COSC 4368
Professor Sen Lin
Mustafa Sahin

# Problem Set 1 Report

## Task 1: Solving the City Road Network Problem Using Graph Algorithms

### Objective

The goal was to find the shortest paths in a city road network. Peter Parker, a computer science intern at Stark Industries, needed to optimize delivery routes by determining the shortest paths from a central intersection to key locations.

### Implementation

- **Graph Model**: The city was modeled as a graph with intersections as nodes and roads as edges.
- **Algorithms**:
  - **DFS (Depth-First Search)**: Explored all possible paths but didn't always find the shortest path.
  - **BFS (Breadth-First Search)**: Found the shortest paths in an unweighted graph.
  - **Dijkstra's Algorithm**: Used for weighted graphs, considering varying travel times.

### Results

- **DFS**: Explored nodes in the order: 0 → 1 → 4 → 6 → 3 → 5 → 7.
- **BFS**: Explored nodes in the order: 0 → 1 → 3 → 7 → 4 → 5 → 6.
- **Dijkstra's Algorithm**:
  - Path to node 7: [0, 7], Distance = 3
  - Path to node 5: [0, 3, 5], Distance = 9
  - Path to node 6: [0, 1, 4, 6], Distance = 10

### Conclusion

Graph traversal algorithms like DFS and BFS were useful for exploring paths, but Dijkstra's algorithm was the most efficient for finding shortest paths considering different travel times. These algorithms are valuable in urban planning and logistics.

---

## Task 2: Optimal Delivery Route Using A* Algorithm

### Objective

The A* algorithm was applied to find the shortest delivery route from a warehouse to a customer's home on a city grid.

**Implementation**

- **Graph Representation**: The city was represented as a grid with intersections as nodes and travel times as edge weights.
- *A Algorithm**: Combined travel costs and estimated distance (heuristic) to efficiently find the shortest path.

**Findings**

- **Efficient Pathfinding**: A* found the shortest path by balancing actual and estimated costs.
- **Adaptability**: The algorithm adjusted to different traffic conditions, making it useful for dynamic environments.

**Conclusion**

A* proved effective in optimizing delivery routes and can be applied in real-world urban logistics and route planning.

---

## Task 3: Resource Allocation Using Randomized Hill Climbing (RHC)

**Objective**

The task was to allocate 100 units of resources across projects to maximize benefits or minimize completion times.

**Implementation**

- **RHC Algorithm**: Started with a random solution and improved it by flipping project allocations. The algorithm selected better solutions that either maximized benefits or minimized time.

**Results**

- **Maximization**: Successfully maximized total benefits to 145.
- **Minimization**: Minimized time to 0 by not allocating resources.

**Conclusion**

RHC efficiently handled resource allocation within constraints. The results may vary due to randomness, but more iterations can improve outcomes in complex scenarios.

Implementations and Results

```python
from collections import deque
import heapq

# Graph representation for BFS and DFS
graph = {
    0: [1, 3, 7],
    1: [4],
    3: [5],
    4: [6],
    5: [6],
    6: [],
    7: [4, 5]
}

# Weighted graph for Dijkstra's algorithm
city_graph_weighted = {
    0: {1: 2, 3: 2, 7: 3},
    1: {4: 4},
    3: {5: 7},
    4: {6: 4},
    5: {6: 2},
    6: {},
    7: {4: 5, 5: 6}
}

# Depth-First Search (DFS)
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=' ')

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Breadth-First Search (BFS)
def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            queue.extend(graph[node])

