Experiment No.4
Study and Implementation of Informed search method: A* Search algorithm.
Date of Performance:
Date of Submission:

Aim: Study and Implementation of A* search algorithm.

Objective: To study the informed searching techniques and its implementation for problem solving.

Theory:

A* (pronounced as "A star") is a computer algorithm that is widely used in path finding and graph traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph. However, the A* algorithm introduces a heuristic into a regular graphsearching algorithm, essentially planning ahead at each step so a more optimal decision is made.

A* is an extension of Dijkstra's algorithm with some characteristics of breadth-first search (BFS). Like Dijkstra, A* works by making a lowest-cost path tree from the start node to the target node. What makes A* different and better for many searches is that for each node, A* uses a function f(n)f(n)f(n) that gives an estimate of the total cost of a path using that node. Therefore, A* is a heuristic function, which differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct.
A* expands paths that are already less expensive by using this function:

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

where

- f(n) = total estimated cost of path through node n
- g(n) = cost so far to reach node n
- h(n) = estimated cost from n to goal. This is the heuristic part of the cost function, so it is like a guess.

Pseudocode

```
import math
import heapq

class Cell:
        def __init__(self):
                self.parent_i = 0
                self.parent_j = 0
```

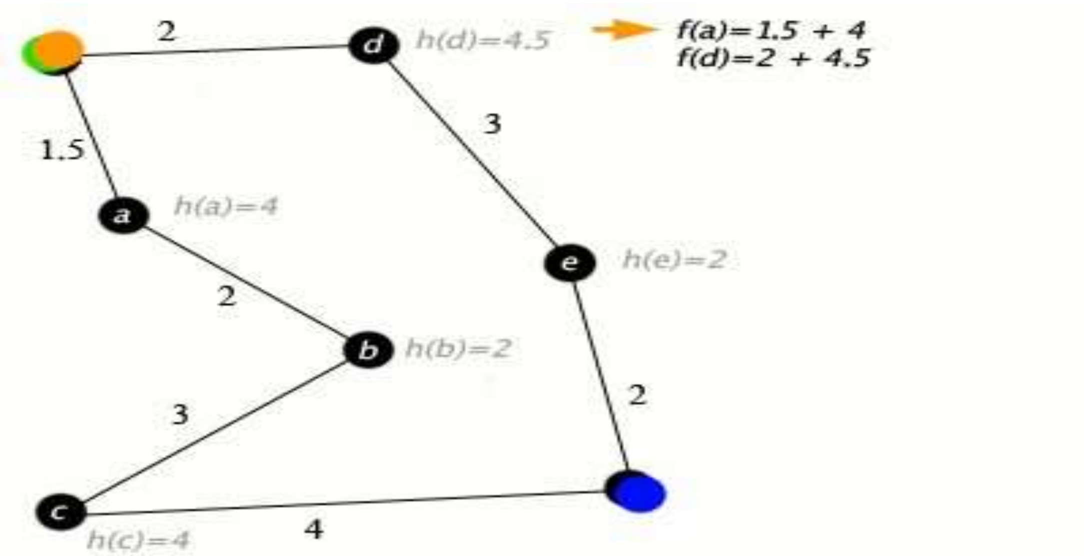
```
                self.f = float('inf')
                self.g = float('inf')
                self.h = 0
ROW = 9
COL = 10

def is_valid(row, col):
        return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

def is_unblocked(grid, row, col):
        return grid[row][col] == 1

def is_destination(row, col, dest):
        return row == dest[0] and col == dest[1]

def calculate_h_value(row, col, dest):
        return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5

def trace_path(cell_details, dest):
        print("The Path is ")
        path = []
        row = dest[0]
        col = dest[1]


        while not (cell_details[row][col].parent_i == row and
cell_details[row][col].parent_j == col):
                path.append((row, col))
                temp_row = cell_details[row][col].parent_i
                temp_col = cell_details[row][col].parent_j
                row = temp_row
                col = temp_col


        path.append((row, col))

        path.reverse()

        for i in path:
                print("->", i, end=" ")
        print()

def a_star_search(grid, src, dest):

        if not is_valid(src[0], src[1]) or not is_valid(dest[0],
dest[1]):
                print("Source or destination is invalid")
                return


        if not is_unblocked(grid, src[0], src[1]) or not
is_unblocked(grid, dest[0], dest[1]):
                print("Source or the destination is blocked")
```

```
            return


        if is_destination(src[0], src[1], dest):
            print("We are already at the destination")
            return


        closed_list = [[False for _ in range(COL)] for _ in range(ROW)]

        cell_details = [[Cell() for _ in range(COL)] for _ in
range(ROW)]

        i = src[0]
        j = src[1]
        cell_details[i][j].f = 0
        cell_details[i][j].g = 0
        cell_details[i][j].h = 0
        cell_details[i][j].parent_i = i
        cell_details[i][j].parent_j = j


        open_list = []
        heapq.heappush(open_list, (0.0, i, j))


        found_dest = False

        while len(open_list) > 0:

            p = heapq.heappop(open_list)


            i = p[1]
            j = p[2]
            closed_list[i][j] = True


            directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1),
(1, -1), (-1, 1), (-1, -1)]
            for dir in directions:
                new_i = i + dir[0]
                new_j = j + dir[1]


                if is_valid(new_i, new_j) and is_unblocked(grid,
new_i, new_j) and not closed_list[new_i][new_j]:

                    if is_destination(new_i, new_j, dest):

                        cell_details[new_i][new_j].parent_i
= i
```

```
                                        cell_details[new_i][new_j].parent_j
= j
                                        print("The destination cell is
found")

                                        trace_path(cell_details, dest)
                                        found_dest = True
                                        return
                                else:

                                        g_new = cell_details[i][j].g + 1.0
                                        h_new = calculate_h_value(new_i,
new_j, dest)
                                        f_new = g_new + h_new


                                        if cell_details[new_i][new_j].f ==
float('inf') or cell_details[new_i][new_j].f > f_new:

                                                heapq.heappush(open_list,
(f_new, new_i, new_j))

                                                cell_details[new_i][new_j].f
= f_new
                                                cell_details[new_i][new_j].g
= g_new
                                                cell_details[new_i][new_j].h
= h_new

        cell_details[new_i][new_j].parent_i = i

        cell_details[new_i][new_j].parent_j = j


        if not found_dest:
                print("Failed to find the destination cell")

def main():

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


        src = [8, 0]
```

```
        dest = [7, 2]

        a_star_search(grid, src, dest)

if __name__ == "__main__":
        main()

output:

The destination cell is found
The Path is
-> (8, 0) -> (8, 1) -> (7, 2)
```

An example of an A* algorithm in action where nodes are cities connected with roads and h(x) is the straight-line distance to target point:



Conclusion: In conclusion, implementing the A* search algorithm offers a powerful and efficient solution for navigating graphs or searching through paths in various applications such as robotics, gaming, and route planning. Its ability to combine the advantages of both Dijkstra's algorithm and greedy best-first search by considering both the cost-so-far and the estimated cost to the goal makes it highly effective in finding the shortest path while minimizing computational resources.