ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY

UNIVERSITY FOR INDUSTRY

College of Electrical and Mechanical Engineering

Department of Software Engineering

Introduction to Artificial Intelligence

Group Members Name	ID Number
Lulit Ayele	ETS0781/13
Meheret Alemu	ETS0813/13
Metasebia Fetene	ETS0849/13
Mihret Tekalgn	ETS0876/13
Milky Siraj	ETS0895/13

Submitted to: Mr. Tesfaye

Submission Date: 29, Dec 2023

Contents

Uninformed Searching Algorithms

Depth Limited Search Algorithm

Depth limited search algorithm is designed for uninformed search to solve the unbounded tree problem that happens to appear in the depth-first search algorithm and can be implemented by fixing a boundary or the depth of the search domain.

This limit can be called the depth limit, making the DFS search algorithm more refined and organized so that it does not move into an infinite loop. And this limit provides the solution to the infinite path problem that originated while using the DFS algorithm. Thus, Depth limited search can be called a revised version of the DFS algorithm.

Algorithm of depth limited search

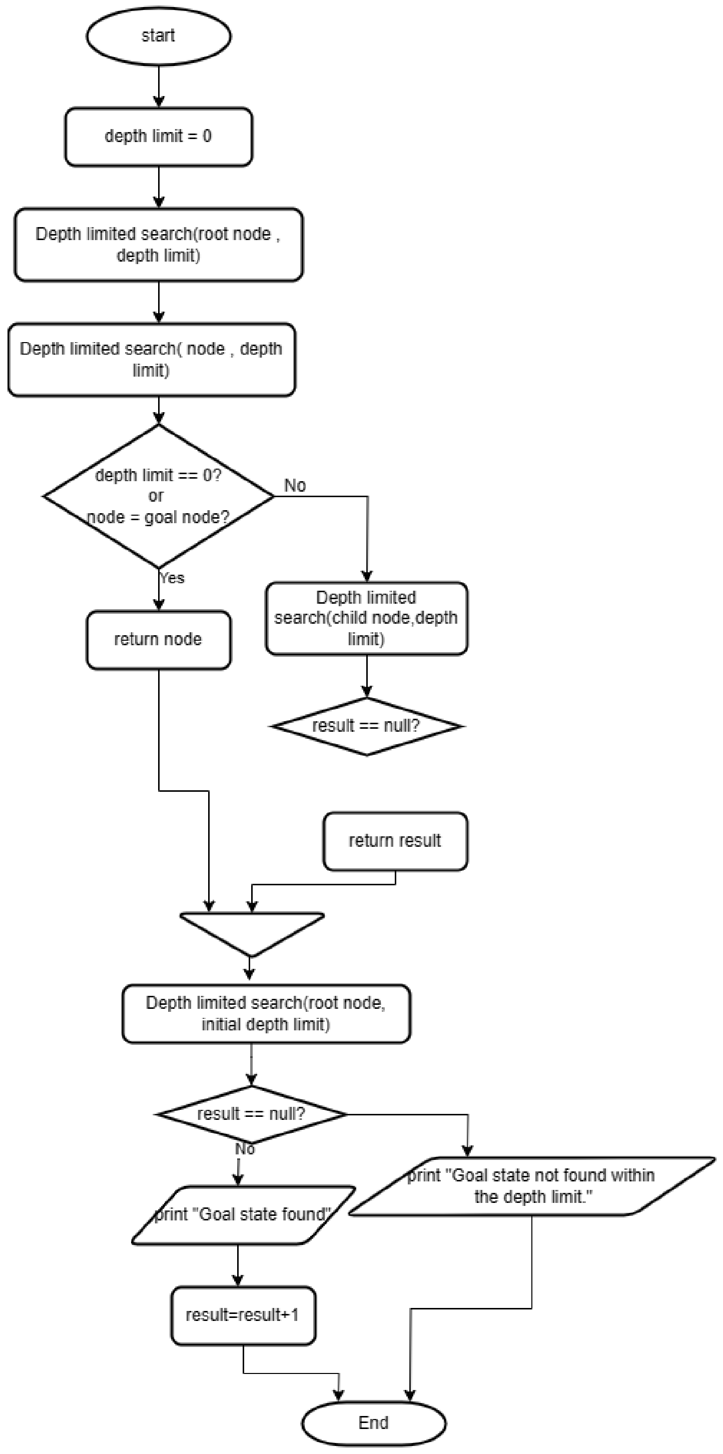
1. The starting node into the stack is PUSH.
2. If there is no node in the stack then stop
3. If the top node of the stack is the goal node, then stop and return the goal state as output.
4. Else we have to POP out the top node from the stack and search process is reinitiated. Find all its neighbors that are in ready state and PUSH them into the stack in any order.
5. Go to step 3.
6. Exit.

Pseudo code for Depth Limited Search Algorithm

```
Set depth_limit = 0
Function Call DepthLimitedSearch(root_node, depth_limit) return solution or failure
Procedure DepthLimitedSearch(node, depth_limit)
If depth_limit = 0 or node is a goal state, then
   Return node
End If
If node is not a goal state and depth_limit > 0, then
   For each child_node of node, do
       Call DepthLimitedSearch(child_node, problem, limit - 1)
      If result is not null, then
         Return result
      End If
   End For
End If
Return null
End Procedure
Call DepthLimitedSearch(root_node, initial_depth_limit)
If result is not null, then    Print "Goal state found: " + result
Else
```

```
    Print "Goal state not found within the depth limit."
End If
Stop
```

Flowchart for Depth Limited Search Algorithm


start
depth limit = 0
Depth limited search(root node , depth limit)
Depth limited search(node , depth limit)
depth limit == 0? or node = goal node?
No
Yes
return node
Depth limited search(child node,depth limit)
result == null?
return result
Depth limited search(root node, initial depth limit)
result == null?
No
print "Goal state found"
print "Goal state not found within the depth limit."
result=result+1
End

Implementation

```
def depth_limited_search(problem, limit=999999):
    """Search the deepest nodes in the search tree first, up to limit depth."""
    node = Node(problem.get_start_state())
    if problem.is_goal_state(node.state):
        return node.solution
    frontier = util.Stack()
    frontier.push(node)
    explored = set([node.state])
    while not frontier.is_empty():
        node = frontier.pop()
        explored.add(node.state)
        for child in node.create_children(problem):
            if child.path_cost <= limit:
                if child.state not in explored:
                    if child not in frontier.list:
                        if problem.is_goal_state(child.state):
                            return child.solution
                        frontier.push(child)
    return []

def iterate_dls(problem, limit=999999, inc=1):
    """Iterate DLS starting from limit and rising in inc increments."""
    result = []
    while result == []:
        result = depth_limited_search(problem, limit)
        limit += inc
    return result
```

Depth-First Search Algorithm

Pseudo code for Depth-First Search (DFS) Algorithm

```
DFS(graph, startNode):
    visited = set()         // Keeps track of visited nodes
    dfsHelper(graph, startNode, visited)
dfsHelper(graph, currentNode, visited):
    visited.add(currentNode)      // Mark current node as visited
    process(currentNode)          // Process or perform operations on the current node
    neighbors = graph.getNeighbors(currentNode)  // Get neighbors of the current node
    for neighbor in neighbors:
        if neighbor not in visited:      // Check if neighbor is not visited
```

```
                dfsHelper(graph, neighbor, visited)  // Recursively call dfsHelper on the neighbor
```

