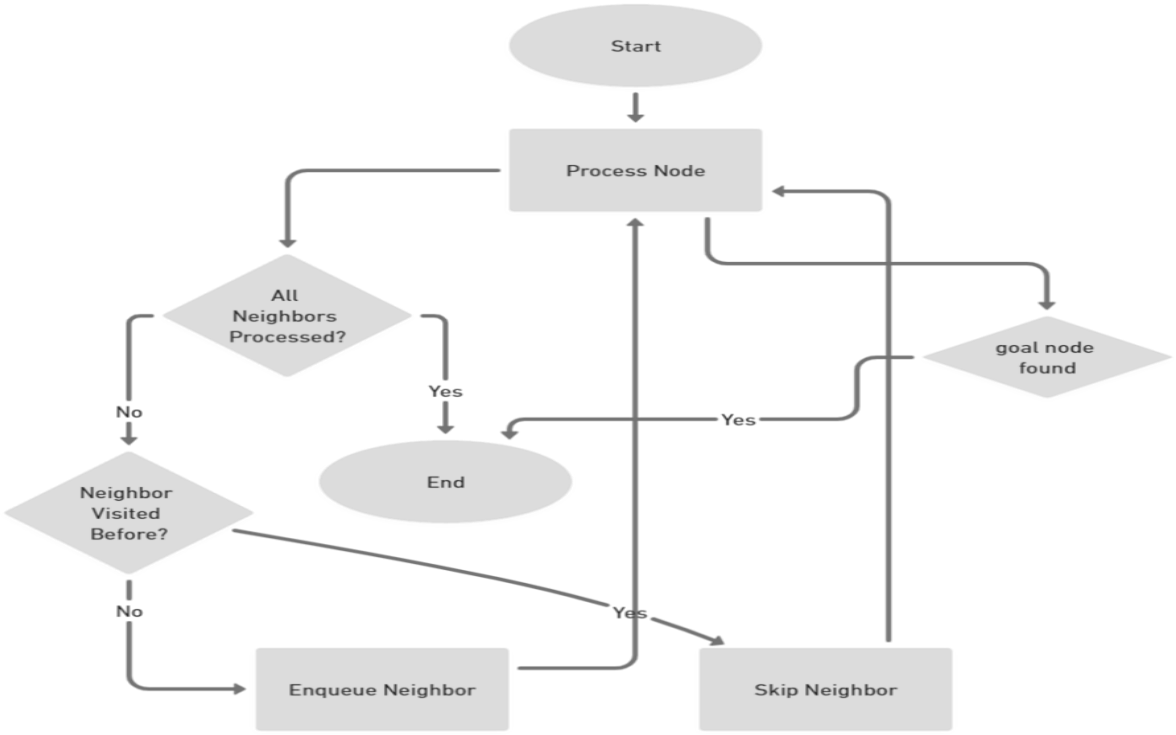
Breadth-First Search Algorithm Pseudocode

```
1.  BFS (G, s)          //Where G is the graph and s is the source node
2.      let Q be queue.
3.      Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.
4.
5.      mark s as visited.
6.      while ( Q is not empty)
7.          //Removing that vertex from queue,whose neighbour will be visited now
8.          v  =  Q.dequeue( )
9.
10.        //processing all the neighbours of v
11.        for all neighbours w of v in Graph G
12.            if w is not visited
13.                Q.enqueue( w )          //Stores w in Q to further visit its neighbour
14.                mark w as visited.
```

In the above code, the following steps are executed:

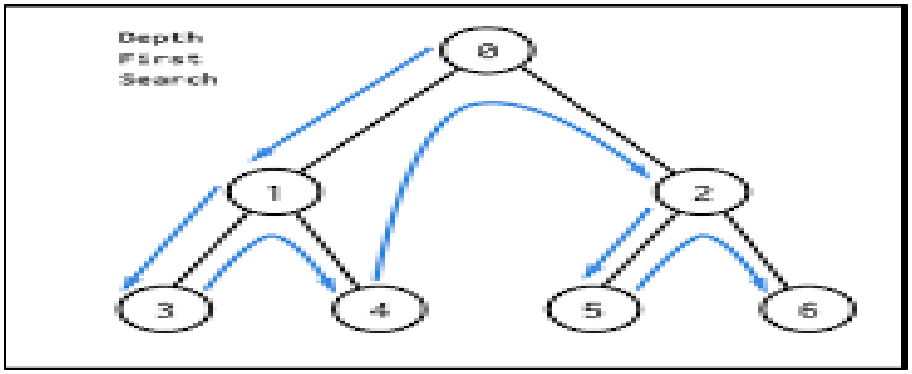
1. (G, s) is input, here G is the graph and s is the root node
2. A queue ‘Q’ is created and initialized with the source node ‘s’
3. All child nodes of ‘s’ are marked
4. Extract ‘s’ from queue and visit the child nodes
5. Process all the child nodes of v
6. Stores w (child nodes) in Q to further visit its child nodes
7. Continue till ‘Q’ is empty[1]

