CS 272: Artificial Intelligence

Assignment No. 01

Informed Search Strategies

Hamza Raheem
CMS ID: 512131
BS Data Science

Abstract

This report explores informed search strategies for path planning in dynamic city grid environments, focusing on autonomous robot navigation. The project implements and evaluates Greedy Best-First Search (GBFS) extended to multi-goal scenarios, Weighted A* with tunable heuristic weighting (α), and Bidirectional A* for efficient large-scale searches. These algorithms leverage heuristics like Manhattan, Euclidean, and a non-admissible variant to guide exploration in a simulated 2D grid with obstacles mimicking urban layouts (e.g., roads and blocks). The CityGrid environment supports 4-connected movement, random obstacle generation at 70-85% density, and dynamic modifications.

1. Introduction

Search algorithms are fundamental to Artificial Intelligence, enabling agents to navigate complex environments and make informed decisions. Among them, informed (heuristic) search strategies utilize problem-specific knowledge to guide the search process more efficiently toward the goal. This assignment focuses on implementing and evaluating several informed search techniques to solve pathfinding problems in a two-dimensional grid environment.

In real-world applications such as autonomous delivery robots operating in urban environments (e.g., Amazon Scout or Starship Technologies bots), efficient path planning is crucial. These systems must navigate complex, obstacle-dense areas while visiting multiple destinations in an optimal sequence. Traditional uninformed searches like Breadth-First Search (BFS), although exhaustive, are computationally expensive and scale poorly in large grids. In contrast, informed strategies incorporate heuristics to prioritize promising paths, significantly reducing exploration time and computational cost.

Building on this foundation, the report focuses on multi-goal pathfinding, where a robot must visit several points (e.g., delivery stops) in sequence. The Greedy Best-First Search (GBFS) algorithm extends heuristic-driven exploration to handle multiple goals efficiently, while Weighted A* and Bidirectional A* — both variants of the classical A* algorithm — introduce tunable optimality and bidirectional search efficiency. These methods are especially relevant for dynamic urban scenarios with temporary obstacles, such as construction zones or road closures.

The study evaluates these algorithms using a simulated CityGrid environment, analyzing key performance metrics such as nodes expanded, total path cost, and runtime. Through this experimental comparison, the report highlights the trade-offs between optimality, speed, and computational efficiency across different informed search strategies.

2. Environment

The CityGrid class simulates a 2D urban map with free cells (0) and obstacles (1), using 4-connected movement (North, South, East, West) for grid-aligned realism, avoiding diagonal "corner-cutting." Obstacles are generated with 70-85% density in a structured pattern—main "roads" every 3-10 cells, plus random noise—for city-like navigability. The grid ensures start (0,0) and goals are reachable via BFS validation. Dynamic updates allow adding/removing obstacles mid-search.

