

EXPERIMENT 04

Name: Rohit Vilas Bhadane

Date: 02.03.2023

Roll No: 04

Grade:

Class: TE Comps

Sign:

Aim: To study uninformed search strategy BFS and DFS and Informed search strategy A* algorithm

Theory:

BREADTH FIRST SEARCH (BFS)

- **Uninformed Search:**

Uninformed search, also known as blind search, is a search strategy in artificial intelligence that explores a problem space without any prior knowledge or heuristic information about the problem being solved.

In an uninformed search, the search algorithm does not use any domain-specific information to guide its search. Instead, it systematically generates new candidate solutions and tests them against the problem goal until it finds a solution or determines that none exists.

There are many types of uninformed search like :

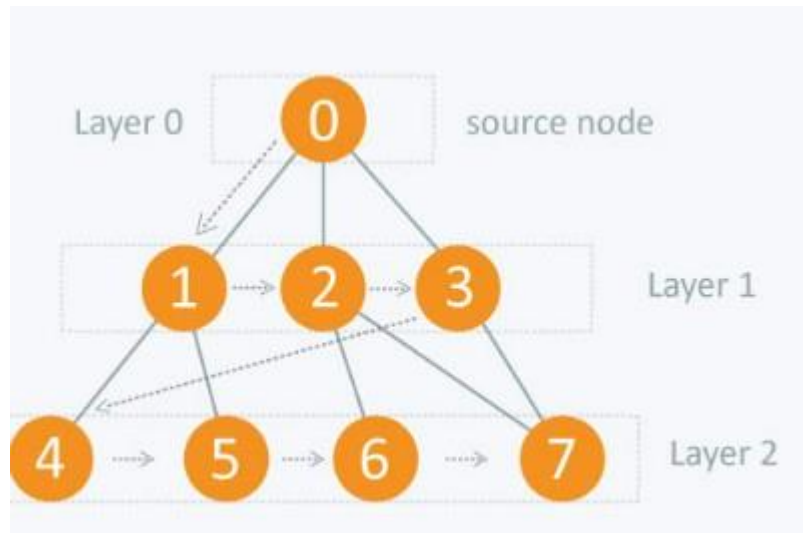
- breadth-first search
- depth search search
- uniform cost search etc.

Breadth First search:

Breadth-First Search (BFS) is an algorithm used in artificial intelligence to traverse and explore a search space or a graph. It is a type of uninformed search algorithm that explores all the neighbours of a node at each level before moving on to the neighbours of those nodes. BFS starts at a particular node in a graph and explores all the nodes at the same level before moving on to the next level. It visits all the nodes in a level before moving to the next level. This means that BFS generates a tree, which is a level-by-level expansion of the search space.

The BFS algorithm uses a queue data structure to store the nodes to be explored. The algorithm starts by enqueueing the starting node and marking it as visited. Then, it dequeues the node, explores all its unvisited neighbours, and enqueues them. The algorithm repeats this process until it has explored all reachable nodes or until it finds the goal node

DIAGRAM:



ALGORITHM:

1. Create an empty queue Q and an empty set Visited.
2. Enqueue the start vertex into Q.
3. Add the start vertex to Visited.
4. While Q is not empty:
5. Dequeue a vertex v from Q.
6. For each neighbor w of v:
7. If w is not in Visited:
8. Enqueue w into Q.
9. Add w to Visited.
10. Process vertex v (e.g. print its value).
11. End while.

CODE:

```

from collections import deque
def bfs(graph, start):
    # Create a queue for BFS
    queue = deque([start])
    # Mark the start node as visited
    visited = set([start])

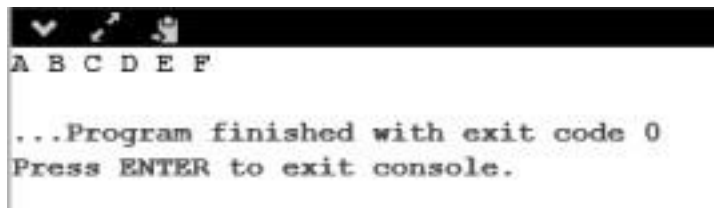
    while queue:
        # Dequeue a vertex from the queue
        vertex = queue.popleft()
        print(vertex, end=' ')
        # Visit all adjacent nodes of the dequeued vertex
        for neighbor in graph[vertex]:
            if neighbor not in visited:

```

```
# Mark the neighbor as visited and enqueue it
visited.add(neighbor)
queue.append(neighbor)
```

```
# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
bfs(graph, 'A')
```

OUTPUT:

A screenshot of a terminal window with a black title bar containing three icons. The terminal output shows a header line 'A B C D E F' followed by a blank line. Then, it displays '...Program finished with exit code 0' and 'Press ENTER to exit console.' on separate lines.

```
A B C D E F

...Program finished with exit code 0
Press ENTER to exit console.
```

DEPTH FIRST SEARCH (DFS):

DFS, or Depth-First Search, is a graph traversal algorithm that is commonly used in AI for various tasks such as pathfinding, puzzle solving, and game playing.