The flowchart



2. Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking



A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

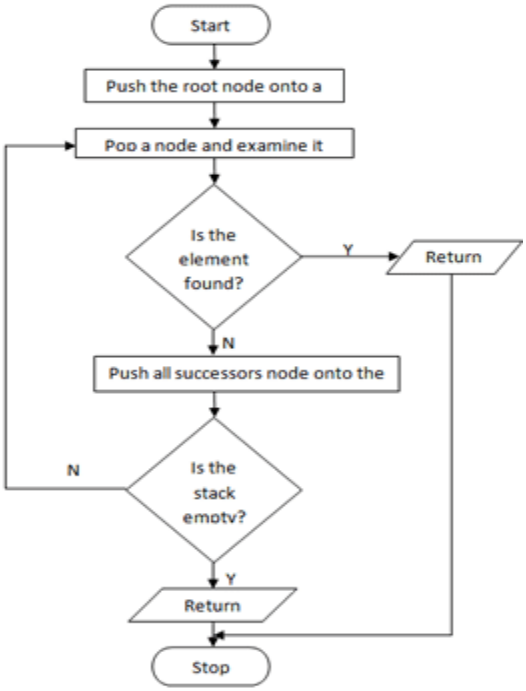
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.[2]

Depth-First Search Algorithm Pseudocode

```
1.      DFS-iterative (G, s):                          //Where G is graph and s is source vertex
2.        let S be stack
3.        S.push( s )          //Inserting s in stack
4.        mark s as visited.
5.        while ( S is not empty):
6.           //Pop a vertex from stack to visit next
7.           v  =  S.top( )
8.           S.pop( )
9.           //Push all the neighbours of v in stack that are not visited
10.          for all neighbours w of v in Graph G:
11.             if w is not visited :
12.                    S.push( w )
13.                   mark w as visited
14.
15.
16.     DFS-recursive(G, s):
17.        mark s as visited
18.        for all neighbours w of s in Graph G:
19.           if w is not visited:
20.              DFS-recursive(G, w)
```

The flow chart



3. Depth limited search

Depth limited search is an uninformed search algorithm which is similar to Depth First Search(DFS). It can be considered equivalent to DFS with a predetermined depth limit 'l'. Nodes at depth l are considered to be nodes without any successors.