Code: CityGrid

```
from collections import deque
Import  random
# CityGrid Environment
# This class represents a simulated city map for an autonomous robot.
# Each cell can be free (0) or obstacle (1).
# The grid supports:
#   - 8-connected movement (diagonals allowed)
#   - Random obstacle generation (city-like layout)
#   - Dynamic updates (add/remove obstacles)
#   - Movement costs (diagonals slightly higher)
class CityGrid:
    def __init__(self, width, height, obstacle_density=.70, seed=None, goals=None):
```

```
    """
    Initialize the city grid environment.
    :param width: number of columns in the grid
    :param height: number of rows in the grid
    :param obstacle_density: approximate proportion of obstacle cells, 0.70 default to      match real world standard.
    :param seed: random seed for reproducibility
    :param goals: initialize grid with goals at initial stage.
    """
    self.width = width
    self.height = height
    self.grid = [[0 for _ in range(width)] for _ in range(height)]
    self.random = random.Random(seed)
    self.start = (0, 0)
    self.goals = goals if goals else [(height - 1, width - 1)]
    self.generate_obstacles(obstacle_density)
# Why choose 4-connected instead of 8-connected?
# In many real-world navigation setups (like warehouse robots,
# street-following delivery bots, or indoor robots),
# movement is restricted to four cardinal directions —
# north, south, east, west.
# This ensures motion adheres to "grid-aligned" movement,
# and prevents corner-cutting through obstacles.
# Each move has a uniform cost = 1.
def get_neighbors(self, node):
    """Return valid 4-connected neighbors for a given cell (y, x)."""
    y, x = node
    directions = [
        (-1, 0),  # North
        (1, 0),   # South
        (0, -1),  # West
        (0, 1)    # East
    ]
    neighbors = []
    for dy, dx in directions:
        ny, nx = y + dy, x + dx
        if 0 <= ny < self.height and 0 <= nx < self.width:
            if self.grid[ny][nx] == 0:  # free cell
                neighbors.append(((ny, nx), 1))  # uniform cost
    return neighbors
def generate_obstacles(self, density=0.70):
    """
    Generate obstacles intelligently to simulate a city environment.
    Roads and blocks alternate in a grid pattern with random noise.
    :param density: approximate fraction of blocked cells
    """
    max_attempts = 100
    attempts = 0
    while attempts < max_attempts:
        attempts += 1
        # Start with an empty grid
        for y in range(self.height):
            for x in range(self.width):
                self.grid[y][x] = 0
        # Generate obstacle layout with dynamic road spacing
        for y in range(self.height):
            for x in range(self.width):
                # Dynamically determine road spacing based on grid size
                road_spacing = max(3, min(self.width, self.height) // 10)
                # Structured city pattern: main roads every few cells
                if (y % road_spacing == 0 or x % road_spacing == 0) and not (y == 0 and x == 0):
                    # Roads are mostly open with a bit of randomness
```

```python
                    self.grid[y][x] = 0 if self.random.random() > 0.15 else 1
                else:
                    # Inner blocks get obstacles based on density
                    if self.random.random() < density:
                        self.grid[y][x] = 1
        # Randomly clear some cells to add alternate routes
        for _ in range(int(self.width * self.height * 0.05)):
            ry = self.random.randint(0, self.height - 1)
            rx = self.random.randint(0, self.width - 1)
            self.grid[ry][rx] = 0
        # Ensure start and goals are open
        self.grid[self.start[0]][self.start[1]] = 0
        for gy, gx in self.goals:
            self.grid[gy][gx] = 0
        # ✅ Check if all goals are reachable from the start
        if all(self.is_reachable(self.start, g) for g in self.goals):
            return  # Success — keep this layout!
    print("   Warning: Failed to generate a valid reachable map after several attempts.")
def is_valid(self, y, x):
    """Check if a cell is inside the grid and not an obstacle."""
    return 0 <= y < self.height and 0 <= x < self.width and self.grid[y][x] == 0
def display(self, start=None, goals=None):
    """
    Print a simple ASCII map of the grid.
    'S' = Start, 'G' = Goal, '█' = obstacle, '.' = free cell.
    """
    for y in range(self.height):
        for x in range(self.width):
            if start and (y, x) == start:
                print("S", end=" ")
            elif goals and (y, x) in goals:
                print("G", end=" ")
            elif self.grid[y][x] == 1:
                print("█", end=" ")
            else:
                print(".", end=" ")
        print()
    print()
def display_path(self, path, start, goal):
    """
    Displays the grid in a simple text format.
    Uses:
      S - start
      G - goal
      █ - obstacle
      . - empty road
      o - path
    """
    path_set = set(path) if path else set()
    print(f"\nDisplaying path from start ({start[0]}, {start[1]}) to goal ({goal[0]}, {goal[1]}):")
    for y in range(self.height):
        for x in range(self.width):
            if (y, x) == start:
                print("S", end=" ")
            elif (y, x) == goal:
                print("G", end=" ")
            elif (y, x) in path_set:
                print("o", end=" ")
            elif self.grid[y][x] == 1:
                print("█", end=" ")
            else:
```

```
                print(".", end=" ")
            print()
        print()
    def display_multi_goal_path(self, path_segments, start, goals):
        """
        Displays the grid showing paths to multiple goals.
        Each segment of the path is shown in sequence.
        Uses:
          S - start
          A/B/C/... - goals
          █ - obstacle
          . - empty road
          o - path to each goal (different symbols per goal)
        """
        # Assign unique symbols for each goal path
        symbols = ['o', '*', '+', 'x', '~', '^', 'Δ']
        goal_marks = [chr(65 + i) for i in range(len(goals))]  # A, B, C, etc.
        # Convert goals list to a dict for labeling
        goal_labels = {tuple(goals[i]): goal_marks[i] for i in range(len(goals))}
        # Combine all path segments
        full_path = set()
        for segment in path_segments:
            full_path.update(segment)
        print("\nDisplaying Multi-Goal GBFS Path:")
        print(f"Start: {start}")
        print(f"Goals: {', '.join([f'{goal_marks[i]}{goals[i]}' for i in range(len(goals))])}\n")
        for y in range(self.height):
            for x in range(self.width):
                if (y, x) == start:
                    print("S", end=" ")
                elif (y, x) in goal_labels:
                    print(goal_labels[(y, x)], end=" ")
                elif (y, x) in full_path:
                    # Choose symbol based on which goal segment it belongs to
                    for i, segment in enumerate(path_segments):
                        if (y, x) in segment:
                            print(symbols[i % len(symbols)], end=" ")
                            break
                elif self.grid[y][x] == 1:
                    print("█", end=" ")
                else:
                    print(".", end=" ")
            print()
        print()
    def modify_obstacle(self, cell, add=True):
        """
        Dynamically add or remove an obstacle.
        :param cell: tuple (y, x)
        :param add: if True, add an obstacle; else remove it
        """
        y, x = cell
        if 0 <= y < self.height and 0 <= x < self.width:
            self.grid[y][x] = 1 if add else 0
    def is_reachable(self, start, goal):
        """
        Use BFS to check if there is any valid path between start and goal.
        Ensures generated maps are navigable.
        """
        if not self.is_valid(*start) or not self.is_valid(*goal):
            return False
        queue = deque([start])
```

```
    visited = {start}
    while queue:
        y, x = queue.popleft()
        if (y, x) == goal:
            return True
        for (ny, nx), _ in self.get_neighbors((y, x)):
            if (ny, nx) not in visited:
                visited.add((ny, nx))
                queue.append((ny, nx))
    return False
```