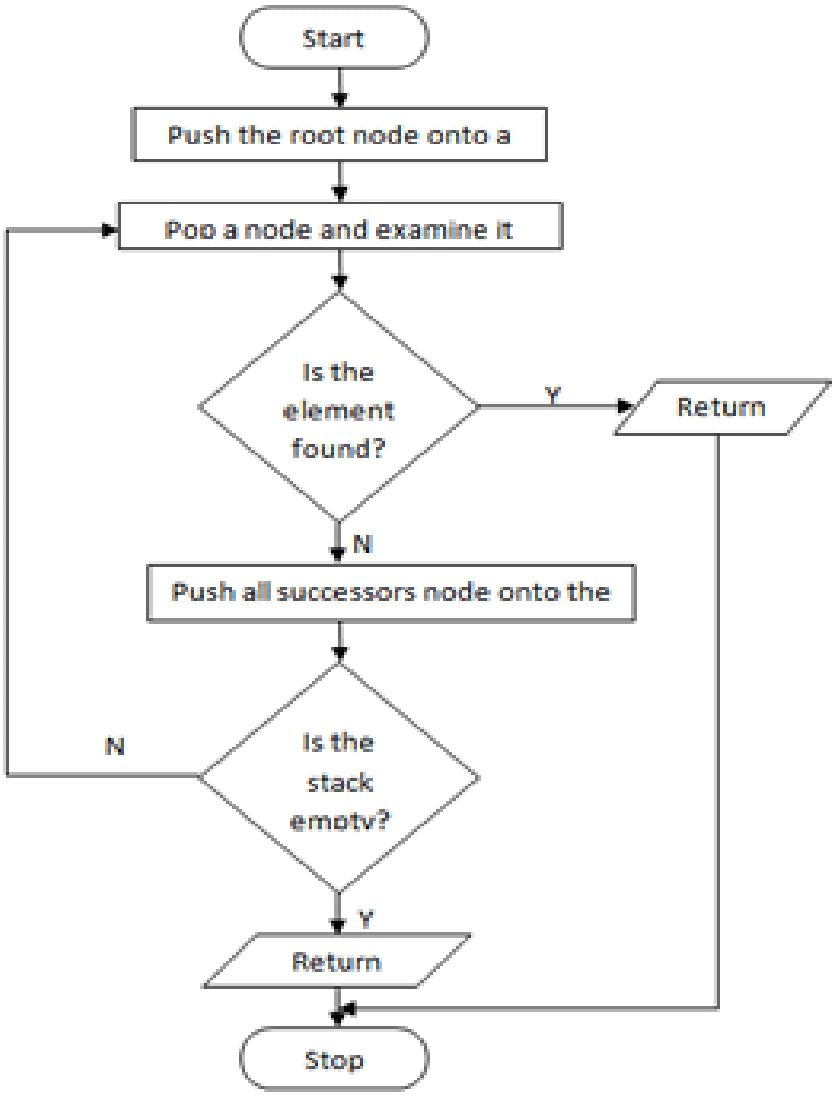
In the pseudo code, graph represents the graph data structure containing nodes and edges. startNode is the starting node from which the DFS traversal will begin. visited is a set that keeps track of visited nodes to avoid revisiting them.

The algorithm starts by initializing the visited set. It then calls the dfsHelper function, passing in the graph, starting node, and the visited set. The dfsHelper function is a recursive function that performs the DFS traversal.

In the dfsHelper function, the current node is marked as visited and processed (performing any desired operations). The algorithm retrieves the neighboring nodes of the current node. For each unvisited neighbor, the dfsHelper function is recursively called on that neighbor. This recursion continues until all reachable nodes from the current node are visited.

Note that process(currentNode) represents the specific operations you want to perform on each node during the DFS traversal. You can customize this part of the pseudo code to suit your specific needs, such as collecting data, updating properties, or performing other operations. Remember to implement the getNeighbors function specific to your programming language or environment to retrieve the neighbors of a node.

Flowchart for Depth-First Search (DFS) algorithm


Start
Push the root node onto a
Pop a node and examine it
Is the element found?
Y
Return
N
Push all successors node onto the
Is the stack empty?
N
Y
Return
Stop

Implementation

```
def dfs(graph, start_node):
    visited = set()     # Set to track visited nodes
    stack = [start_node]     # Stack to store nodes to visit

    while stack:
        node = stack.pop()     # Get the next node from the stack

        if node not in visited:
            visited.add(node)
            print(node)     # Process the node (e.g., print or perform custom operations)

            # Add unvisited neighbors of the current node to the stack
        neighbors = graph[node]
        for neighbor in neighbors:
            if neighbor not in visited:
                stack.append(neighbor)

# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': ['G'],
    'E': ['G'],
    'F': ['G'],
    'G': []
}

start_node = 'A'
dfs(graph, start_node)

Output: A C F G B E D
```

Breadth-First Search Algorithm

The **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level i.e Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited. To do this a queue is used and all the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

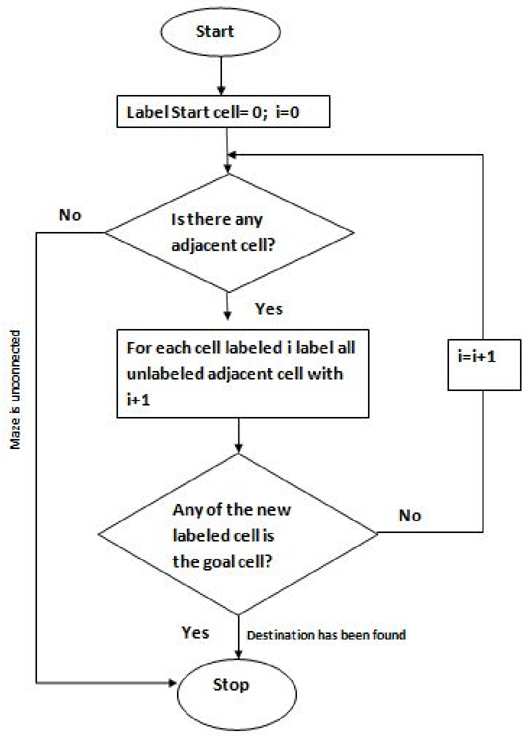
Pseudo code for Breadth first Search Algorithm

```
1. create an empty queue Initialize a queue data structure and an empty set to keep track
   of visited nodes.
2. Enqueue the start node into the queue and mark it as visited.
3. Repeat the following steps while the queue is not empty:
   a. Dequeue a node from the front of the queue and assign it to a variable (let's call it
   current node).
   b. Process or examine the current node.
   c. Enqueue all unvisited neighbor nodes of the current node into the queue and mark
   them as visited.
4. If the target node is found during the search, you can stop and return the result.
```

5. If the queue becomes empty without finding the target node, it means the target node is not reachable from the start node.

```
function BFS WallFollower(maze, start node, target node):
  create an empty set to keep track of visited nodes
  enqueue the start node into the queue
  add the start node to the visited set
  while the queue is not empty:
    current node = dequeue from the queue
    if current node is the target node:
      return "Target node found"
    for each neighbor node of current node:
      if neighbor node is not in the visited set and neighbor node is accessible:
        enqueue neighbor node into the queue
        add neighbor node to the visited set
  return "Target node not found"
```

Flowchart for Breadth-First Search


Start
Label Start cell= 0; i=0
Is there any adjacent cell?
No
Yes
Maze is unconnected
For each cell labeled i label all unlabeled adjacent cell with i+1
i=i+1
Any of the new labeled cell is the goal cell?
No
Yes
Destination has been found
Stop

Implementation

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([(start, [start])])
    visited.add(start)
    shortest_paths = {start: [start]}

    while queue:
        node, path = queue.popleft()

        print(f"Visiting node: {node}")
        print(f"Shortest path to {node}: {path}")

        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
                visited.add(neighbor)
                shortest_paths[neighbor] = path + [neighbor]

    return shortest_paths

# Example
 graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

start_node = 'A'

shortest_paths = bfs(graph, start_node)

print("\nShortest paths:")
for node, path in shortest_paths.items():
    print(f"{node}: {path}")
```

Output:

```
Visiting node: A
Shortest path to A: ['A']
Visiting node: B
Shortest path to B: ['A', 'B']
```

```
Visiting node: C
Shortest path to C: ['A', 'C']
Visiting node: D
Shortest path to D: ['A', 'B', 'D']
Visiting node: E
Shortest path to E: ['A', 'B', 'E']
Visiting node: F
Shortest path to F: ['A', 'C', 'F']

Shortest paths:
A: ['A']
B: ['A', 'B']
C: ['A', 'C']
D: ['A', 'B', 'D']
E: ['A', 'B', 'E']
F: ['A', 'C', 'F']
```

Bidirectional Search Algorithm

Pseudo code of Bidirectional Search

The bidirectional search algorithm is a graph search algorithm that efficiently finds the shortest path between two nodes in a graph. It operates by simultaneously searching forward from the start node and backward from the goal node until the two searches meet. This approach significantly reduces the search space compared to traditional single-direction search algorithms like breadth-first search or depth-first search.

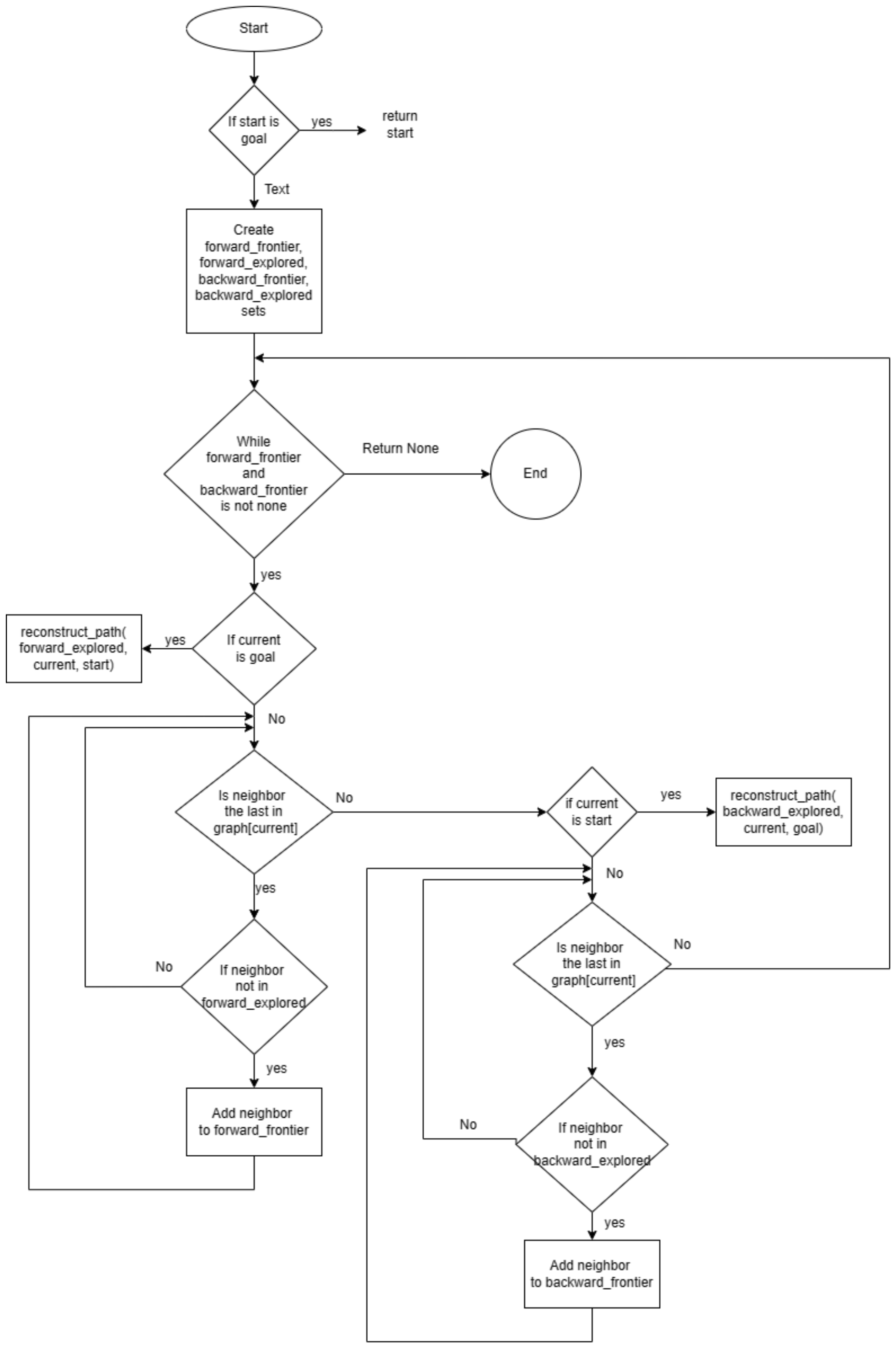
The pseudo code outlines the algorithm's steps:

1. Initialize forward and backward frontiers with the start and goal nodes, respectively.
2. Maintain explored sets for both directions to avoid revisiting nodes.
3. While both frontiers are not empty, iterate: a. Pop a node from the forward frontier, mark it as explored, and add its unvisited neighbors to the forward frontier. b. Check if the current forward node is the goal; if so, reconstruct the path using reconstruct_path. c. Pop a node from the backward frontier, mark it as explored, and add its unvisited neighbors to the backward frontier. d. Check if the current backward node is the start; if so, reconstruct the path using reconstruct_path.
4. If the search stops without finding a path, return None.
5. The reconstruct_path function traces the path from the current node to the start or goal node by traversing through the explored set and backtracking to the respective origin.

The bidirectional search algorithm effectively combines the advantages of both forward and backward search, resulting in a more efficient and time-saving approach for finding the shortest path between two nodes in a graph.

```
function bidirectional_search(graph, start, goal):
    if start equals goal:
        return [start]

    forward_frontier = [start]
    forward_explored = set()
    backward_frontier = [goal]
    backward_explored = set()

    while forward_frontier is not empty and backward_frontier is not empty:
        current = remove the first node from forward_frontier
        add current to forward_explored

        if current equals goal:
            return call reconstruct_path with forward_explored, current, and start

        for each neighbor in graph[current]:
            if neighbor is not in forward_explored:
                add neighbor to forward_frontier

        current = remove the first node from backward_frontier
        add current to backward_explored

        if current equals start:
            return call reconstruct_path with backward_explored, current, and goal

        for each neighbor in graph[current]:
            if neighbor is not in backward_explored:
                add neighbor to backward_frontier

    return None

function reconstruct_path(explored, current, start):
    path = []
    while current is not equal to start:
        for each parent in explored:
            if current is in graph[parent]:
                add current to path
                current = parent
                break

    reverse the path
    return path
```

Flowchart of Bidirectional Search


Start
If start is goal
yes
return start
Text
Create forward_frontier, forward_explored, backward_frontier, backward_explored sets
While forward_frontier and backward_frontier is not none
Return None
End
yes
If current is goal
yes
reconstruct_path(forward_explored, current, start)
No
Is neighbor the last in graph[current]
No
if current is start
yes
reconstruct_path(backward_explored, current, goal)
yes
If neighbor not in forward_explored
No
yes
Add neighbor to forward_frontier
No
Is neighbor the last in graph[current]
No
yes
If neighbor not in backward_explored
No
yes
Add neighbor to backward_frontier

Implementation

```
from collections import deque

def bidirectional_search(graph, start, target):
    # Check if the start and target nodes are the same
    if start == target:
        return [start]

    # Initialize the forward and backward queues
    forward_queue = deque([(start, [start])])
    backward_queue = deque([(target, [target])])