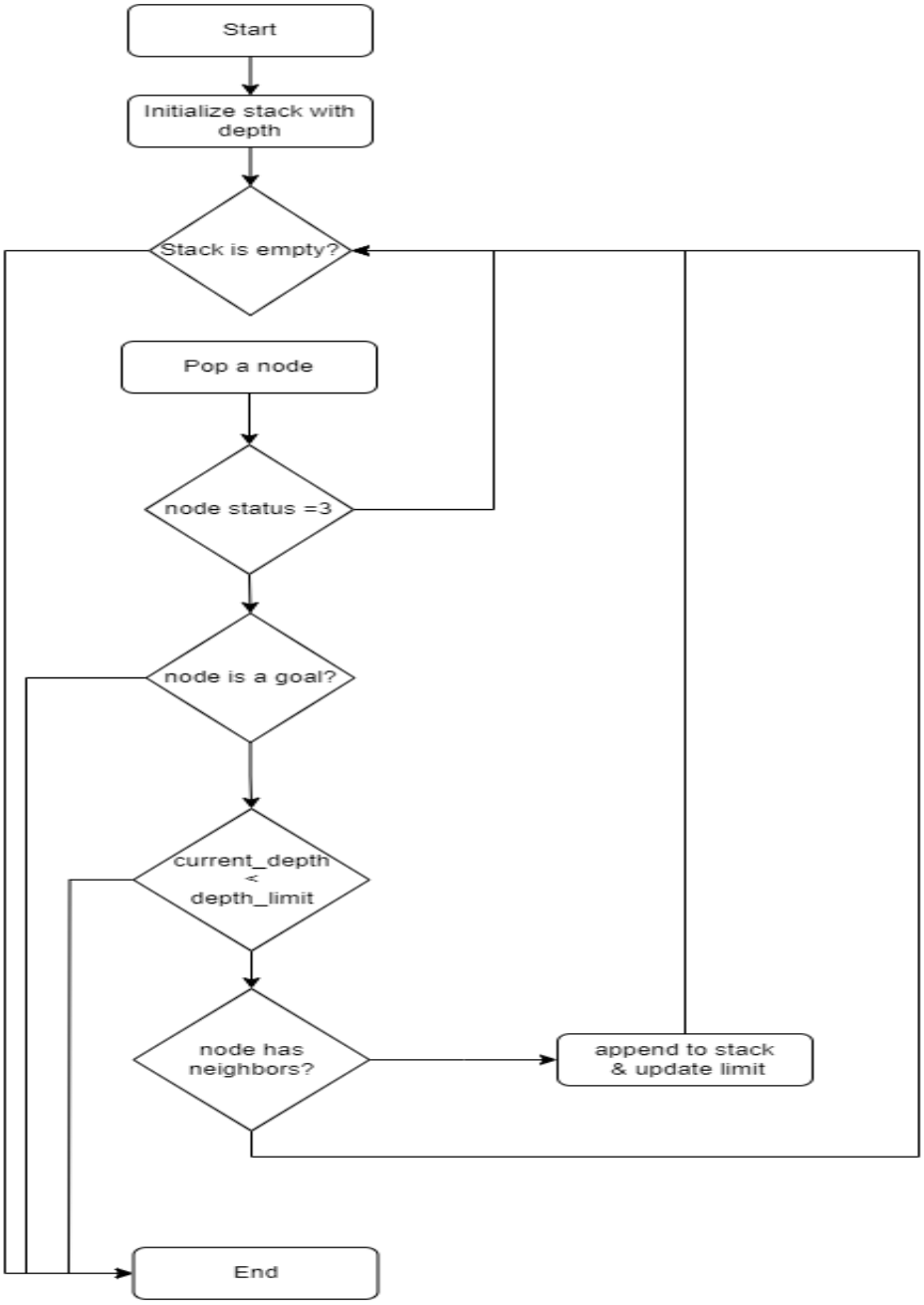
Depth limited search may be thought of as a solution to DFS's infinite path problem; in the Depth limited search algorithm, DFS is run for a finite depth 'l', where 'l' is the depth limit.

Pseudocode

1) Set STATUS=1(ready) for each of the given nodes in graph G.
2) Push the Source node or the Starting node onto the stack and set its STATUS=2(waiting).
3) Repeat steps 4 to 5 until the stack is empty or the goal node has been reached.
4) Pop the top node T of the stack and set its STATUS=3(visited).
5) Push all the neighbors of node T onto the stack in the ready state (STATUS=1) and with a depth less than or equal to depth limit 'l' and set their STATUS=2(waiting).
   (END OF LOOP)
6) END

```
1.  function DLS(node, depth)
2.      if depth = 0 or node is a goal
3.          return node
4.      else if depth > 0
5.          for each child of node
6.              result = DLS(child, depth-1)
7.              if result != cutoff
8.                  return result
9.      return cutoff
```

The Flowchart



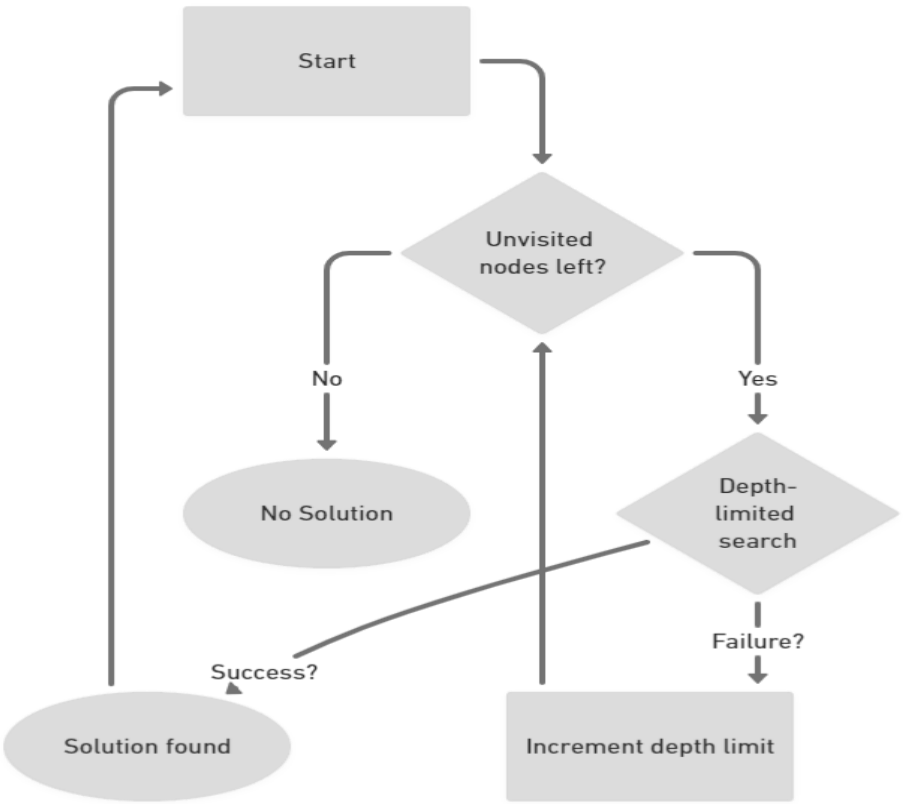
4. Iterative Deepening Depth First Search