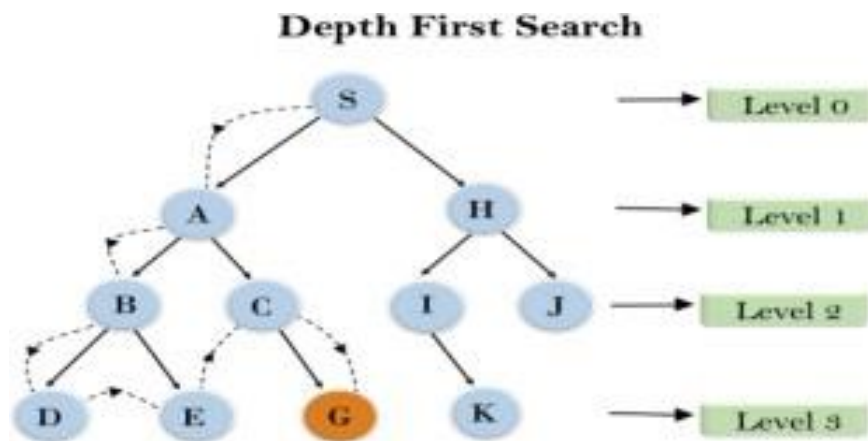
The algorithm starts at a root node and explores as far as possible along each branch before backtracking. This means that it prioritizes visiting the deepest unexplored nodes first, and only returns to earlier nodes once all possibilities at a given depth have been exhausted.

In the context of pathfinding, DFS can be used to find a path from a starting node to a goal node by exploring all possible paths. However, since DFS does not consider the cost of each path, it may not always find the shortest path.

DFS can also be used in conjunction with heuristics to search for a solution to a puzzle, such as the 8-puzzle or the Rubik's cube. By exploring the state space of the puzzle and attempting various moves, DFS can eventually find a solution if one exists. However, as with pathfinding, DFS may not always find the optimal solution.

Overall, DFS is a powerful algorithm that is simple to implement and can be used in a wide variety of AI applications. However, it is important to consider its limitations, such as its inability to handle cycles in graphs and its tendency to get stuck in infinite loops.

DIAGRAM:



ALGORITHM :

- Choose a starting node and mark it as visited.
- Push the starting node onto a stack.
- While the stack is not empty, do the following:
 - a. Pop the top node from the stack.
 - b. If the popped node is the goal node, return success and stop the search.
 - c. Otherwise, for each adjacent node of the popped node that has not been visited, mark it as visited and push it onto the stack.

- If the stack becomes empty and the goal node has not been found, return failure.

CODE:

```
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()
        visited.add(start)
        print(start)
        for neighbor in graph[start] - visited:
            dfs_recursive(graph, neighbor, visited)
    graph = {
        1: {2, 3},
        2: {1, 3, 4, 5},
        3: {1, 2, 6, 7},
        4: {2, 5},
        5: {2, 4},
        6: {3},
        7: {3},
    }
    dfs_recursive(graph, 1)
```

OUTPUT:

```
1
2
3
6
7
4
5
5
3
```

```
...Program finished with exit code 0
Press ENTER to exit console.□
```

A* ALGORITHM

Informed Search:

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

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

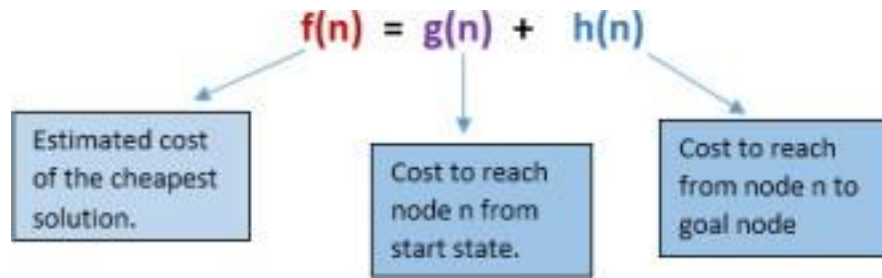
A* Algorithm:

A* is a popular and efficient search algorithm used in artificial intelligence and many other fields. It is a pathfinding algorithm that is used to find the shortest path between two points in a graph or a grid. A* is a best-first search algorithm that combines the advantages of Dijkstra's algorithm (which is an uninformed search algorithm that guarantees the shortest path) and greedy best-first search (which is a heuristic search algorithm that is faster but does not guarantee the shortest path).