    # Initialize the forward and backward visited sets
    forward_visited = set([start])
    backward_visited = set([target])

    while forward_queue and backward_queue:
        # Perform the forward search
        node, path = forward_queue.popleft()

        for neighbor in graph[node]:
            if neighbor not in forward_visited:
                forward_visited.add(neighbor)
                forward_queue.append((neighbor, path + [neighbor]))

                if neighbor in backward_visited:
                    # Concatenate the forward and backward paths
                    forward_path = path + [neighbor]
                    backward_path = backward_queue[0][1][::-1]
                    return forward_path + backward_path

        # Perform the backward search
        node, path = backward_queue.popleft()

        for neighbor in graph[node]:
            if neighbor not in backward_visited:
                backward_visited.add(neighbor)
                backward_queue.append((neighbor, [neighbor] + path))

                if neighbor in forward_visited:
                    # Concatenate the forward and backward paths
                    forward_path = forward_queue[0][1]
                    backward_path = [neighbor] + path[::-1]
                    return forward_path + backward_path

    # No path exists between the start and target nodes
    return None


# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E', 'G'],
    'G': ['F']
}

start_node = 'A'
target_node = 'G'

path = bidirectional_search(graph, start_node, target_node)
if path:
    # Remove duplicate occurrences of nodes in the path
    path = list(dict.fromkeys(path))
    print(f"Shortest path from {start_node} to {target_node}: {' -> '.join(path)}")
else:
    print(f"No path exists between {start_node} and {target_node}.")
```

Output: Shortest path from A to G: A -> C -> F -> G

Uniform Cost Search Algorithm

Pseudo code for Uniform Cost Search Algorithm

The goal of UCS is to find the path with the lowest cost from a start node to a goal node. It explores the graph by gradually expanding nodes in a breadth-first manner, considering the cost associated with each node and the cumulative cost of reaching that node from the start node.

Here's how the Uniform Cost Search algorithm works:

1. Initialize an empty priority queue, typically implemented as a min-heap, to store the nodes to be expanded.
2. Initialize a dictionary or a data structure to store the cumulative cost of reaching each node from the start node.
3. Enqueue the start node with a cost of 0 into the priority queue.
4. While the priority queue is not empty, do the following:
    a. Dequeue the node with the lowest cost from the priority queue.
    b. If the dequeued node is the goal node, terminate the search and return the optimal path.
    c. Otherwise, expand the dequeued node by generating its neighboring nodes.
    d. For each neighbor, calculate the cumulative cost of reaching it from the start node by adding the cost of the current node and the edge cost between the current node and the neighbor.

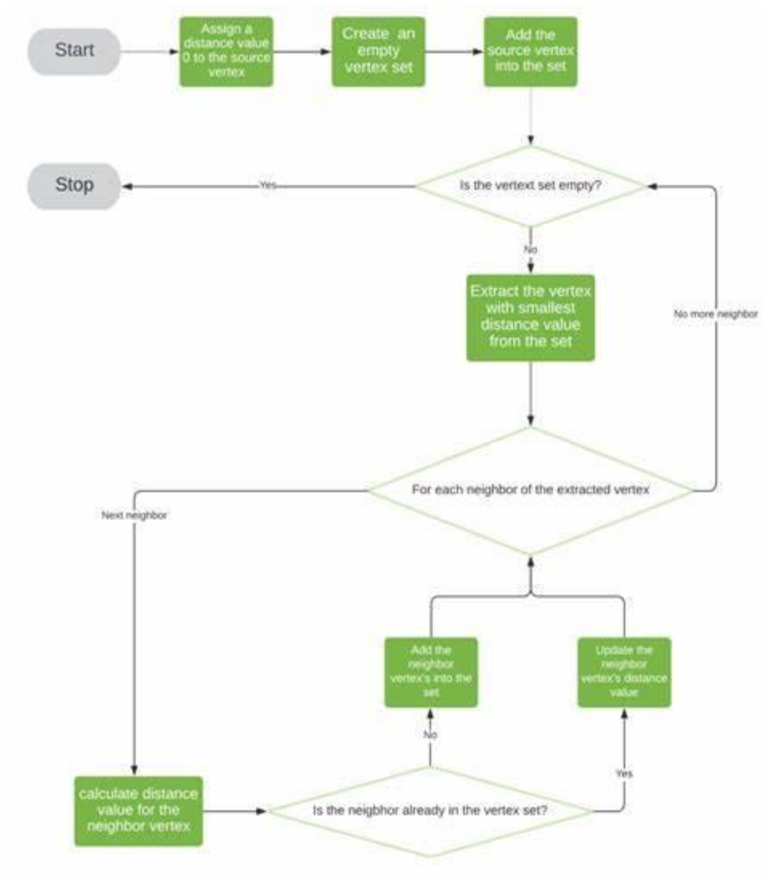
    e. If the neighbor has not been visited before or the new cost is lower than the previously recorded cost, update the cost and enqueue the neighbor into the priority queue.
5. If the priority queue becomes empty and the goal node has not been reached, there is no path from the start node to the goal node.

UCS guarantees to find the optimal solution as long as the edge costs are non-negative. However, if there are infinite paths with non-zero costs, UCS may not terminate. Additionally, UCS can be memory-intensive since it needs to keep track of the cumulative cost for each node.

```
function uniformCostSearch(graph, start, goal):
   // Initialize an empty priority queue
   frontier = PriorityQueue()
   frontier.enqueue(start, 0)  // Enqueue the start node with a cost of 0

   // Initialize a dictionary to store the cumulative cost of reaching each node
   costSoFar = {}
   costSoFar[start] = 0

   while not frontier.isEmpty():
      currentNode = frontier.dequeue()  // Dequeue the node with the lowest cost

      if currentNode == goal:
         return reconstructPath(start, goal)  // Goal reached, return the optimal path

      for neighbor in graph.neighbors(currentNode):
         newCost = costSoFar[currentNode] + graph.edgeCost(currentNode, neighbor)

         if neighbor not in costSoFar or newCost < costSoFar[neighbor]:
            costSoFar[neighbor] = newCost
            frontier.enqueue(neighbor, newCost)

   // No path found
   return null


function reconstructPath(start, goal):
   // Reconstruct the optimal path from start to goal
   path = []
   currentNode = goal

   while currentNode != start:
      path.insert(0, currentNode)
      currentNode = currentNode.parent
```

```
            path.insert(0, start)
            return path
```

Flowchart of Uniform Cost Search


Start
Assign a distance value 0 to the source vertex
Create an empty vertex set
Add the source vertex into the set
Stop
Yes
Is the vertext set empty?
No
Extract the vertex with smallest distance value from the set
No more neighbor
For each neighbor of the extracted vertex
Next neighbor
Add the neighbor vertex's into the set
Update the neighbor vertex's distance value
No
Yes
calculate distance value for the neighbor vertex
Is the neigbhor already in the vertex set?

Implementation

```
import heapq

class Node:
    def __init__(self, state, cost, parent=None):
        self.state = state
        self.cost = cost
        self.parent = parent

    def __lt__(self, other):
        return self.cost < other.cost

def uniform_cost_search(initial_state, goal_test, successors):
    # Create a priority queue to store the nodes
    queue = []
    # Create a set to store the visited states
    visited = set()
```

```
 # Create the initial node with cost 0
 initial_node = Node(initial_state, 0)
 # Add the initial node to the queue
 heapq.heappush(queue, initial_node)

 while queue:
     # Pop the node with the lowest cost from the queue
     current_node = heapq.heappop(queue)

     # Check if the current node is the goal
     if goal_test(current_node.state):
         # If it is, construct the path from the initial state to the goal state
         path = []
         while current_node:
             path.append(current_node.state)
             current_node = current_node.parent
         path.reverse()
         return path

     # Add the current state to the visited set
     visited.add(current_node.state)

     # Generate the successors of the current state
     for successor_state, successor_cost in successors(current_node.state):
         # Calculate the total cost of the successor node
         total_cost = current_node.cost + successor_cost

