CSE 404

Artificial Intelligence Lab

A* Search

Submitted By:	Submitted To:
Afrin Sultana Akhi (22101095)	Lecturer
Section**: B-2**	**Nahida Marzan**
Semester**: 3-2**	Department of Computer
Session**: Fall 2024**	Science and Engineering
	University of Asia Pacific

A Search for Robot Navigation with Dynamic Cost

1. Introduction

Robots are increasingly deployed in structured environments such as warehouses, factories, and logistics hubs. Efficient robot navigation is critical in such domains, especially when the environment contains **obstacles** and **cells with varying traversal costs** (e.g., slippery floors, carpets, uneven terrain).

The goal of this project is to design and implement an **intelligent robot navigation system** that uses the **A*** search algorithm with different heuristics to find the optimal path in a warehouse grid. The robot must consider both **terrain costs** and **movement directions** while avoiding obstacles.

2. Problem Statement

- The warehouse is modeled as an **m × n grid**.
- Some cells are **obstacles** (impassable).
- Each non-obstacle cell has a **terrain cost** (default = 1, or specified in input).
- The robot starts at a given cell and must reach the goal cell with **minimum total cost**.
- Movement rules:
    - **Horizontal/Vertical move** → cost = terrain cost of destination cell
    - **Diagonal move** → cost = 1.4 × terrain cost of destination cell

The project compares the effectiveness of **three heuristics** in A*:

1. **Manhattan Distance**
2. **Diagonal Distance**
3. **Euclidean Distance**

3. Objectives

1. Implement an intelligent navigation algorithm for grid-based environments with **dynamic terrain costs**.
2. Incorporate **8-directional movement** with adjusted diagonal costs.
3. Test and compare three different heuristics for A* search.
4. Measure **path cost, path length, explored nodes, and runtime**.
5. Provide a **comparison analysis** to determine which heuristic performs best.

4. Methodology

4.1 Input Format

- Grid dimensions: `m n`
- Obstacles: `k` followed by `k` lines of coordinates
- Terrain costs: `c` followed by `c` lines of coordinates with costs
- Starting cell: `startx starty`
- Goal cell: `goalx goaly`

4.2 Movement Cost Rules

- Horizontal/Vertical = terrain cost of destination cell
- Diagonal = 1.4 × terrain cost of destination cell

4.3 Algorithm – A* Search

- Maintain **OPEN list** (priority queue based on f = g + h)
- Maintain **CLOSED list** (expanded/explored nodes)
- For each neighbor:
  - Compute tentative cost `g’ = g(current) + move_cost`
  - Update if better path found
- Stop when goal node is expanded

4.4 Heuristics Used

1. **Manhattan Distance**:
   h(x,y)=∣goalx−x∣+∣goaly−y∣h(x,y) = |goal_x - x| + |goal_y - y|
2. **Diagonal Distance**:
   h(x,y)=max(|goalx−x|,|goaly−y|)h(x,y) = \max(|goal_x - x|, |goal_y - y|)
3. **Euclidean Distance**:
   h(x,y)=(goalx−x)2+(goaly−y)2h(x,y) = \sqrt{(goal_x - x)^2 + (goal_y - y)^2}

5. Implementation

The solution was implemented in **Python** using:

- `heapq` → priority queue for OPEN list
- `time` → runtime measurement
- `math` → Euclidean/Diagonal calculations
- Custom functions for parsing input, applying terrain costs, and reconstructing paths

The algorithm also records:

- Path taken
- Path cost
- Explored nodes (in order of expansion)
- Total explored count
- Runtime

2. Reading Input

```python
def read_input(filename):
    with open(filename, 'r') as f:
        lines = [line.strip() for line in f if line.strip()]
    m, n = map(int, lines[0].split())
    k = int(lines[1])
    obstacles = set(tuple(map(int, lines[i+2].split())) for i in range(k))
    idx = 2 + k
    c = int(lines[idx])
    terrain = {}
    for i in range(c):
        x, y, cost = lines[idx+1+i].split()
        terrain[(int(x), int(y))] = float(cost)
    start = tuple(map(int, lines[idx+1+c].split()))
    goal = tuple(map(int, lines[idx+2+c].split()))
    return m, n, obstacles, terrain, start, goal
```

- Reads input file (like `input.txt`).
- Extracts:
  - `m, n`: grid dimensions
  - `obstacles`: cells that cannot be crossed
  - `terrain`: dictionary where `(x,y)` has a custom movement cost

- `start`, `goal`: start and target positions

3. Neighbor Generation

```
def get_neighbors(pos, m, n, obstacles):
    x, y = pos
    moves = [(-1,0),(1,0),(0,-1),(0,1),(-1,-1),(-1,1),(1,-1),(1,1)]
    neighbors = []
    for dx, dy in moves:
        nx, ny = x+dx, y+dy
        if 0 <= nx < m and 0 <= ny < n and (nx, ny) not in obstacles:
            neighbors.append((nx, ny, dx, dy))
    return neighbors
```

- Returns **all valid moves** (8 directions).
- Ignores cells outside the grid or that are obstacles.
- Keeps track of `(nx, ny)` and movement `(dx, dy)`.

4. Terrain & Move Cost

```
def move_cost(terrain, cell, dx, dy):
    base = terrain_cost(terrain, cell)
      if abs(dx) + abs(dy) == 2:
        return 1.4 * base
    return base
```

- Straight move → cost = base.
- Diagonal move → cost ≈ √2 times base (approximated as `1.4`).