IDS combines the benefits of DFS with Breadth First Search (BFS). The graph is explored using DFS, but the depth limit steadily increased until the target is located. In other words, IDS continually runs DFS, raising the depth limit each time, until the desired result is obtained. Iterative deepening is a method that makes sure the search is thorough (i.e., it discovers a solution if one exists) and efficient (i.e., it finds the shortest path to the goal).

The pseudocode for IDS

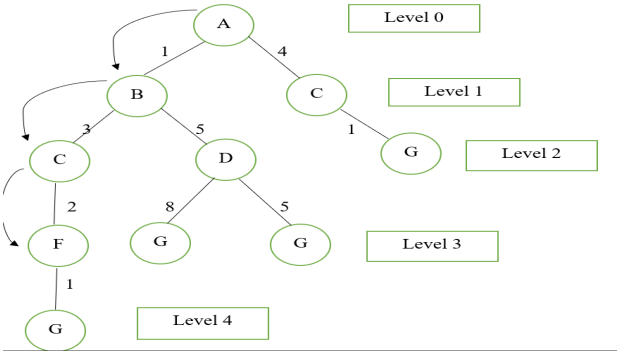
```
1. function iterativeDeepeningSearch(root, goal):
2.     depth = 0
3.     while True:
4.         result = depthLimitedSearch(root, goal, depth)
5.         if result == FOUND:
6.             return goal
7.         if result == NOT_FOUND:
8.             return None
9.         depthdepth = depth + 1
10.
11.  function depthLimitedSearch(node, goal, depth):
12.      if node == goal:
13.          return FOUND
14.      if depth == 0:
15.          return NOT_FOUND
16.      for child in node.children:
17.          result = depthLimitedSearch(child, goal, depth - 1)
18.          if result == FOUND:
19.              return FOUND
20.      return NOT_FOUND
```

The Flowchart



5. uniform cost search

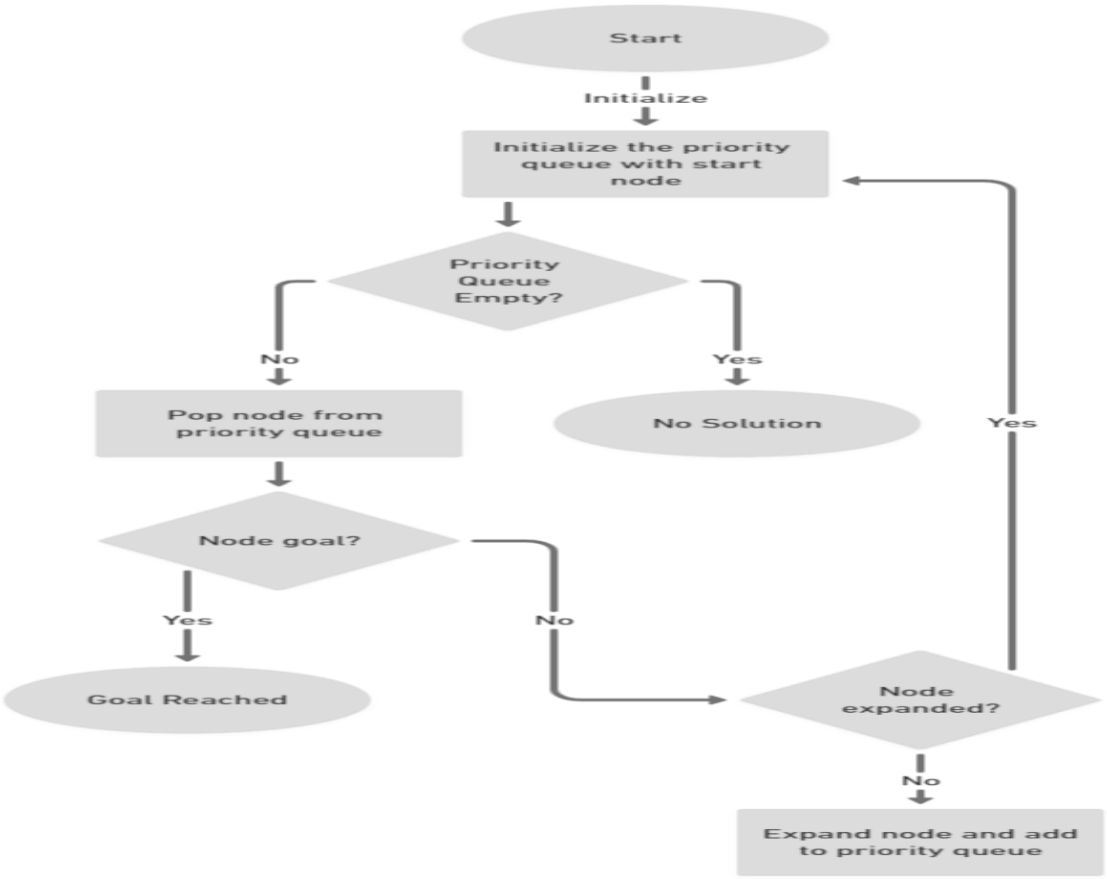
Uniform Cost Search is the best algorithm for a search problem, which does not involve the use of heuristics. It can solve any general graph for optimal cost. Uniform Cost Search as it sounds searches in branches that are more or less the same in cost. The algorithm's worst-case time and space complexity are both in O(b1+⌊C*/ ε⌋).