         # Create the successor node
         successor_node = Node(successor_state, total_cost, current_node)

         # Check if the successor state has already been visited
         if successor_state not in visited:
             # Add the successor node to the queue
             heapq.heappush(queue, successor_node)
         else:
             # If the successor state has already been visited,
             # find the node with the same state in the queue
             # and update its cost if the new cost is lower
             for node in queue:
                 if node.state == successor_state and node.cost > total_cost:
                     node.cost = total_cost
                     node.parent = current_node

 # If no path is found, return None
return None
```

```
# Example usage

# Define the goal test function
def goal_test(state):
    return state == "G"

# Define the successors function
def successors(state):
    if state == "A":
        return [("B", 1), ("C", 3)]
    elif state == "B":
        return [("D", 5), ("E", 2)]
    elif state == "C":
        return [("F", 4)]
    elif state == "E":
        return [("G", 1)]
    else:
        return []

# Define the initial state
initial_state = "A"

# Run the uniform cost search algorithm
result = uniform_cost_search(initial_state, goal_test, successors)

# Print the result
if result:
    print("Path found:", result)
else:
    print("No path found.")
```

Iterative Deepening Depth-First Search

Pseudo code for Iterative Deepening Depth-First Search

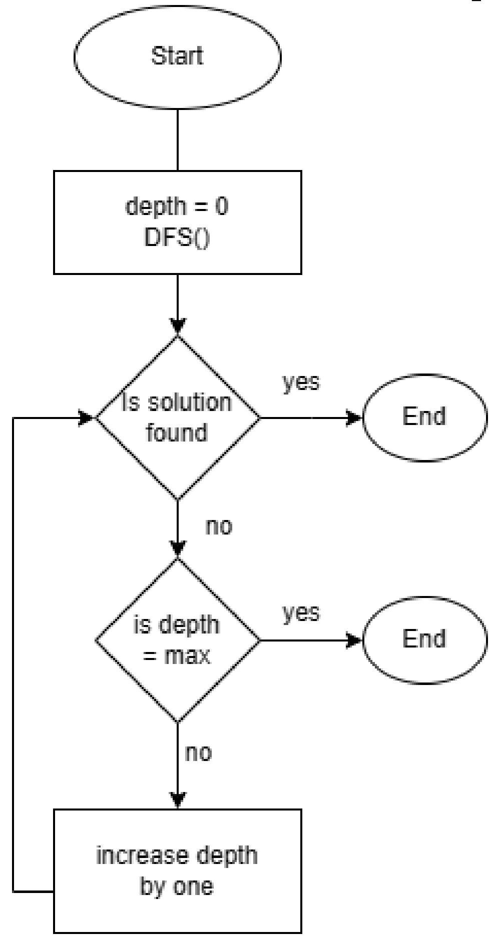
The IDDFS algorithm performs a series of depth-limited searches, gradually increasing the depth limit in each iteration. It combines the completeness of breadth-first search (BFS) with the space efficiency of depth-first search (DFS) by avoiding the need to store the entire search tree in memory. It explores the search space in a depth-first manner but with increasing depth limits to ensure that the goal node is eventually found if it exists within the search space.

1. Start with a depth limit of 0.
2. Initialize an empty set to track visited nodes.

3. Add the start node to the visited set.
4. Perform a depth-limited search from the start node using the current depth limit.
    a. Call the `depthLimitedSearch()` function with the start node, goal node, visited set, and depth limit.
    b. Inside the `depthLimitedSearch()` function:
    c. If the current node is the goal node, return the path from the start node to the current node.
    d. If the depth limit is 0, return null to indicate failure.
    e. Iterate over the neighbors of the current node.
    f. For each unvisited neighbor, add it to the visited set.
    g. Recursively call `depthLimitedSearch()` on the neighbor with the reduced depth limit.
    h. If the recursive call returns a non-null value, prepend the current node to the path and return it.
    i. If no goal node is found among the neighbors, return null.
5. If the depth-limited search returns a non-null value, it means the goal node has been found. Return the path from the start node to the goal node.
6. If the depth-limited search returns null, increment the depth limit by 1 and go back to step 2.
7. If all depth limits have been exhausted and the goal node has not been found, return null to indicate failure.

```
function iterativeDeepeningSearch(graph, start, goal):
   for depthLimit = 0 to infinity:
      visited = {}  // Initialize an empty set to track visited nodes
      visited.add(start)
      result = depthLimitedSearch(start, goal, visited, depthLimit)

      if result is not null:
         return result  // Goal found, return the path

   return null  // Goal not found within any depth limit


function depthLimitedSearch(node, goal, visited, depthLimit):
   if node == goal:
      return [node]  // Goal found, return the path

   if depthLimit == 0:
      return null  // Reached depth limit, return failure

   for neighbor in graph.neighbors(node):
      if neighbor not in visited:
         visited.add(neighbor)
         result = depthLimitedSearch(neighbor, goal, visited, depthLimit - 1)

         if result is not null:
             result.insert(0, node)  // Prepend the current node to the path
             return result  // Goal found, return the path

   return null  // Goal not found at this depth
```

Flow-chart for Iterative Deepening Depth-First Search



Implementation

```
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent

def iterative_deepening_search(initial_state, goal_test, successors, max_depth):
    for depth in range(max_depth):
        result = depth_limited_search(initial_state, goal_test, successors, depth)
        if result is not None:
            return result
    return None

def depth_limited_search(state, goal_test, successors, max_depth, depth=0):
    if depth > max_depth:
        return None

    if goal_test(state):
        return state

    for successor_state in successors(state):
        result = depth_limited_search(successor_state, goal_test, successors, max_depth, depth + 1)
        if result is not None:
            return result

   return None
   # Example usage

   # Define the goal test function
   def goal_test(state):
       return state == "G"

   # Define the successors function
   def successors(state):
       if state == "A":
           return ["B", "C"]
       elif state == "B":
           return ["D", "E"]
       elif state == "C":
           return ["F"]
       elif state == "E":
           return ["G"]
       else:
           return []

   # Define the initial state
   initial_state = "A"

   # Define the maximum depth
   max_depth = 3

   # Run the iterative deepening search algorithm
   result = iterative_deepening_search(initial_state, goal_test, successors, max_depth)
```