Sample Generation Output:

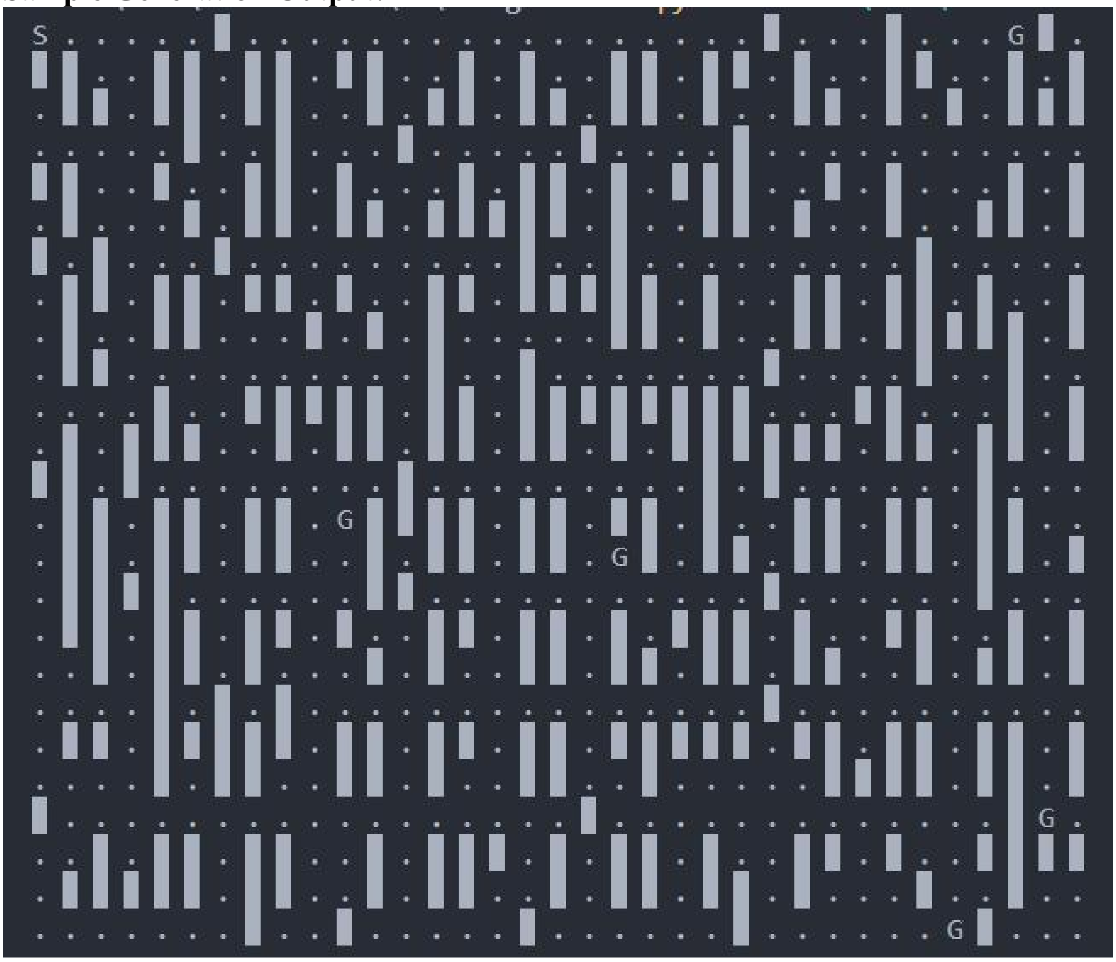

Figure 1: Sample Generated Static Grid (25x35, 85% Density)

3. Theoretical Background

Informed search strategies differ from uninformed ones by incorporating heuristic functions that estimate the cost to reach the goal from a given node. This estimation helps prioritize paths that are more promising, resulting in faster and more efficient searches.
The **A*** algorithm forms the foundation for many informed search techniques. It uses the evaluation function:

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

where $g(n)$ represents the actual cost from the start node to n, and $h(n)$ is the heuristic estimate of the cost from n to the goal.
In **Weighted A***, a weighting factor α (alpha) is introduced to control the influence of the heuristic:

$$f(n) = g(n) + \alpha \times h(n)$$

When α > 1, the algorithm becomes greedier, favoring paths that appear closer to the goal based on the heuristic, thereby improving speed but possibly reducing optimality.
Greedy Best-First Search (GBFS), on the other hand, relies solely on the heuristic value:

$$f(n) = h(n)$$

This means GBFS expands the node that seems closest to the goal, often leading to fast results but sometimes sub-optimal paths.
Bidirectional A*runs two simultaneous A* searches: one forward from the start and one backward from the goal. When the two frontiers meet, the search terminates. This drastically reduces the search space and is especially efficient in large environments.