The pseudocode for UCS

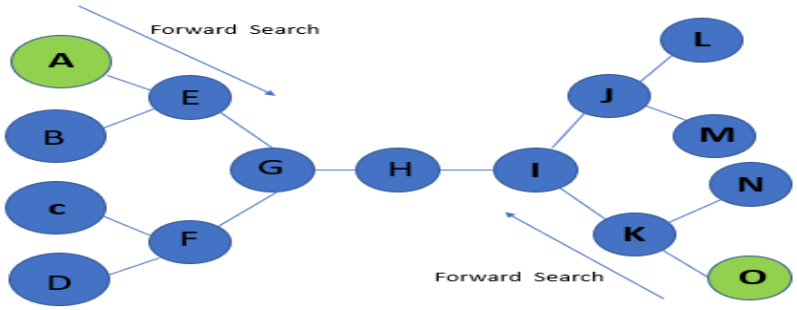
```
1. function UCS(Graph, start, target):
2.      Add the starting node to the opened list. The node has
3.      has zero distance value from itself
4.      while True:
5.        if opened is empty:
6.          break # No solution found
7.        selecte_node = remove from opened list, the node with
8.                     the minimun distance value
9.        if selected_node == target:
10.         calculate path
11.         return path
12.       add selected_node to closed list
13.       new_nodes = get the children of selected_node
14.       if the selected node has children:
15.         for each child in children:
16.           calculate the distance value of child
17.           if child not in closed and opened lists:
18.             child.parent = selected_node
19.             add the child to opened list
20.           else if child in opened list:
21.             if the distance value of child is lower than
22.              the corresponding node in opened list:
23.                child.parent = selected_node
24.                add the child to opened list
```

The Flowchart



6. Bidirectional search algorithm

Bidirectional Search is Graph Search Algorithm where two graph traversals (BFS) take place at the same time and is used to find the shortest distance between a fixed start vertex and end vertex. It is a faster approach, reduces the time required for traversing the graph.



Pseudocode[3]

```
1. startq = Queue for BFS from start node
2. endq = Queue for BFS from end node
3. parent= Array where startparent[i] is parent of node i
4. visited= Array where visited[i]=True if node i has been encountered
5. while startq is not empty and endq is not empty
```

```
6.      perform next iteration of BFS for startq (also save the parent of children in parent array)
7.      perform next iteration of BFS for endq
8.      if we have encountered the intersection node
9.          save the intersection node
10.         break
11. using intersection node, find the path using parent array
```