```
# Print the result
if result:
    print("Goal state found:", result)
else:
    print("Goal state not found within the maximum depth.")
```

Informed Search Algorithm

Best-First Search Algorithm

Pseudo code for Best-First Search

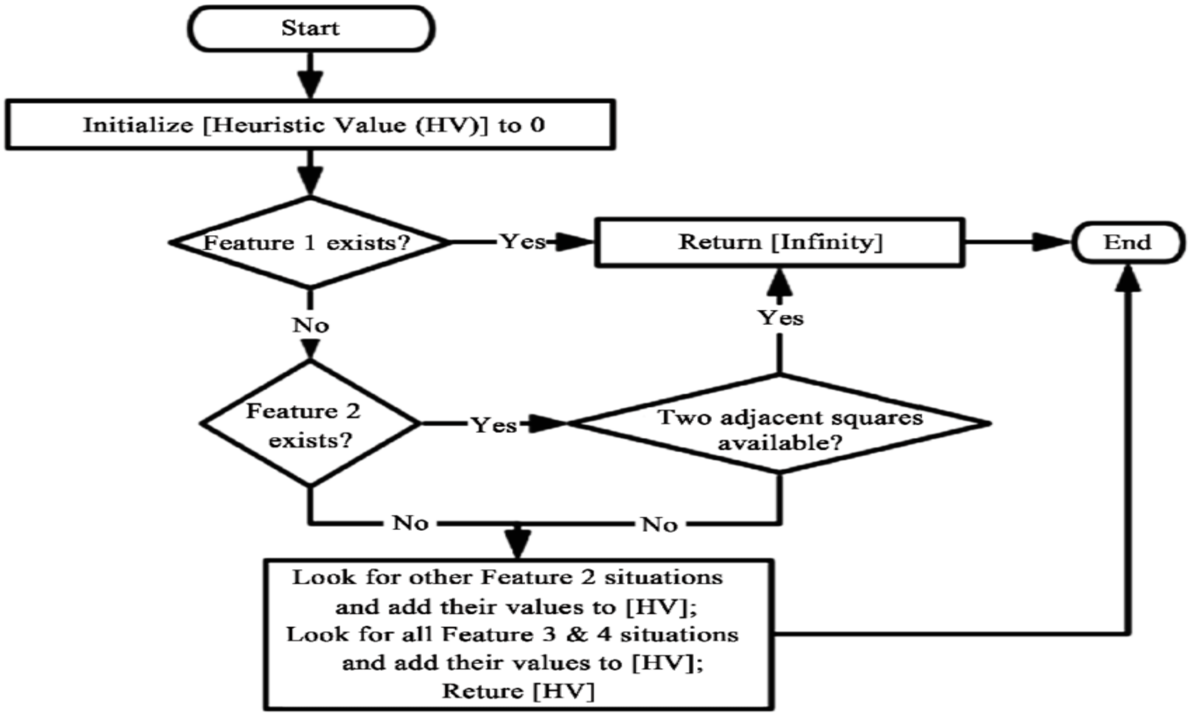
Best-First Search (BFS) is an informed graph traversal algorithm that explores a graph by selecting the most promising node based on a heuristic function. It combines the advantages of both breadth-first search (BFS) and depth-first search (DFS) algorithms. BFS uses a priority queue to prioritize nodes according to their heuristic values. The heuristic function provides an estimate of the cost from the current node to the goal node.

The pseudo code outlines the algorithm's steps:

1. Create two empty lists: OPEN and CLOSED.
2. Begin from the initial node (N) and add it to the ordered OPEN list.
3. Repeat the following steps until reaching the goal node:
   a. If the OPEN list is empty, exit the loop, returning ‘False’.
   b. Select the first/top node (N) from the OPEN list and move it to the CLOSED list. Also, record the parent node information.
   c. If N is the goal node, move the node to the Closed list and exit the loop returning ‘True’. The solution can be found by backtracking the path.
   d. If N is not the goal node, expand node N to generate immediate next nodes linked to node N and add them to the OPEN list.
   e. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n).

```
function BestFirstSearch(graph, start_node, target_node, heuristic_function):
    create an empty priority queue
    create an empty set to keep track of visited nodes
    enqueue start node into the priority queue with priority based on heuristic function(start node)
    while the priority queue is not empty:
        current node = dequeue from the priority queue
        if current node is the target node:
            return "Target node found"
        if current node is not in the visited set:
            add current node to the visited set
            for each neighbor node of current node:
                if neighbor node is not in the visited set:
                    enqueue neighbor node into the priority queue with priority based on heuristic function(neighbor node)
    return "Target node not found"
```

Flowchart for Best-First Search


Start
Initialize [Heuristic Value (HV)] to 0
Feature 1 exists?
Yes
Return [Infinity]
End
No
Yes
Feature 2 exists?
Yes
Two adjacent squares available?
No
No
Look for other Feature 2 situations and add their values to [HV]; Look for all Feature 3 & 4 situations and add their values to [HV]; Reture [HV]

Implementation

```
import heapq

def best_first_search(graph, start, goal, heuristic):
    visited = set()
    priority_queue = [(heuristic[start], start)]
    heapq.heapify(priority_queue)

    while priority_queue:
        _, current_node = heapq.heappop(priority_queue)
        visited.add(current_node)
        print(current_node, end=' ')  # Print node instead of storing it for this example

        if current_node == goal:
            return True  # Goal node found

        for neighbor in graph[current_node]:
            if neighbor not in visited:
                priority = heuristic[neighbor]
                heapq.heappush(priority_queue, (priority, neighbor))

    return False  # Goal node not found

# Example
 graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

heuristic = {
    'A': 5,
    'B': 3,
    'C': 2,
    'D': 4,
    'E': 1,
    'F': 2
}

start_node = 'A'
goal_node = 'F'
```

```
found = best_first_search(graph, start_node, goal_node, heuristic)

if found:
   print("\nGoal node found!")
else:
   print("\nGoal node not found!")
```

Output:

```
A C F
Goal node found!
```

A* Algorithm

Pseudo code for A* Algorithm

```
function A_Star(start, goal):
   // Create open and closed sets
   openSet = PriorityQueue()
   closedSet = Set()

   // Initialize the start node
   startNode = Node(start)
   startNode.gCost = 0
   startNode.hCost = heuristic(start, goal)
   startNode.fCost = startNode.gCost + startNode.hCost

   // Add the start node to the open set
   openSet.add(startNode)

   while openSet is not empty:
      // Get the node with the lowest fCost from the open set
      currentNode = openSet.pop()

      // Check if the current node is the goal node
      if currentNode is goal:
         return reconstructPath(currentNode)

      // Add the current node to the closed set
      closedSet.add(currentNode)

      // Explore the neighbors of the current node
      for each neighbor in currentNode.neighbors:
         // Skip the neighbor if it is in the closed set
```

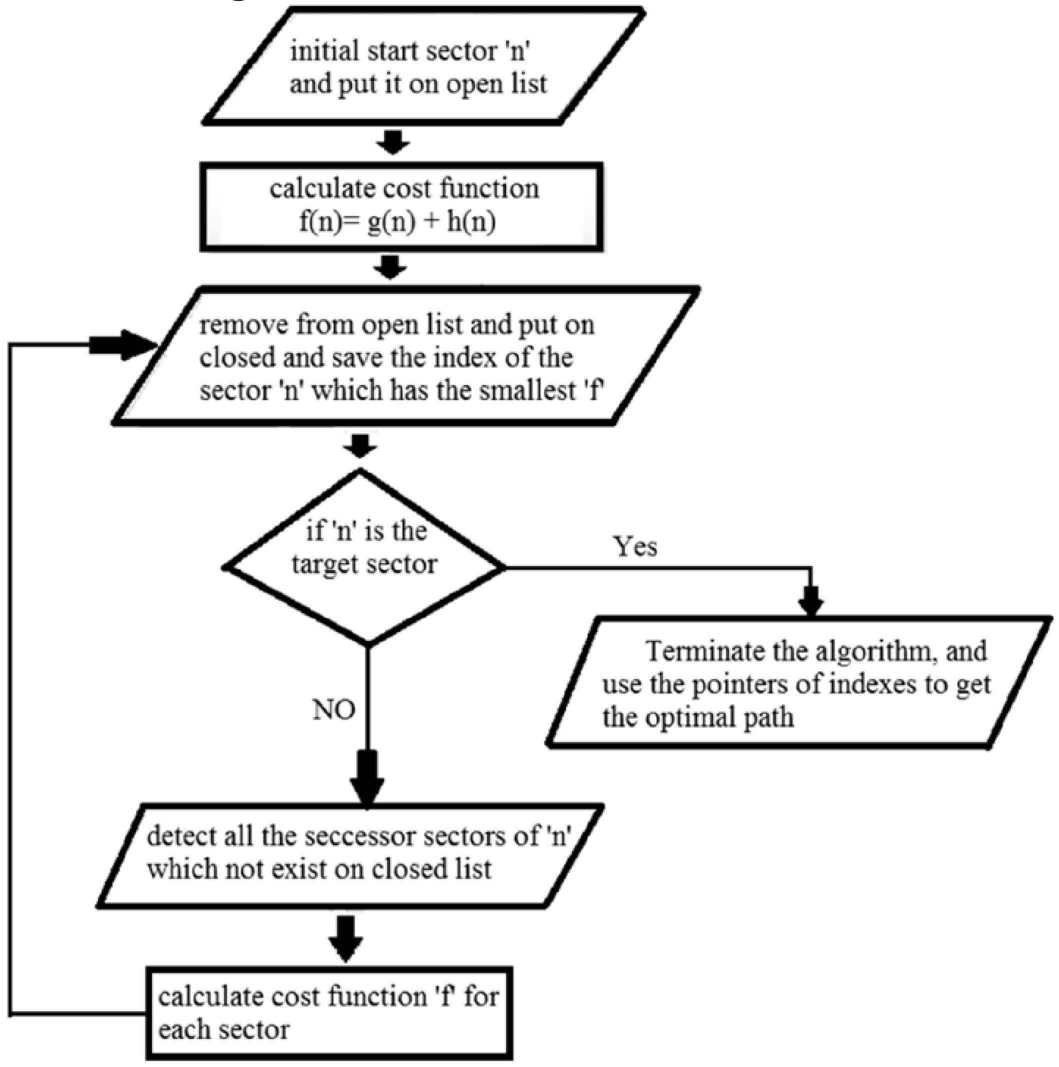
```
            if neighbor is in closedSet:
                continue