5. Heuristics

```python
def manhattan(a, b):
    return abs(a[0]-b[0]) + abs(a[1]-b[1])

def diagonal(a, b):
    return max(abs(a[0]-b[0]), abs(a[1]-b[1]))

def euclidean(a, b):
    return math.hypot(a[0]-b[0], a[1]-b[1])
```

- **Manhattan**: grid distance with only horizontal/vertical moves.
- **Diagonal**: minimum steps when diagonal moves allowed.
- **Euclidean**: straight-line (as the crow flies).

6. A* Search

```python
def astar(m, n, obstacles, terrain, start, goal, heuristic_fn):
    open_list = []
    heapq.heappush(open_list, (0, start))
    came_from = {}
    g_score = {start: 0}
    explored = []
    closed_set = set()
    while open_list:
        _, current = heapq.heappop(open_list)
        if current in closed_set:
            continue
        explored.append(current)
        closed_set.add(current)
        if current == goal:
            break
        for neighbor, dx, dy in [(nb[:2], nb[2], nb[3]) for nb in get_neighbors(current, m, n, obstacles)]:
            tentative_g = g_score[current] + move_cost(terrain, neighbor, dx, dy)
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic_fn(neighbor, goal)
                heapq.heappush(open_list, (f_score, neighbor))
    path = []
    node = goal
    if node in came_from or node == start:
        while node != start:
            path.append(node)
            node = came_from[node]
        path.append(start)
        path.reverse()
    else:
        path = []
    cost = sum(
        move_cost(terrain, path[i], path[i][0]-path[i-1][0], path[i][1]-path[i-1][1])
        for i in range(1, len(path))
    ) if path else float('inf')
```

- **Open list** = frontier nodes (priority queue ordered by `f = g + h`).
- **Closed set** = already processed nodes.
  At each step:
  1. Pop node with lowest `f`.
  2. If it's the goal → stop.
  3. Expand neighbors → calculate new cost.
  4. If better path found → update parent and push to queue.

7. Running All Heuristics

```python
def run_all(filename):
    m, n, obstacles, terrain, start, goal = read_input(filename)
    heuristics = [
        ("Manhattan", manhattan),
        ("Diagonal", diagonal),
        ("Euclidean", euclidean)
    ]
    results = []
    for name, hfn in heuristics:
        t0 = time.time()
        path, cost, explored = astar(m, n, obstacles, terrain, start, goal, hfn)
        runtime = time.time() - t0
        print(f"--- {name} Heuristic ---")
        print(f"Path: {path}")
        print(f"Path Cost: {round(cost, 4)}")
        print(f"Explored Nodes: {explored}")
        print(f"Total Explored: {len(explored)}")
        print(f"Runtime: {runtime:.6f} seconds\n")
        results.append((name, round(cost,4), len(path), len(explored), float(f"{runtime:.6f}")))
    print("Heuristic\tPath Cost\tPath Length\tTotal Explored Nodes\tRuntime (s)")
    for r in results:
        print(f"{r[0]}\t{r[1]}    \t\t{r[2]}\t\t{r[3]}\t\t\t{r[4]}")
```

- Runs A* three times with different heuristics.
- Prints path, cost, explored nodes, runtime.
- Shows a comparison table.

---

8. Main Entry

```python
if __name__ == "__main__":
    run_all("input.txt")
```

- Program starts by reading `input.txt` and running all heuristics.

6. Sample Input & Output

Input

5 5
2
1 1
3 3
3
0 1 2
1 2 3
2 2 5
0 0
4 4

Output

Manhattan Heuristic

- Path: [(0,0), (1,0), (2,1), (3,2), (4,3), (4,4)]
- Path Cost: **6.2**
- Explored Nodes: [(0,0), (1,0), (2,1), (3,2), (4,3), (4,4)]
- Total Explored: 6
- Runtime: 0.000116 s

Diagonal Heuristic

- Path: [(0,0), (1,0), (2,1), (3,2), (4,3), (4,4)]
- Path Cost: **6.2**
- Explored Nodes: [(0,0), (1,0), (2,1), (3,2), (0,1), (2,0), (4,3), (4,4)]
- Total Explored: 8
- Runtime: 0.000092 s

Euclidean Heuristic

- Path: [(0,0), (1,0), (2,1), (3,2), (4,3), (4,4)]
- Path Cost: **6.2**
- Explored Nodes: [(0,0), (1,0), (2,1), (3,2), (4,3), (4,4)]
- Total Explored: 6
- Runtime: 0.000068 s

7. Comparison Table

Heuristic	Path Cost	Path Length	Total Explored Nodes	Runtime (s)
Manhattan	6.2	6	6	0.000061
Diagonal	6.2	6	8	0.000062
Euclidean	6.2	6	6	0.000058

8. Analysis & Discussion

- All three heuristics produced the **same optimal path** with identical cost (6.2).
- **Euclidean** heuristic explored fewer nodes than Diagonal, and had the **fastest runtime**.
- **Manhattan** was admissible but slightly less efficient due to overestimating cost in diagonal movement scenarios.
- **Diagonal heuristic** is tailored for 8-directional movement but expanded more nodes in this case.
- For larger, more complex grids with many obstacles and varied terrain, **Euclidean heuristic** is expected to consistently balance efficiency and accuracy.

9. Conclusion

This project successfully demonstrated **A*** search for robot navigation with **dynamic terrain costs**. The robot was able to compute the minimum-cost path considering **both terrain and diagonal movement rules**.

The comparison revealed that while all heuristics guarantee optimality, **Euclidean performed best overall** in terms of runtime and node expansions. This makes it a strong candidate for real-world robotic navigation where speed and efficiency are critical.

10. Future Work

- Extend to **dynamic environments** (moving obstacles).
- Implement **real-time replanning** (D* or Lifelong Planning A*).
- Introduce **energy constraints** for the robot.
- Apply the algorithm on an actual robot in a warehouse setting.