IMPLEMENTATION OF SEARCHING TECHNIQUES IN AI

UNIFORM COST SEARCH

```
# Python3 implementation of above approach
def uniform_cost_search(goal, start):

	# minimum cost upto
	# goal state from starting
	global graph,cost
	answer = []

	# create a priority queue
	queue = []

	# set the answer vector to max value
	for i in range(len(goal)):
		answer.append(10**8)

	# insert the starting index
	queue.append([0, start])

	# map to store visited node
	visited = {}

	# count
	count = 0

	# while the queue is not empty
	while (len(queue) > 0):

		# get the top element of the
```

```
queue = sorted(queue)
p = queue[-1]

# pop the element
del queue[-1]

# get the original value
p[0] *= -1

# check if the element is part of
# the goal list
if (p[1] in goal):

    # get the position
    index = goal.index(p[1])

    # if a new goal is reached
    if (answer[index] == 10**8):
        count += 1

    # if the cost is less
    if (answer[index] > p[0]):
        answer[index] = p[0]

    # pop the element
    del queue[-1]

    queue = sorted(queue)
    if (count == len(goal)):
        return answer

# check for the non visited nodes
```

```
            # which are adjacent to present node
            if (p[1] not in visited):
                for i in range(len(graph[p[1]])):

                    # value is multiplied by -1 so that
                    # least priority is at the top
                    queue.append( [(p[0] + cost[(p[1], graph[p[1]][i])])* -1,
graph[p[1]][i]])

            # mark as visited
            visited[p[1]] = 1

    return answer

# main function
if __name__ == '__main__':

    # create the graph
    graph,cost = [[] for i in range(8)],{}

    # add edge
    graph[0].append(1)
    graph[0].append(3)
    graph[3].append(1)
    graph[3].append(6)
    graph[3].append(4)
    graph[1].append(6)
    graph[4].append(2)
    graph[4].append(5)
    graph[2].append(1)
    graph[5].append(2)
    graph[5].append(6)
```

```
graph[6].append(4)

# add the cost
cost[(0, 1)] = 2
cost[(0, 3)] = 5
cost[(1, 6)] = 1
cost[(3, 1)] = 5
cost[(3, 6)] = 6
cost[(3, 4)] = 2
cost[(2, 1)] = 4
cost[(4, 2)] = 4
cost[(4, 5)] = 3
cost[(5, 2)] = 6
cost[(5, 6)] = 3
cost[(6, 4)] = 7

# goal state
goal = []

# set the goal
# there can be multiple goal states
goal.append(6)

# get the answer
answer = uniform_cost_search(goal, 0)

# print the answer
print("Minimum cost from 0 to 6 is = ",answer[0])
```

A* SEARCH

Input:

Start Node: A

Goal Node: F

Nodes: A, B, C, D, E, F

Edges with weights:

(A, B, 1), (A, C, 4),(B, D, 3), (B, E, 5),(C, F, 2),(D, F, 1), (D, E, 1),(E, F, 2)

Output:

```
# Python program for A* Search Algorithm

import math

import heapq


# Define the Cell class



class Cell:

   def __init__(self):

     # Parent cell's row index

       self.parent_i = 0

   # Parent cell's column index

       self.parent_j = 0

 # Total cost of the cell (g + h)

       self.f = float('inf')

   # Cost from start to this cell

       self.g = float('inf')

   # Heuristic cost from this cell to destination

       self.h = 0



# Define the size of the grid

ROW = 9

COL = 10
```

```python
# Check if a cell is valid (within the grid)


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

# Check if a cell is unblocked


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

# Check if a cell is the destination


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

# Calculate the heuristic value of a cell (Euclidean distance to destination)


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

# Trace the path from source to destination


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

```
    # Trace the path from destination to source using parent cells
    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

    # Add the source cell to the path
    path.append((row, col))
    # Reverse the path to get the path from source to destination
    path.reverse()

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

# Implement the A* search algorithm


def a_star_search(grid, src, dest):
    # Check if the source and destination are valid
    if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
        print("Source or destination is invalid")
        return

    # Check if the source and destination are unblocked
    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
```

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

# Initialize the closed list (visited cells)
closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
# Initialize the details of each cell
cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]

# Initialize the start cell details
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

# Initialize the open list (cells to be visited) with the start cell
open_list = []
heapq.heappush(open_list, (0.0, i, j))

# Initialize the flag for whether destination is found
found_dest = False

# Main loop of A* search algorithm
while len(open_list) > 0:
    # Pop the cell with the smallest f value from the open list
    p = heapq.heappop(open_list)
```

```
        # Mark the cell as visited
        i = p[1]
        j = p[2]
        closed_list[i][j] = True

        # For each direction, check the successors
        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 the successor is valid, unblocked, and not visited
            if is_valid(new_i, new_j) and is_unblocked(grid, new_i, new_j) and not closed_list[new_i][new_j]:
                # If the successor is the destination
                if is_destination(new_i, new_j, dest):
                    # Set the parent of the destination cell
                    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 and print the path from source to destination
                    trace_path(cell_details, dest)
                    found_dest = True
                    return
                else:
                    # Calculate the new f, g, and h values
                    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 the cell is not in the open list or the new f value is smaller
```

```
                if cell_details[new_i][new_j].f == float('inf') or cell_details[new_i][new_j].f > f_new:
                    # Add the cell to the open list
                    heapq.heappush(open_list, (f_new, new_i, new_j))
                    # Update the cell details
                    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 the destination is not found after visiting all cells
    if not found_dest:
        print("Failed to find the destination cell")

# Driver Code


def main():
    # Define the grid (1 for unblocked, 0 for blocked)
    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]
    ]

    # Define the source and destination
```

```python
    src = [8, 0]
    dest = [0, 0]

    # Run the A* search algorithm
    a_star_search(grid, src, dest)


if __name__ == "__main__":
    main()
```