            // Calculate the tentative gCost for the neighbor
            tentativeGCost = currentNode.gCost + distance(currentNode, neighbor)

            // Check if the neighbor is already in the open set
            if neighbor is not in openSet:
                // Add the neighbor to the open set and calculate its costs
                neighbor.gCost = tentativeGCost
                neighbor.hCost = heuristic(neighbor, goal)
                neighbor.fCost = neighbor.gCost + neighbor.hCost
                neighbor.parent = currentNode
                openSet.add(neighbor)
            else if tentativeGCost < neighbor.gCost:
                // Update the neighbor's costs and parent if the new path is better
                neighbor.gCost = tentativeGCost
                neighbor.fCost = neighbor.gCost + neighbor.hCost
                neighbor.parent = currentNode

    // No path found
    return null

function reconstructPath(node):
    path = []
    while node is not null:
        path.add(node)
        node = node.parent
    return reverse(path)
```

In this pseudo code, start represents the starting position, goal represents the target position, and heuristic is a function that estimates the cost from a given node to the goal node. The distance function calculates the cost to move from one node to its neighbor. The implementation uses a priority queue (`openSet`) to store the nodes to be explored and a set (`closedSet`) to store the nodes that have already been visited.

The algorithm starts by initializing the start node with its costs and adding it to the open set. It then enters a loop where it selects the node with the lowest total cost (`fCost`) from the open set. The algorithm checks if the current node is the goal node, and if so, it reconstructs the path from the start to the goal node and returns it. Otherwise, it adds the current node to the closed set and explores its neighbors. For each neighbor, it calculates the tentative cost to reach that neighbor and checks if it is already in the open set. If the neighbor is not in the open set, it is added with its costs calculated. If the neighbor is already in the open set, the algorithm checks if the new path has a lower cost and updates the neighbor's costs and parent if needed. The loop continues until the open set is empty, indicating that no path to the goal was found.

The reconstructPath function is used to trace back the path from the goal node to the start node by following the parent pointers. The resulting path is then reversed to obtain the correct order. Modification and adaptation depending on the specific problem and programming language used.

Flowchart for A* Algorithm


initial start sector 'n' and put it on open list
calculate cost function f(n)= g(n) + h(n)
remove from open list and put on closed and save the index of the sector 'n' which has the smallest 'f'
if 'n' is the target sector
Yes
Terminate the algorithm, and use the pointers of indexes to get the optimal path
NO
detect all the seccessor sectors of 'n' which not exist on closed list
calculate cost function 'f' for each sector

Implementation

```
import heapq

def heuristic(node, goal_node):
    # Example heuristic function (number of nodes between current node and goal node)
    return abs(ord(goal_node) - ord(node))

def astar(graph, start_node, goal_node, heuristic):
    # Initialize the open and closed sets
    open_set = [(0, start_node)]  # Priority queue, using f-score as the priority
    closed_set = set()

    # Initialize dictionaries to store g-scores and parents
    g_scores = {node: float('inf') for node in graph}
    g_scores[start_node] = 0
    parents = {}

    while open_set:
        # Pop the node with the lowest f-score from the open set
        f_score, current_node = heapq.heappop(open_set)

        if current_node == goal_node:
            # Goal node reached, reconstruct and return the path
            path = reconstruct_path(parents, current_node)
            return path

        closed_set.add(current_node)

        # Explore the neighbors of the current node
        for neighbor in graph[current_node]:
            # Calculate the tentative g-score for the neighbor
            tentative_g_score = g_scores[current_node] + graph[current_node][neighbor]

            if tentative_g_score < g_scores[neighbor]:
                # Update the g-score and parent for the neighbor
                g_scores[neighbor] = tentative_g_score
                parents[neighbor] = current_node

                # Calculate the f-score for the neighbor
                f_score = tentative_g_score + heuristic(neighbor, goal_node)

                if neighbor not in closed_set:
                    # Add the neighbor to the open set if it's not already there
                    heapq.heappush(open_set, (f_score, neighbor))
    # No path found
    return None

def reconstruct_path(parents, node):
    path = [node]
    while node in parents:
        node = parents[node]
        path.append(node)
    path.reverse()
    return path

# Example usage:
graph = {
    'A': {'B': 5, 'C': 3},
    'B': {'D': 2, 'E': 4},
    'C': {'F': 6},
    'D': {'G': 7},
    'E': {'G': 3},
    'F': {'G': 1},
    'G': {}
}

start_node = 'A'
goal_node = 'G'

path = astar(graph, start_node, goal_node, heuristic)
print("Path:", path)
```

Output: Path: ['A', 'C', 'F', 'G']

AO* Search Algorithm

The AO* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept. AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems. The AND side of the graph represents a set of tasks that must be completed to achieve the main goal, while the OR side of the graph represents different methods for accomplishing the same main goal. Best-first search is what the AO* algorithm does.

Algorithm of AO* search algorithm

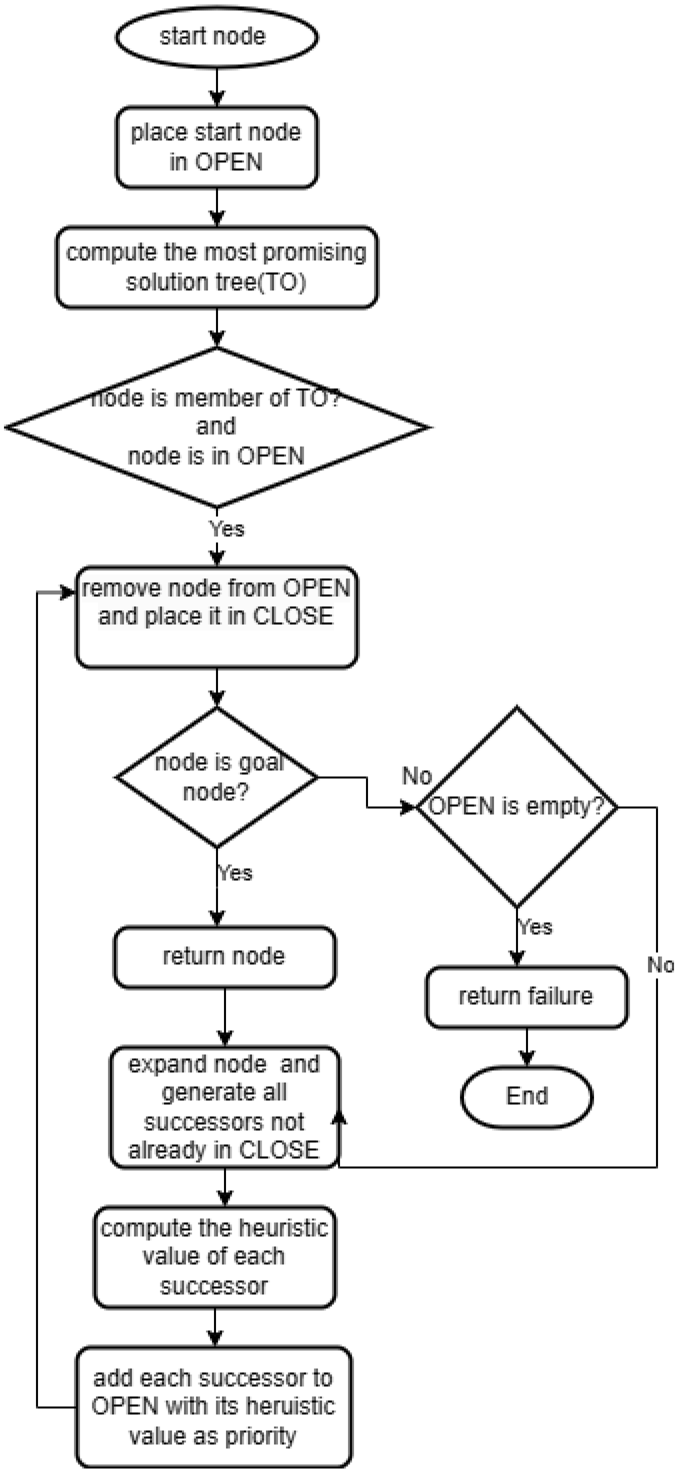
1. Place the starting node into OPEN.
2. Compute the most promising solution tree, say T0.

3. Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in CLOSE.
4. If n is the terminal goal node, then level n as solved and level all the ancestors of n as solved.
5. Generate all the successors of n that are not already on CLOSE.
6. Compute the heuristic value of each successor.
7. Add each successor to OPEN with its heuristic value as its priority.

Pseudo code for AO* search algorithm