```

```python
# Dijkstra's Algorithm for Weighted Graph
def dijkstra(graph, start):
    min_distance = {node: float('infinity') for node in graph}
    min_distance[start] = 0
    pq = [(0, start)]
    came_from = {start: None}

    while pq:
        current_distance, current_node = heapq.heappop(pq)

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < min_distance[neighbor]:
                min_distance[neighbor] = distance
                came_from[neighbor] = current_node
                heapq.heappush(pq, (distance, neighbor))

    return min_distance, came_from

def reconstruct_path(came_from, start, target):
    path = []
    current = target
    while current is not None:
        path.append(current)
        current = came_from[current]
    path.reverse()
    if path[0] == start:
        return path
    else:
        return "Path does not exist"

# Run and display DFS order
print("DFS Order of nodes visited:")
dfs(graph, 0)
print("\n")

# Run and display BFS order
print("BFS Order of nodes visited:")
bfs(graph, 0)
print("\n")

# Run Dijkstra's algorithm and reconstruct paths
shortest_distances, paths_came_from = dijkstra(city_graph_weighted, 0)
for target in [7, 5, 6]:
    path = reconstruct_path(paths_came_from, 0, target)
    print(f"Shortest Path to {target} using Dijkstra's: {path} Distance = {shortest_distances[target]}")
```

```
PS C:\Users\musta\OneDrive\Documents\GitHub\cosc4368-artificial-intelligence>
DFS Order of nodes visited:
0 1 4 6 3 5 7

BFS Order of nodes visited:
0 1 3 7 4 5 6

Shortest Path to 7 using Dijkstra's: [0, 7] Distance = 3
Shortest Path to 5 using Dijkstra's: [0, 3, 5] Distance = 9
Shortest Path to 6 using Dijkstra's: [0, 1, 4, 6] Distance = 10
```

```python
import heapq
import math

# Graph representation with nodes as grid positions and weights as edge costs from the image
graph = {
    (0, 0): {(1, 0): 4, (0, 1): 2},
    (0, 1): {(0, 0): 2, (1, 1): 4, (0, 2): 3},
    (0, 2): {(0, 1): 3, (1, 2): 4, (0, 3): 6},
    (0, 3): {(0, 2): 6, (1, 3): 2, (0, 4): 8},
    (0, 4): {(0, 3): 8, (1, 4): 4},

    (1, 0): {(0, 0): 4, (2, 0): 3, (1, 1): 1},
    (1, 1): {(1, 0): 1, (0, 1): 4, (2, 1): 3, (1, 2): 2},
    (1, 2): {(1, 1): 2, (0, 2): 4, (2, 2): 1, (1, 3): 1},
    (1, 3): {(1, 2): 1, (0, 3): 2, (2, 3): 3, (1, 4): 2},
    (1, 4): {(1, 3): 2, (0, 4): 4, (2, 4): 6},

    (2, 0): {(1, 0): 3, (3, 0): 9, (2, 1): 3},
    (2, 1): {(2, 0): 3, (1, 1): 3, (3, 1): 2, (2, 2): 1},
    (2, 2): {(2, 1): 1, (1, 2): 1, (3, 2): 7, (2, 3): 3},
    (2, 3): {(2, 2): 3, (1, 3): 3, (3, 3): 2, (2, 4): 5},
    (2, 4): {(2, 3): 5, (1, 4): 6, (3, 4): 5},

    (3, 0): {(2, 0): 9, (4, 0): 2, (3, 1): 2},
    (3, 1): {(3, 0): 2, (2, 1): 2, (4, 1): 5, (3, 2): 7},
    (3, 2): {(3, 1): 7, (2, 2): 7, (4, 2): 6, (3, 3): 7},
    (3, 3): {(3, 2): 7, (2, 3): 2, (4, 3): 9, (3, 4): 3},
    (3, 4): {(3, 3): 3, (2, 4): 5, (4, 4): 1},

    (4, 0): {(3, 0): 2, (4, 1): 5},
    (4, 1): {(4, 0): 5, (3, 1): 5, (4, 2): 6},
    (4, 2): {(4, 1): 6, (3, 2): 6, (4, 3): 9},
    (4, 3): {(4, 2): 9, (3, 3): 9, (4, 4): 1},
    (4, 4): {(3, 4): 1, (4, 3): 1}
}

# Heuristic function (Euclidean distance)
def heuristic(a, b):
    return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)

# A* Algorithm
def a_star(graph, start, goal):
    open_list = []
    heapq.heappush(open_list, (0, start))
    came_from = {}
    g_score = {node: float('infinity') for node in graph}
    g_score[start] = 0
    f_score = {node: float('infinity') for node in graph}
    f_score[start] = heuristic(start, goal)

    while open_list:
        current = heapq.heappop(open_list)[1]

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            return path[::-1], g_score[goal]

        for neighbor, cost in graph[current].items():
            tentative_g_score = g_score[current] + cost
            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
                heapq.heappush(open_list, (f_score[neighbor], neighbor))

    return None, float('infinity')
```

```python
# Define the start and goal nodes
start = (0, 0)
goal = (2, 3)

# Run the A* algorithm
path, travel_time = a_star(graph, start, goal)
print(f"Optimal path: {path}")
print(f"Total travel time: {travel_time}")