4. Implementation Overview

The project was implemented in Python with a modular structure, ensuring clarity and maintainability. Each module handles a specific aspect of the problem, as described below:
- **grid.py**: Defines the environment, start/goal locations, and obstacle handling.
- **heuristics.py**: Contains heuristic functions such as Manhattan, Euclidean, and Non-Admissible.
- **search.py**: Implements the core search algorithms (A*, Weighted A*, Bidirectional A*, GBFS).
- **visualize.py / visualize_enhanced.py**: Handles visual representations of exploration and path-finding progress.

GitHub Link: https://github.com/hamzaraheem06/Informed-Search-Strategies

5. Task 1: Multi-Goal Greedy Best-First Search

The objective of this task is to extend the Greedy Best-First Search (GBFS) algorithm to handle multiple goals. The robot must visit all destinations (for example, A, B, and C) efficiently while minimizing heuristic estimates. The heuristic is recalculated dynamically based on the nearest unvisited goal.

Multi-goal Greedy Best First Search Implementation:

```
def greedy_best_first_search(start, goal, grid, heuristic):
    """
    Greedy Best-First Search (GBFS)
    Uses only heuristic to guide the search.
    Returns:
        path, nodes_expanded, frontier_history
    """
    height, width = len(grid), len(grid[0])
    directions = [(-1,0), (1,0), (0,-1), (0,1)]

    def in_bounds(y,x):
        return 0 <= y < height and 0 <= x < width
    open_set = []
    heapq.heappush(open_set, (heuristic(start, goal), start))
    came_from = {}
    visited = set()
    nodes_expanded = 0
    # To store frontier snapshots for visualization
    frontier_history = []
    while open_set:
        _, current = heapq.heappop(open_set)
        if current in visited:
            continue
        visited.add(current)
        nodes_expanded += 1
        # Record current frontier (for visualization)
        frontier_snapshot = [node for (_, node) in open_set]
```

```
        frontier_history.append(frontier_snapshot)
        if current == goal:
            # reconstruct path
            path = [current]
            while current in came_from:
                current = came_from[current]
                path.append(current)
            path.reverse()
            return path, nodes_expanded, frontier_history
                    cy, cx = current
        for dy, dx in directions:
            ny, nx = cy + dy, cx + dx
            if not in_bounds(ny, nx) or grid[ny][nx] == 1:
                continue
            neighbor = (ny, nx)
            if neighbor not in visited:
                came_from[neighbor] = current
                heapq.heappush(open_set, (heuristic(neighbor, goal), neighbor))
    return None, nodes_expanded, frontier_history
def multi_goal_gbfs(start, goals, grid, heuristic):
    """
    Multi-Goal Greedy Best-First Search
    ----------------------------------
    Visit all goals one by one using GBFS.
    Always move to the nearest unvisited goal.
    """
    current_start = start
    remaining_goals = goals[:]
    full_path = []
    total_nodes = 0
    path_segments = []
    all_frontiers = []
    while remaining_goals:
        # Find the nearest goal (by heuristic)
        nearest_goal = min(remaining_goals, key=lambda g: heuristic(current_start, g))
        # Run GBFS to that goal
        path, nodes, frontier_history = greedy_best_first_search(current_start, nearest_goal, grid, heuristic)
        if path is None:
            print(f"❌ No path found to goal {nearest_goal}")
            break
        # Append path (avoid duplicating the starting cell)
        path_segments.append(path)
        if full_path:
            full_path.extend(path[1:])
        else:
            full_path.extend(path)
        total_nodes += nodes
        all_frontiers.extend(frontier_history)
        # Move to next goal
        current_start = nearest_goal
        remaining_goals.remove(nearest_goal)
    return path_segments, full_path, total_nodes, all_frontiers
```

GBFS rapidly explores towards the nearest goal but may choose a misleading path when the heuristic is not admissible. The visualization highlights how the algorithm’s frontier expands aggressively toward goal estimates.