The Flowchart

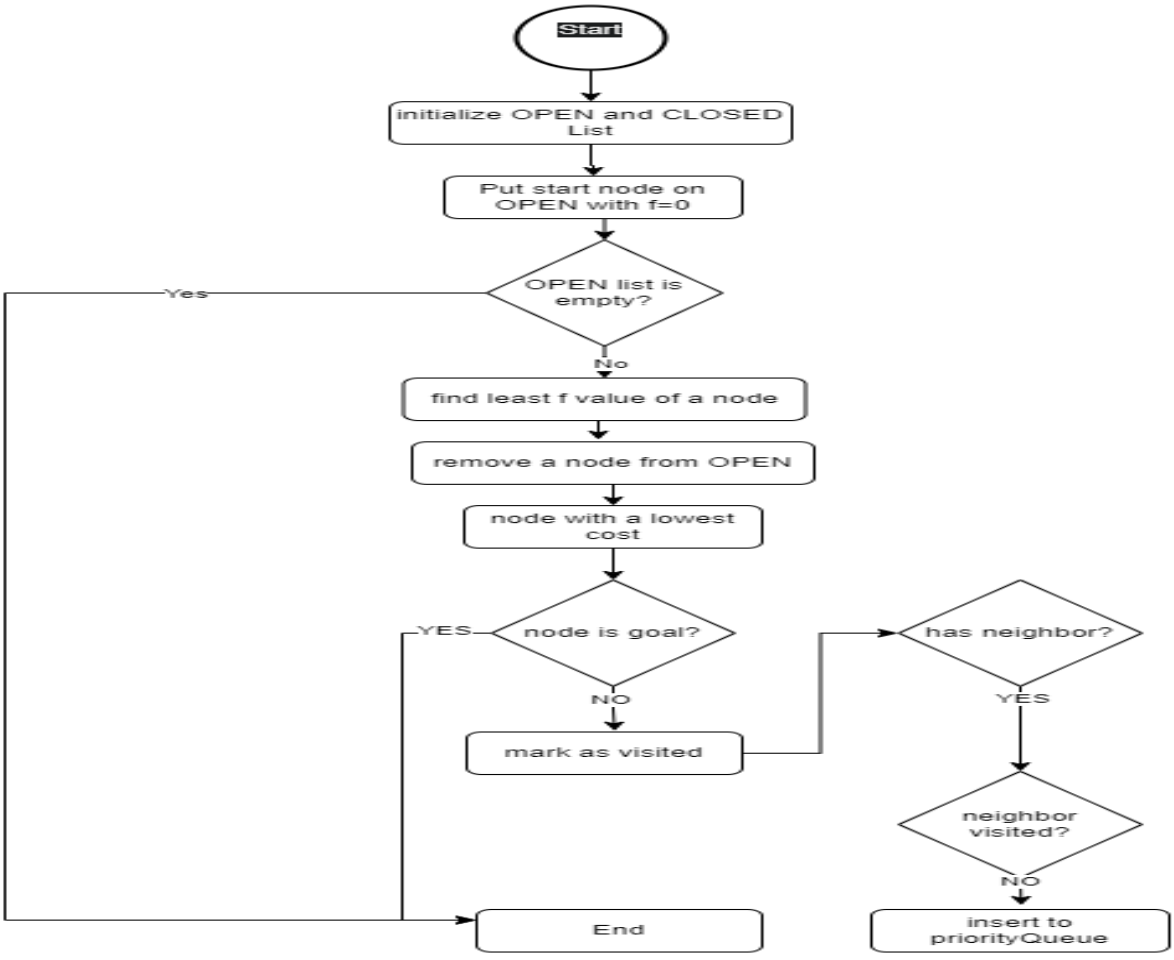
7. BFS(best first search)

The best first search uses the concept of a priority queue and heuristic search. It is a search algorithm that works on a specific rule. The aim is to reach the goal from the initial state via the shortest path. The best First Search algorithm in artificial intelligence is used for for finding the shortest path from a given starting node to a goal node in a graph. The algorithm works by expanding the nodes of the graph in order of increasing the distance from the starting node until the goal node is reached.

Pseudocode for Best First Search

```
1.  Best-First-Search(Graph g, Node start)
2.      1) Create an empty PriorityQueue
3.         PriorityQueue pq;
4.      2) Insert "start" in pq.
5.         pq.insert(start)
6.      3) Until PriorityQueue is empty
7.            u = PriorityQueue.DeleteMin
8.            If u is the goal
9.               Exit
10.           Else
11.              Foreach neighbor v of u
12.                 If v "Unvisited"
13.                     Mark v "Visited"
14.                     pq.insert(v)
15.              Mark u "Examined"
16. End procedure[4]
```