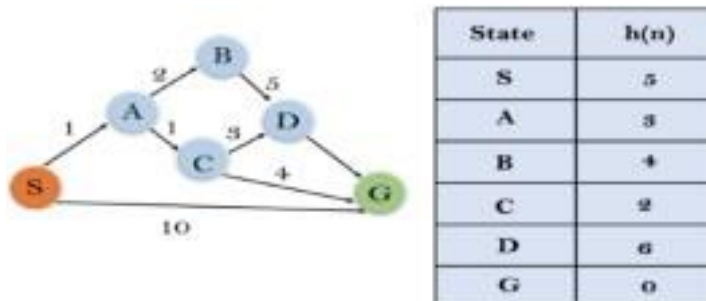
The algorithm works by maintaining a priority queue of nodes to be expanded. Each node is assigned a score that represents the cost of the path from the start node to that node and an estimate of the cost of the path from that node to the goal node (called the heuristic value). The score is calculated as the sum of the cost and the heuristic value. The algorithm expands the node with the lowest score first and adds its neighbours to the priority queue.

The heuristic function used in A* is a permissible and consistent function that estimates the cost of the path from a node to the goal node. An admissible heuristic is one that never overestimates the actual cost, and a consistent heuristic is one that satisfies the triangle inequality (i.e., the heuristic value of a node is always less than or equal to the sum of the cost of the path from the node to its neighbour and the heuristic value of the neighbour).

A* guarantees to find the optimal path (i.e., the shortest path) as long as the heuristic function is admissible. A* is widely used in many applications, including robotics, video games, and route planning. However, A* can be computationally expensive if the graph or the grid is large or complex, and other search algorithms, such as Dijkstra's algorithm or breadth-first search, may be more suitable for some cases.



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



Solution:

Initialization: $\{(S, 5)\}$

Iteration 1: $\{(S \rightarrow A, 4), (S \rightarrow G, 10)\}$

Iteration 2: $\{(S \rightarrow A \rightarrow C, 4), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration 3: $\{(S \rightarrow A \rightarrow C \rightarrow G, 6), (S \rightarrow A \rightarrow C \rightarrow D, 11), (S \rightarrow A \rightarrow B, 7), (S \rightarrow G, 10)\}$

Iteration 4: will give the final result, as $S \rightarrow A \rightarrow C \rightarrow G$ it provides the optimal path with cost 6.

ALGORITHM:

- Step 1: Place the starting node in the OPEN list.
- Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stop.
- Step 3: Select the node from the OPEN list which has the smallest value of evaluation

- function (g+h), if node n is the goal node then return success and stop, otherwise
- Step 4: Expand node n, generate all its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute the evaluation function for n' and place it into the Open list.
- Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- Step 6: Return to Step 2.

CODE:

import heap

```
def astar(start, goal, h_func, neighbors_func):
    """A* search algorithm."""
    heap = [(0, start)]
    came_from = {start: None}
    cost_so_far = {start: 0}

    while heap:
        _, current = heapq.heappop(heap)

        if current == goal:
            path = [current]
            while current != start:
                current = came_from[current]
                path.append(current)
            path.reverse()
            return path

        for neighbor in neighbors_func(current):
            new_cost = cost_so_far[current] + 1
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + h_func(neighbor, goal)
                heapq.heappush(heap, (priority, neighbor))
                came_from[neighbor] = current

    return None

def euclidean_dist(a, b):
    return ((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2) ** 0.5

def grid_neighbors(node):
    x, y = node
    return [(x-1, y), (x, y-1), (x+1, y), (x, y+1)]

grid = [[0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0],
```



```
[0, 1, 0, 0, 0],  
[0, 1, 0, 0, 0],  
[0, 0, 0, 0, 0]]  
start = (0, 0)  
goal = (4, 4)  
  
path = astar(start, goal, euclidean_dist,  
grid_neighbors)print(path)
```

OUTPUT:



```
[(0, 0), (0, 1), (1, 1), (1, 2), (2, 2), (2, 3), (3, 3), (3, 4), (4, 4)]  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Conclusion:

Hence, We have successfully studied and implemented uninformed search techniques BFS and DFS and informed search technique A* Algorithm.