Multi GBFS Path:

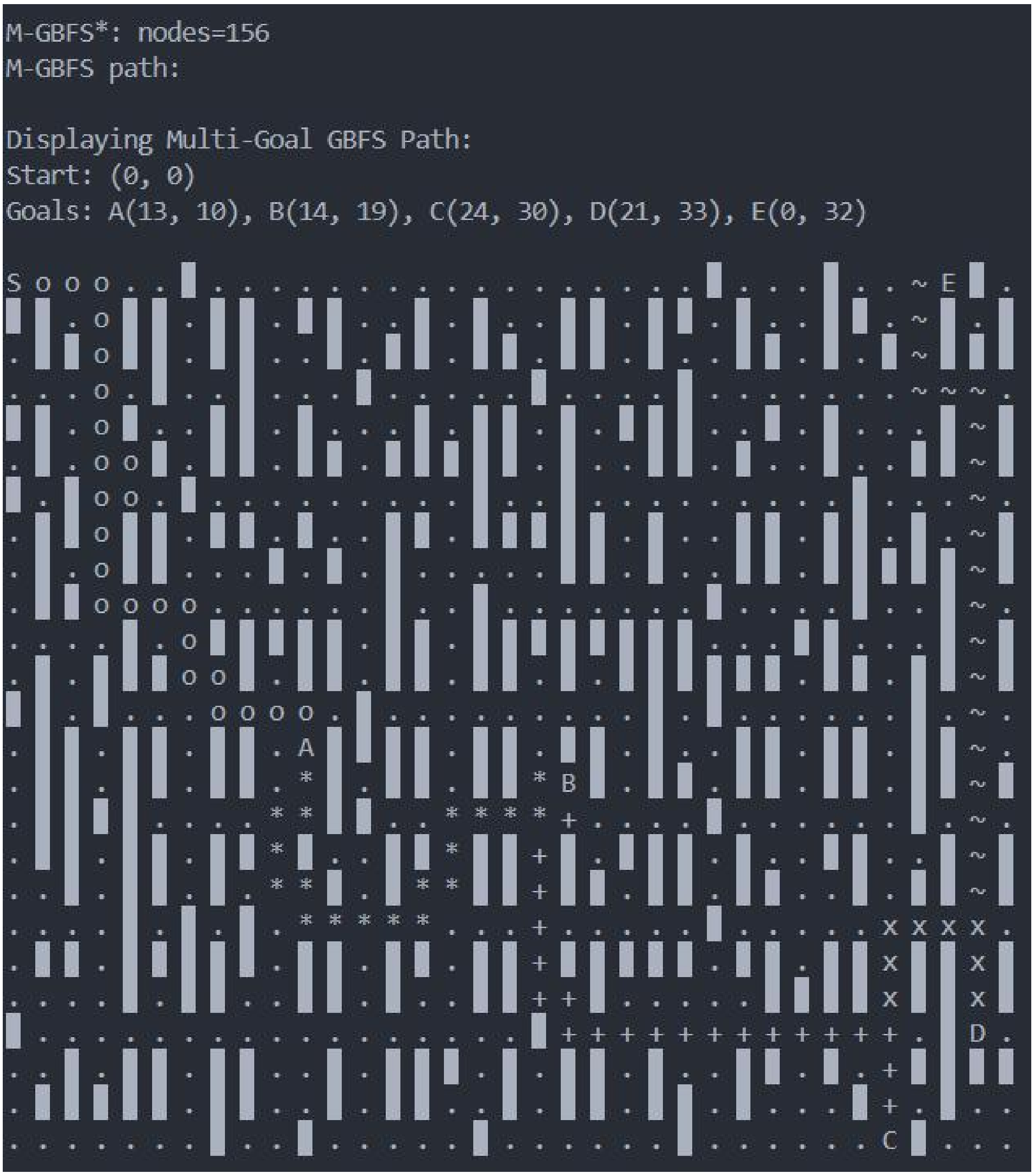

Figure 2: Multi-Goal GBFS Path Visualization

6. Task 2: Weighted A* and Bidirectional A*

In this task, two variants of A* are explored: Weighted A* and Bidirectional A*. Weighted A* applies a tunable parameter α that scales the heuristic's impact, making the search greedier as α increases. Bidirectional A* executes simultaneous forward and backward searches, resulting in faster convergence for large grids.

Weighted A* Search Implementation:

```
def weighted_a_star(start, goal, grid, heuristic, alpha=1.0):
    """
    Weighted A* Search Algorithm
    start: (y, x)
    goal: (y, x)
    grid: 2D list
    heuristic: function(node, goal)
```

```
    alpha: weight for heuristic (α > 1 = greedier)
    Returns:
        path, total_cost, nodes_expanded
    """
    height, width = len(grid), len(grid[0])
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    def in_bounds(y, x):
        return 0 <= y < height and 0 <= x < width
    def cost(a, b):
        return 1  # 4-connected
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {start: 0}
    nodes_expanded = 0
    while open_set:
        f, current = heapq.heappop(open_set)
        nodes_expanded += 1
        if current == goal:
            # reconstruct path
            path = [current]
            while current in came_from:
                current = came_from[current]
                path.append(current)
            path.reverse()
            return path, g_score[goal], nodes_expanded
        cy, cx = current
        for dy, dx in directions:
            ny, nx = cy + dy, cx + dx
            if not in_bounds(ny, nx) or grid[ny][nx] == 1:
                continue
            neighbor = (ny, nx)
            tentative_g = g_score[current] + cost(current, neighbor)
            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + alpha * heuristic(neighbor, goal)
                heapq.heappush(open_set, (f_score, neighbor))
    return None, float('inf'), nodes_expanded
```