import random

# Objective function for minimization: f(x) = (x - 3)^2
def minimization_function(x):
    return (x - 3) ** 2

# Objective function for maximization: f(x) = -x^2 + 5
def maximization_function(x):
    return -(x) ** 2 + 5

# Randomized Hill Climbing Algorithm
def randomized_hill_climbing(objective_function, x_start, iterations=1000, step_size=1, maximize=False):
    current_x = x_start
    current_value = objective_function(current_x)

    for _ in range(iterations):
        # Generate a neighboring solution by making a small random change
        neighbor_x = current_x + random.choice([-step_size, step_size])
        neighbor_value = objective_function(neighbor_x)

        # For maximization, move if the neighbor's value is higher
        if maximize:
            if neighbor_value > current_value:  # Maximization condition
                current_x = neighbor_x
                current_value = neighbor_value
        else:
            # For minimization, move if the neighbor's value is lower
            if neighbor_value < current_value:  # Minimization condition
                current_x = neighbor_x
                current_value = neighbor_value

    return current_x, current_value

# Test Minimization Function
optimal_x, optimal_value = randomized_hill_climbing(minimization_function, x_start=0, iterations=1000, maximize=False)
print(f"Optimal x for minimization: {optimal_x}, Function value: {optimal_value}")

# Test Maximization Function
optimal_x, optimal_value = randomized_hill_climbing(maximization_function, x_start=0, iterations=1000, maximize=True)
print(f"Optimal x for maximization: {optimal_x}, Function value: {optimal_value}")
```

```
PS C:\Users\musta\OneDrive\Documents\GitHub\cosc4368-artificial-intelligence>
Optimal path: [(1, 0), (1, 1), (1, 2), (1, 3), (2, 3)]
Total travel time: 11
Optimal x for minimization: 3, Function value: 0
Optimal x for maximization: 0, Function value: 5
```

```python
import random

# Define the minimization function: f(x) = (x - 3)^2
def minimize_function(x):
    return (x - 3) ** 2

# Define the maximization function: f(x) = -x^2 + 5
def maximize_function(x):
    return -(x ** 2) + 5

# Randomized Hill Climbing Algorithm
def hill_climbing_optimization(objective_fn, start_x, num_iterations=1000, step=1, maximize=False):
    current_solution = start_x
    current_value = objective_fn(current_solution)

    for _ in range(num_iterations):
        # Generate a neighbor by modifying the current solution slightly
        neighbor_x = current_solution + random.choice([-step, step])
        neighbor_value = objective_fn(neighbor_x)

        # For maximization, update the current solution if the neighbor is better
        if maximize:
            if neighbor_value > current_value:
                current_solution, current_value = neighbor_x, neighbor_value
        # For minimization, update the current solution if the neighbor is better
        else:
            if neighbor_value < current_value:
                current_solution, current_value = neighbor_x, neighbor_value

    return current_solution, current_value

# Test Minimization
optimal_x, optimal_value = hill_climbing_optimization(minimize_function, start_x=0, maximize=False)
print(f"Optimal solution for minimization: x={optimal_x}, Value={optimal_value}")

# Test Maximization
optimal_x, optimal_value = hill_climbing_optimization(maximize_function, start_x=0, maximize=True)
print(f"Optimal solution for maximization: x={optimal_x}, Value={optimal_value}")


# Randomized Hill Climbing for resource allocation
def calculate_benefit(solution, project_list):
    return sum(project['benefit'] for idx, project in enumerate(project_list) if solution[idx] == 1)

def calculate_time(solution, project_list):
    return sum(project['est_time'] for idx, project in enumerate(project_list) if solution[idx] == 1)

# Feasibility check to ensure we stay within the resource constraints
def check_feasibility(solution, project_list, available_resources):
    total_resources = sum(project['resource'] for idx, project in enumerate(project_list) if solution[idx] == 1)
    return total_resources <= available_resources