The flowchart



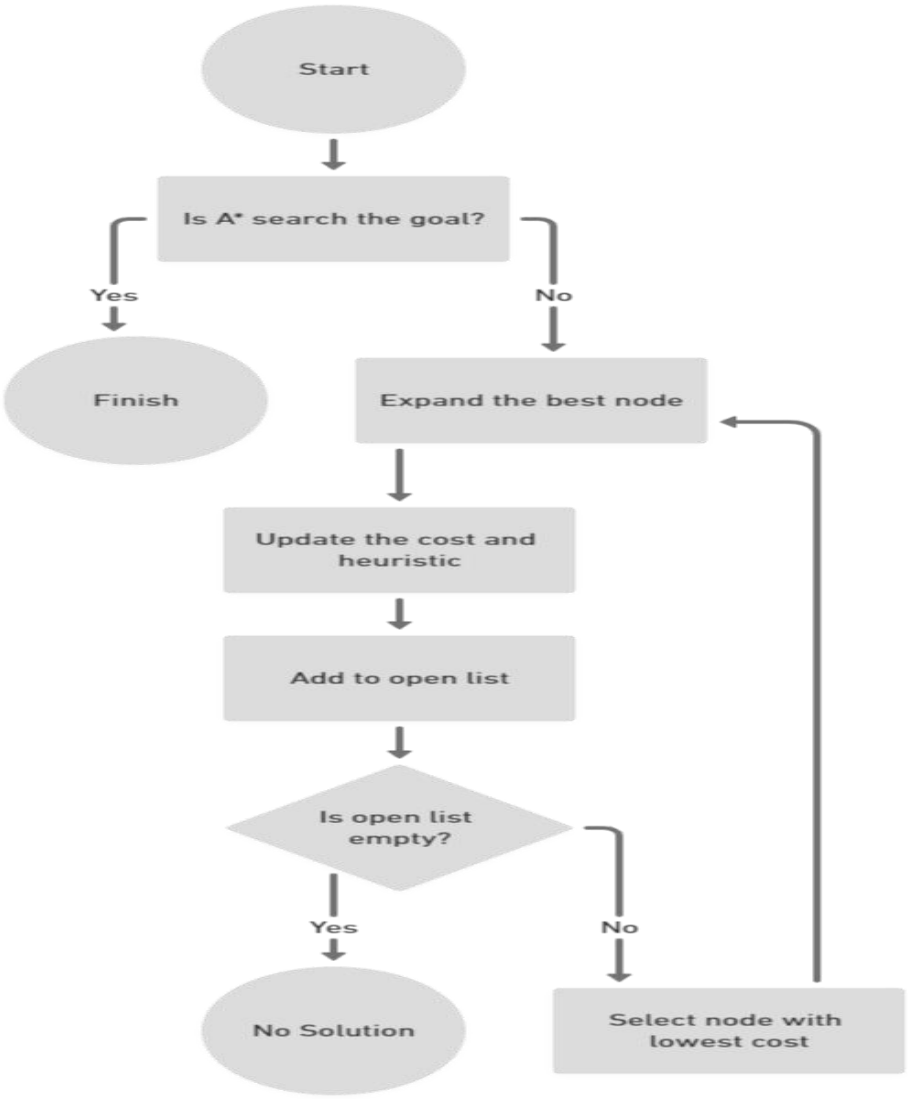
8. A* search

A* is a popular and widely used graph traversal and pathfinding algorithm. It is employed in various applications, including robotics, video games, and artificial intelligence. The primary purpose of A* is to find the shortest path from a starting point to a goal in a graph, considering the cost of traversing each edge.

Pseudocode for A*

```
1.  1.  Initialize the open list
2.  2.  Initialize the closed list
3.      put the starting node on the open
4.      list (you can leave its f at zero)
5.
6.  3.  while the open list is not empty
7.      a) find the node with the least f on
8.         the open list, call it "q"
9.
10.     b) pop q off the open list
11.
12.     c) generate q's 8 successors and set their
13.        parents to q
14.
15.     d) for each successor
16.        i) if successor is the goal, stop search
17.
18.        ii) else, compute both g and h for successor
19.          successor.g = q.g + distance between
20.                        successor and q
21.          successor.h = distance from goal to
22.          successor (This can be done using many
23.          ways, we will discuss three heuristics-
24.          Manhattan, Diagonal and Euclidean
25.          Heuristics)
26.
27.          successor.f = successor.g + successor.h
28.
29.        iii) if a node with the same position as
30.             successor is in the OPEN list which has a
31.           lower f than successor, skip this successor
32.
33.        iV) if a node with the same position as
34.             successor  is in the CLOSED list which has
35.             a lower f than successor, skip this successor
36.             otherwise, add  the node to the open list
37.     end (for loop)
38.
39.     e) push q on the closed list
40.     end (while loop)
```

The flowchart



9. AO* Algorithm

The **AO* algorithm** is based on AND-OR graphs to break complex problems into smaller ones and then solve them. The AND side of the graph represents those tasks that need to be done with each other to reach the goal, while the OR side stands alone for a single task.