Bidirectional A* Search Implementation:

```
def bidirectional_a_star(start, goal, grid, heuristic):
    """
    Bidirectional A* Search Algorithm
    --------------------------------
    Runs A* from both start and goal until frontiers meet.

    Returns:
        path, total_cost, nodes_expanded
    """
    height, width = len(grid), len(grid[0])
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    def in_bounds(y, x):
        return 0 <= y < height and 0 <= x < width
    def cost(a, b):
        return 1
    # open sets for both searches
    open_fwd = [(0, start)]
    open_bwd = [(0, goal)]
    # cost and parents for both directions
    g_fwd = {start: 0}
```

```python
g_bwd = {goal: 0}
came_from_fwd = {}
came_from_bwd = {}
visited_fwd = set()
visited_bwd = set()
nodes_expanded = 0
meeting_node = None
while open_fwd and open_bwd:
    # Expand from start side
    _, current_fwd = heapq.heappop(open_fwd)
    visited_fwd.add(current_fwd)
    nodes_expanded += 1
    if current_fwd in visited_bwd:
        meeting_node = current_fwd
        break
    cy, cx = current_fwd
    for dy, dx in directions:
        ny, nx = cy + dy, cx + dx
        if not in_bounds(ny, nx) or grid[ny][nx] == 1:
            continue
        neighbor = (ny, nx)
        tentative_g = g_fwd[current_fwd] + cost(current_fwd, neighbor)
        if tentative_g < g_fwd.get(neighbor, float('inf')):
            came_from_fwd[neighbor] = current_fwd
            g_fwd[neighbor] = tentative_g
            f_score = tentative_g + heuristic(neighbor, goal)
            heapq.heappush(open_fwd, (f_score, neighbor))
    # Expand from goal side
    _, current_bwd = heapq.heappop(open_bwd)
    visited_bwd.add(current_bwd)
    nodes_expanded += 1
    if current_bwd in visited_fwd:
        meeting_node = current_bwd
        break
    cy, cx = current_bwd
    for dy, dx in directions:
        ny, nx = cy + dy, cx + dx
        if not in_bounds(ny, nx) or grid[ny][nx] == 1:
            continue
        neighbor = (ny, nx)
        tentative_g = g_bwd[current_bwd] + cost(current_bwd, neighbor)
        if tentative_g < g_bwd.get(neighbor, float('inf')):
            came_from_bwd[neighbor] = current_bwd
            g_bwd[neighbor] = tentative_g
            f_score = tentative_g + heuristic(neighbor, start)
            heapq.heappush(open_bwd, (f_score, neighbor))
if meeting_node is None:
    return None, float('inf'), nodes_expanded
# reconstruct path
path_fwd = []
node = meeting_node
while node in came_from_fwd:
    path_fwd.append(node)
    node = came_from_fwd[node]
path_fwd.append(start)
path_fwd.reverse()
path_bwd = []
node = meeting_node
while node in came_from_bwd:
    path_bwd.append(node)
    node = came_from_bwd[node]
```

```
# skip meeting node to avoid duplication
path_bwd = path_bwd[1:] if path_bwd else []
full_path = path_fwd + path_bwd
total_cost = g_fwd[meeting_node] + g_bwd[meeting_node]
return full_path, total_cost, nodes_expanded
```

Weighted A* Search Path:

Figure 3:Weighted A* Path Visualization

Bidirectional A* Search Path:

Figure 4: Bidirectional A* GBFS Path Visualization

Comparison of Bidirectional & Weighted A*:
Static Environment:

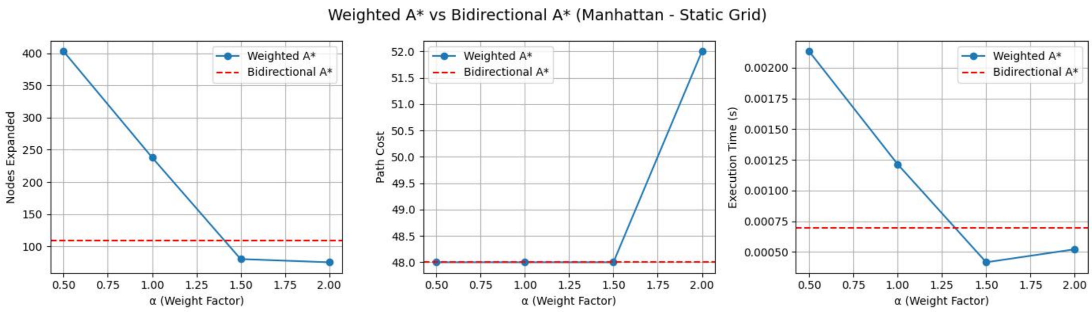

Figure 5: Weighted vs Bidirectional A* on a static environment

Dynamic Environment:

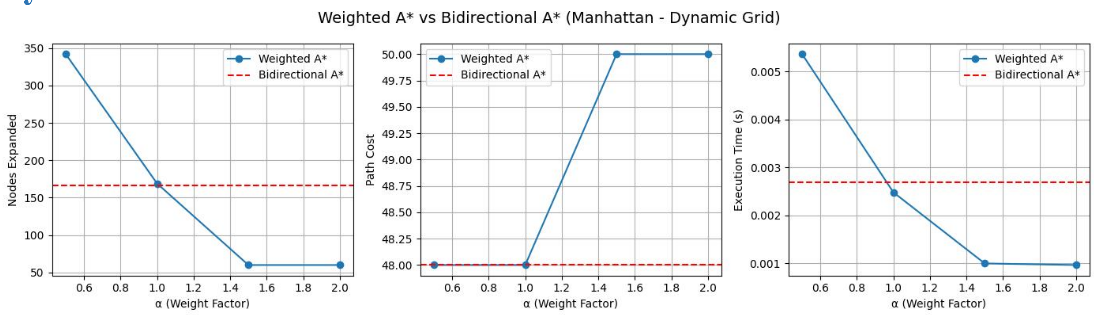

Figure 6: Figure 5: Weighted vs Bidirectional A* on a dynamic environment

7. Task 3: Experimental Evaluation

The algorithms were tested across varying heuristic functions (Manhattan, Euclidean, and Non-Admissible) and multiple α values (e.g., 0.5, 1.0, 1.5).

Heuritic Functions implementation:

```
import math

def manhattan(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
def euclidean(a, b):
    return math.sqrt((a[0] - b[0])**2 + (a[1] - b[1])**2)
def non_admissible(a, b):
    return 1.5 * manhattan(a, b)
```

The results include the number of nodes expanded, path cost, and execution time.

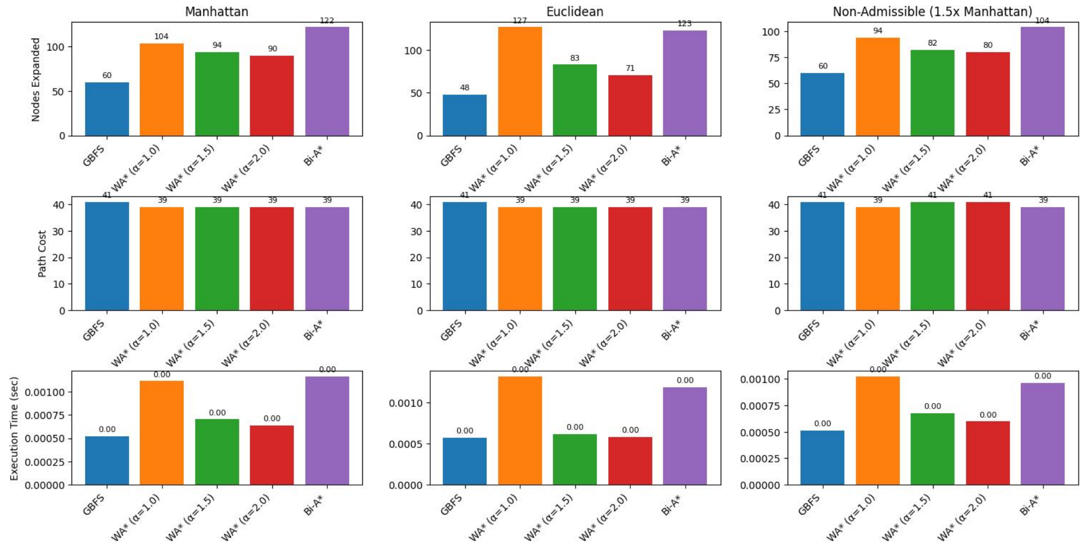

Figure 7: Bar plots for factors across different heuristics

During testing, some runs showed execution time close to zero. This occurs when the grid size is small, the path is direct, or the environment and heuristic align perfectly, resulting in negligible computational overhead.

As α increases, the Weighted A* algorithm leans towards GBFS-like behavior—fast but less optimal. Conversely, lower α values yield more optimal paths at the cost of higher computation. Bidirectional A* consistently showed reduced exploration, confirming its advantage for symmetric grids.