```

```python
# Randomized Hill Climbing for project selection
def hill_climbing_project_selection(project_list, available_resources, objective_fn, maximize=True):
    # Initialize random solution
    current_solution = [random.randint(0, 1) for _ in range(len(project_list))]

    # Ensure the solution fits the resource constraints
    while not check_feasibility(current_solution, project_list, available_resources):
        current_solution = [random.randint(0, 1) for _ in range(len(project_list))]

    current_value = objective_fn(current_solution, project_list)

    for _ in range(1000):
        # Create a neighbor solution by flipping a project choice
        neighbor_solution = current_solution[:]
        project_idx = random.randint(0, len(project_list) - 1)
        neighbor_solution[project_idx] = 1 - neighbor_solution[project_idx]

        # Check feasibility of the new solution
        if check_feasibility(neighbor_solution, project_list, available_resources):
            neighbor_value = objective_fn(neighbor_solution, project_list)

            # Update if the neighbor solution improves the objective
            if (maximize and neighbor_value > current_value) or (not maximize and neighbor_value < current_value):
                current_solution, current_value = neighbor_solution, neighbor_value

    return current_solution, current_value

# Test case 1: Maximize benefit
projects_1 = [
    {'resource': 20, 'benefit': 40},
    {'resource': 30, 'benefit': 50},
    {'resource': 25, 'benefit': 30},
    {'resource': 15, 'benefit': 25}
]
resources_available = 100
solution, value = hill_climbing_project_selection(projects_1, resources_available, calculate_benefit, maximize=True)
print(f"Maximizing Benefit - Solution: {solution}, Total Benefit: {value}")

# Test case 2: Minimize time
projects_2 = [
    {'resource': 10, 'est_time': 15},
    {'resource': 40, 'est_time': 60},
    {'resource': 20, 'est_time': 30},
    {'resource': 25, 'est_time': 35},
    {'resource': 5, 'est_time': 10}
]
solution, value = hill_climbing_project_selection(projects_2, resources_available, calculate_time, maximize=False)
print(f"Minimizing Time - Solution: {solution}, Total Time: {value}")

# Test case 3: Maximize benefit
projects_3 = [
    {'resource': 50, 'benefit': 80},
    {'resource': 30, 'benefit': 45},
    {'resource': 15, 'benefit': 20},
    {'resource': 25, 'benefit': 35}
]
solution, value = hill_climbing_project_selection(projects_3, resources_available, calculate_benefit, maximize=True)
print(f"Maximizing Benefit - Solution: {solution}, Total Benefit: {value}")
```

```
PS C:\Users\musta\OneDrive\Documents\GitHub\cosc4368-artificial-intelligence>
Optimal solution for minimization: x=3, Value=0
Optimal solution for maximization: x=0, Value=5
Maximizing Benefit - Solution: [1, 1, 1, 1], Total Benefit: 145
Minimizing Time - Solution: [0, 0, 0, 0, 0], Total Time: 0
Maximizing Benefit - Solution: [1, 1, 1, 0], Total Benefit: 145
```