```
function AO_Search(node, problem) returns solution or failure
  if Goal-Test(problem, State[node]) then return node
  else if Failure-Test(problem, State[node]) then return failure
  else
    for each child in Expand(node, problem) do
      result = AO_Search(child, problem)
      if result != failure then return result
    return failure
```

Flowchart for AO* Search Algorithm


start node
place start node in OPEN
compute the most promising solution tree(TO)
node is member of TO? and node is in OPEN
Yes
remove node from OPEN and place it in CLOSE
node is goal node?
No
OPEN is empty?
Yes
return node
Yes
return failure
No
expand node and generate all successors not already in CLOSE
End
compute the heuristic value of each successor
add each successor to OPEN with its heruistic value as priority

Implementation

AO* search algorithm phyton code implementation

```
# Import heapq module
import heapq

# Define the AO* algorithm
def ao_star(start, goal, h_func, succ_func):
    # Initialize the open list
    open_list = [(h_func(start), start)]
    # Initialize the closed list
    closed_list = set()
    # Initialize the g score
    g_score = {start: 0}
    # Initialize the f score
    f_score = {start: h_func(start)}
    # Initialize the came_from dictionary
    came_from = {}

    # Loop until the open list is empty
    while open_list:
        # Get the node with the lowest f score
        current = heapq.heappop(open_list)[1]
        # Check if the current node is the goal
        if current == goal:
            # Reconstruct the path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            path.reverse()
            return path
        # Add the current node to the closed list
        closed_list.add(current)
        # Loop through the successors of the current node
        for successor in succ_func(current):
            # Calculate the tentative g score
            tentative_g_score = g_score[current] + 1
            # Check if the successor is already in the closed list
            if successor in closed_list:
                # Check if the tentative g score is greater than or equal to the existing g score
                if tentative_g_score >= g_score.get(successor, float('inf')):
                    continue
            # Check if the successor is not in the open list
            if successor not in [i[1] for i in open_list]:
                # Add the successor to the open list
                heapq.heappush(open_list, (tentative_g_score + h_func(successor), successor))
            # Update the came_from dictionary
            came_from[successor] = current
            # Update the g score
            g_score[successor] = tentative_g_score
            # Update the f score
            f_score[successor] = tentative_g_score + h_func(successor)

    # If the open list is empty and the goal has not been found, return None
    return None



#Example of an AO* search algorithm implementation
# Cost to find the AND and OR path
def Cost(H, condition, weight = 1):
    cost = {}
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node]+weight for node in AND_nodes)
        cost[Path_A] = PathA

    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B =' OR '.join(OR_nodes)
        PathB = min(H[node]+weight for node in OR_nodes)
        cost[Path_B] = PathB
    return cost

# Update the cost
def update_cost(H, Conditions, weight=1):
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()
    least_cost= {}
    for key in Main_nodes:
        condition = Conditions[key]
        print(key,':', Conditions[key],'>>>', Cost(H, condition, weight))
        c = Cost(H, condition, weight)
        H[key] = min(c.values())
        least_cost[key] = Cost(H, condition, weight)
    return least_cost

# Print the shortest path
def shortest_path(Start,Updated_cost, H):
    Path = Start
    if Start in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)

        # FIND MINIMIMUM PATH KEY
        Next = key[Index].split()
        # ADD TO PATH FOR OR PATH
        if len(Next) == 1:

            Start =Next[0]
            Path += '<--' +shortest_path(Start, Updated_cost, H)
        # ADD TO PATH FOR AND PATH
        else:
            Path +='<--('+key[Index]+') '

            Start = Next[0]
            Path += '[' +shortest_path(Start, Updated_cost, H) + ' + '

            Start = Next[-1]
            Path +=  shortest_path(Start, Updated_cost, H) + ']'

    return Path


H = {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I':0, 'J':0}

Conditions = {
 'A': {'OR': ['B'], 'AND': ['C', 'D']},
 'B': {'OR': ['E', 'F']},
 'C': {'OR': ['G'], 'AND': ['H', 'I']},
 'D': {'OR': ['J']}
}
# weight
weight = 1
# Updated cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

Output:

```
Updated Cost :
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}
Shortest Path :
 A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```

Conclusion

In conclusion, uninformed search and informed search algorithms differ in their approach to exploring a search space. Uninformed search algorithms, such as depth-first search (DFS) and breadth-first search (BFS), systematically explore the search space without any additional information about the problem domain. These algorithms are relatively simple to implement and guarantee completeness in finding a solution if one exists. However, they may not be efficient in terms of time or space complexity, especially when dealing with large or complex search spaces.

On the other hand, informed search algorithms, such as A* search and Greedy Best-First search, utilize heuristic information to guide the search towards the goal more efficiently. These algorithms make use of heuristics or cost estimates to prioritize the exploration of promising paths. By incorporating additional knowledge, informed search algorithms can often find solutions more quickly and with fewer expanded nodes compared to uninformed search algorithms. However, the quality of the heuristic and the accuracy of the estimated costs greatly impact the performance and optimality of these algorithms.

Implementing these search algorithms involves encoding the problem domain, defining the search space, and designing the data structures and algorithms accordingly. The choice of programming language and data structures depends on the specific requirements of the problem and the available resources. Python, with its simplicity and rich ecosystem of libraries, is a popular choice for implementing search algorithms.

By implementing search algorithms in Python, we can explore and solve a wide range of problems, such as path finding in graphs, puzzle solving, and optimization. These algorithms can be implemented using a variety of techniques, including recursive or iterative approaches, data structures like queues or stacks, and leveraging object-oriented programming principles.

In summary, the implementation of search algorithms in Python provides a powerful toolkit for solving complex problems by systematically exploring the search space. The choice of algorithm depends on the problem requirements, available resources, and trade-offs between completeness, optimality, and efficiency.