Code for the above plot:

```
import time
import matplotlib.pyplot as plt
```

```python
from grid import CityGrid
from search import greedy_best_first_search, weighted_a_star, bidirectional_a_star
from heuristics import manhattan, euclidean, non_admissible

def main():
    # Set up the grid environment (same as main.py for consistency)
    goals = [(13, 10), (14, 19), (24, 30), (21, 33), (0, 32)]
    grid_env = CityGrid(width=35, height=25, seed=42, obstacle_density=0.85, goals=goals)
    start = grid_env.start
    # Choose a single goal for analysis (using the second goal for consistency)
    goal = goals[1]  # (14, 19)
    # Define heuristics to test (two admissible, one non-admissible)
    heuristics = [
        ("Manhattan", manhattan),
        ("Euclidean", euclidean),
        ("Non-Admissible (1.5x Manhattan)", non_admissible)
    ]
    # Define algorithms and parameters (including multiple weighted A* alphas)
    algorithms = [
        ("GBFS", greedy_best_first_search, None),
        ("WA* (α=1.0)", weighted_a_star, 1.0),
        ("WA* (α=1.5)", weighted_a_star, 1.5),
        ("WA* (α=2.0)", weighted_a_star, 2.0),
        ("Bi-A*", bidirectional_a_star, None)
    ]
    # Prepare a structure to store results
    # results[algorithm_name][heuristic_name] = {"nodes": ..., "cost": ..., "time": ...}
    results = {alg[0]: {} for alg in algorithms}
    # Run each algorithm with each heuristic and record metrics
    for alg_name, alg_func, alpha in algorithms:
        for heur_name, heur_func in heuristics:
            start_time = time.perf_counter()
            if alg_name == "GBFS":
                path, nodes, _ = alg_func(start, goal, grid_env.grid, heur_func)
                cost = len(path) - 1  # cost = number of moves (4-connected, cost=1 each)
            elif alg_name.startswith("WA*"):
                path, cost, nodes = alg_func(start, goal, grid_env.grid, heur_func, alpha=alpha)
            elif alg_name == "Bi-A*":
                path, cost, nodes = alg_func(start, goal, grid_env.grid, heur_func)
            else:
                path, cost, nodes = None, float('inf'), 0
            end_time = time.perf_counter()
            elapsed = end_time - start_time
            results[alg_name][heur_name] = {"nodes": nodes, "cost": cost, "time": elapsed}
            print(f"{alg_name}, {heur_name}: cost = {cost}, nodes = {nodes}, time = {elapsed:.6f} sec")
    # Visualization: subplots for each metric (rows) and heuristic (columns)
    metrics = ["nodes", "cost", "time"]
    metric_labels = {"nodes": "Nodes Expanded", "cost": "Path Cost", "time": "Execution Time (sec)"}
    fig, axes = plt.subplots(len(metrics), len(heuristics), figsize=(12, 9))
    # Define a consistent algorithm order and colors
    alg_order = [alg[0] for alg in algorithms]
    colors = plt.get_cmap('tab10').colors
    alg_colors = {alg: colors[i % len(colors)] for i, (alg, _, _) in enumerate(algorithms)}
    # Fill in subplots
    for i, metric in enumerate(metrics):
        for j, (heur_name, _) in enumerate(heuristics):
            ax = axes[i][j]
            # Collect metric values for each algorithm
            values = [results[alg_name][heur_name][metric] for alg_name in alg_order]
            # Create bar chart for this subplot
            bars = ax.bar(alg_order, values, color=[alg_colors[name] for name in alg_order])
```

```
        # Set column title for heuristic (top row)
        if i == 0:
            ax.set_title(heur_name)
        # Set row label for metric (first column)
        if j == 0:
            ax.set_ylabel(metric_labels[metric])
        # Rotate x-axis labels for readability
        ax.set_xticks(range(len(alg_order)))
        ax.set_xticklabels(alg_order, rotation=45, ha='right')
        # Annotate bars with their value
        for bar in bars:
            height = bar.get_height()
            label = f"{height:.2f}" if metric == "time" else f"{int(height)}"
            ax.annotate(label,
                        xy=(bar.get_x() + bar.get_width() / 2, height),
                        xytext=(0, 3), textcoords="offset points",
                        ha='center', va='bottom', fontsize=8)
plt.tight_layout()
plt.show()
```

8. Discussion

Each algorithm exhibits unique characteristics:

- Greedy Best-First Search: Fastest but not guaranteed optimal; relies heavily on heuristic accuracy.
- Weighted A*: Provides a tunable balance between speed and optimality using α.
- Bidirectional A*: Efficient in large or symmetric spaces; requires careful handling of merging conditions.

The experiments demonstrate that heuristic design greatly impacts performance. The Manhattan heuristic performs well in grid-based motion with four directions, while Euclidean heuristics perform better for diagonal movement.

9. Conclusion and Future Work

This assignment provided hands-on experience with advanced informed search algorithms. The comparative evaluation revealed the importance of heuristic quality and weighting factors in determining search performance. Future improvements could include implementing dynamic weighting schemes or integrating learning-based heuristics for adaptive search behavior in complex environments.

10. References

[1] Russell, S., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach (4th Edition).
[2] Course Lecture Notes: CS 272 – Artificial Intelligence, NUST.
[3] GFG Articles on A*, GBFS, and Bidirectional Search.
[4] Medium Blog Posts on Heuristic Optimization and Pathfinding.