# AI Assignment 2

**BINARY GRID PROBLEM CODE:**

```
from copy import deepcopy
import math
class State:
    def __init__(self, pos, grid, goal):
        self.pos = pos          # (i, j) coordinates
        self.grid = grid        # reference to the grid
        self.goal = goal        # goal coordinates
    def goalTest(self):
        return self.pos == self.goal
    def moveGen(self):
        children = []
        directions = [(-1,0), (1,0), (0,-1), (0,1),
                      (-1,-1), (-1,1), (1,-1), (1,1)]
        n = len(self.grid)
        i, j = self.pos
        for di, dj in directions:
            ni, nj = i + di, j + dj
            if 0 <= ni < n and 0 <= nj < n and self.grid[ni][nj] == 0:
                children.append(State((ni, nj), self.grid, self.goal))
        return children
    def __eq__(self, other):
        return self.pos == other.pos
    def __hash__(self):
        return hash(self.pos)
    def __str__(self):
        return str(self.pos)
    def h(self):
        # Euclidean distance
```

```
        (x1, y1) = self.pos
        (x2, y2) = self.goal
        return math.sqrt((x1-x2)**2 + (y1-y2)**2)
    def stepCost(self, other):
        return 1   # cost = 1 per step
    def reconstructPath(node, parent_map):
      path = [node.pos]
      parent = parent_map[node]
      while parent is not None:
        path.append(parent.pos)
        parent = parent_map[parent]
      path.reverse()
      return path
# ---------------- Best First Search ----------------
def best_first_search(start, grid):
    n = len(grid)
    if grid[0][0] == 1 or grid[n-1][n-1] == 1:   # Start or goal blocked
        return -1, []
    OPEN = [(start, None, start.h())]   # (Node, Parent, h)
    CLOSED = []
    parent_map = {start: None}
    while OPEN:
        OPEN.sort(key=lambda x: x[2])   # sort by h
        nodeTriple = OPEN.pop(0)
        N, parent, hVal = nodeTriple
        if N.goalTest():
            path = reconstructPath(N, parent_map)
            return len(path), path
        CLOSED.append(nodeTriple)
        for M in N.moveGen():
            if M not in [n for (n,_,_) in OPEN] and M not in [n for (n,_,_) in CLOSED]:
                parent_map[M] = N
```

```
            OPEN.append((M, N, M.h()))

    return -1, []
# ---------------- A* Search ----------------
def propagateImprovement(M, parent_map, g, f, CLOSED):
    for X in M.moveGen():
        k = M.stepCost(X)
        new_g = g[M] + k
        if new_g < g.get(X, float('inf')):
            parent_map[X] = M
            g[X] = new_g
            f[X] = g[X] + X.h()
            if X in CLOSED:
                propagateImprovement(X, parent_map, g, f, CLOSED)
def a_star(start, grid):
    n = len(grid)
    if grid[0][0] == 1 or grid[n-1][n-1] == 1:   # Start or goal blocked
        return -1, []
    OPEN = [start]
    CLOSED = []
    parent_map = {start: None}
    g = {start: 0}
    f = {start: g[start] + start.h()}
    while OPEN:
        N = min(OPEN, key=lambda s: f.get(s, float('inf')))
        OPEN.remove(N)
        if N.goalTest():
            path = reconstructPath(N, parent_map)
            return len(path), path
            CLOSED.append(N)
        for M in N.moveGen():
            k = N.stepCost(M)
```

```
            new_g = g[N] + k

            if new_g < g.get(M, float('inf')):
                parent_map[M] = N
                g[M] = new_g
                f[M] = g[M] + M.h()

                if M in CLOSED:
                    propagateImprovement(M, parent_map, g, f, CLOSED)
                elif M not in OPEN:
                    OPEN.append(M)
return -1, []
# Example inputs
def run_example(grid):
    start = State((0,0), grid, (len(grid)-1, len(grid)-1))
    bfs_len, bfs_path = best_first_search(start, grid)
    a_len, a_path = a_star(start, grid)
    print(f"Best First Search → Path length: {bfs_len}, Path: {bfs_path}")
    print(f"A* Search → Path length: {a_len}, Path: {a_path}")
if __name__ == "__main__":
    print("\nExample 1:")
    run_example([[0,1],[1,0]])
    print("\nExample 2:")
    run_example([[0,0,0],[1,1,0],[1,1,0]])
    print("\nExample 3:")
    run_example([[1,0,0],[1,1,0],[1,1,0]])
```

**Output:**

```
user@tejaswinichelluri:~$ python3 bestfirst.py

Example 1:
Best First Search → Path length: 2, Path: [(0, 0), (1, 1)]
A* Search → Path length: 2, Path: [(0, 0), (1, 1)]

Example 2:
Best First Search → Path length: 4, Path: [(0, 0), (0, 1), (1, 2), (2, 2)]
A* Search → Path length: 4, Path: [(0, 0), (0, 1), (1, 2), (2, 2)]

Example 3:
Best First Search → Path length: -1, Path: []
A* Search → Path length: -1, Path: []
user@tejaswinichelluri:~$
```

**Comparison between Best First Search and A*:**

Best First Search chooses paths that appear closest to the goal based only on the heuristic, so it is generally faster but does not guarantee the shortest path. In contrast, A* combines both the actual path cost and the heuristic, which makes it more reliable and ensures the optimal solution when one exists.

In the examples, both algorithms produced the same results when a clear optimal path was available. However, in larger or more complex grids, Best First Search may take a non-optimal route, while A* will always find the shortest valid path, though at the cost of exploring more nodes and using more memory.