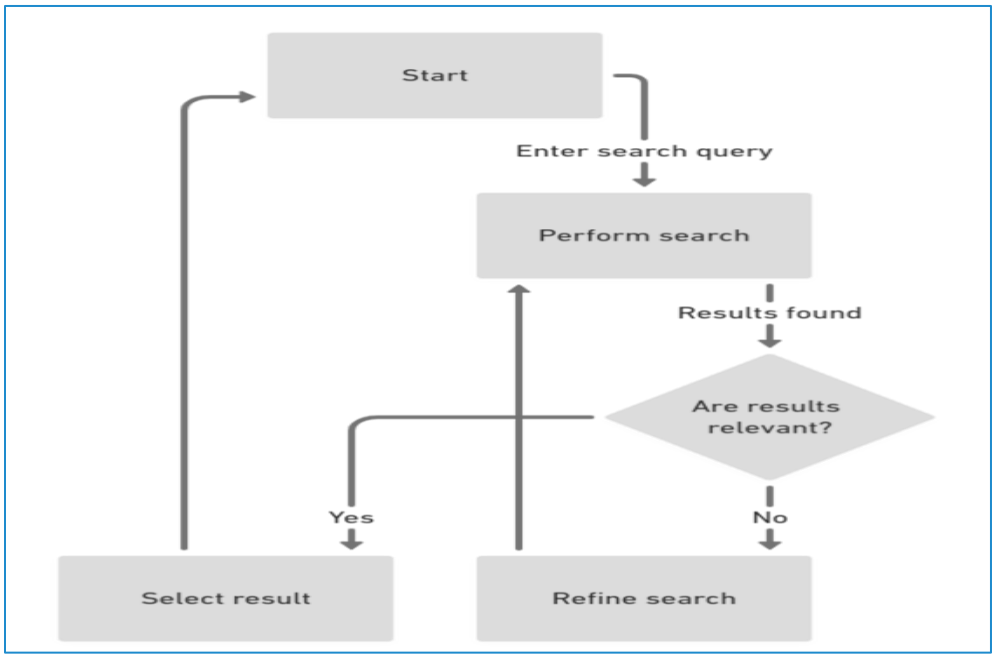
AO* works based on this formula: F(n) = G(n) + H(n), where G(n) is the actual cost of going from the starting node to the current node, H(n) is the estimated or heuristic cost of going from the current node to the goal node, and F(n) is the actual cost of going from the starting node to the goal node.[5]

Pseudocode for the AO*

```
1. Data: Graph, StartNode
```

```
2.  Result: The minimum cost path from StartNode to GoalNode
3.  CurrentNode← StartNode
4.  while There is a new path with lower cost from StartNode to the GoalNode do
5.      calculate the cost of path from the current node to the goal node through each of its successor nodes;
6.      if the successor node is connected to other successor nodes by AND-ARCS then
7.          sum up the cost of all paths in the AND-ARC; return the total cost;
8.      else
9.          calculate the cost of the single path in the OR side; return the single cost;
10.     end
11.     find the minimum cost path
12.     CurrentNode← Successor NodeOfMinimumCost Path
13.     if CurrentNode has no successor node then
14.         do the backpropagation and correct the estimated costs;
15.         Current Node StartNode
16.         return Current Node, New estimated costs;
17.     else
18.         return null;
19.     end
20. end
21. return The minimum cost path;
```

Flowchart AO*



Reference

[1] "Breadth First Search Tutorials & Notes | Algorithms | HackerEarth." Accessed: Dec. 04, 2023. [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

[2] "Depth First Search Tutorials & Notes | Algorithms | HackerEarth." Accessed: Dec. 04, 2023. [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

[3] "Bidirectional Search." Accessed: Dec. 04, 2023. [Online]. Available: https://iq.opengenus.org/bidirectional-search/

[4] "Best First Search (Informed Search) - GeeksforGeeks." Accessed: Dec. 04, 2023. [Online]. Available: https://www.geeksforgeeks.org/best-first-search-informed-search/

[5] "AO* algorithm - Artificial intelligence - GeeksforGeeks." Accessed: Dec. 21, 2023. [Online]. Available: https://www.geeksforgeeks.org/ao-algorithm-artificial-intelligence/