COMPSCI 2XC3 Final Project

Xun Cheng, Lihan Zhang, Shaoyuan Wu

April 2025

Part 1: Team charter

1. Group Communication Protocol

Our group has established the following weekly work plan and communication method:

Day	Activity
Monday	Weekly meeting on Zoom
Tuesday–Friday	Individual work
Saturday	Communicate questions via Teams chat
Sunday	Collaborative coding session

We use **Zoom** for weekly meetings every Monday to discuss general progress and planning. For daily collaboration and asynchronous communication, we have a private **Microsoft Teams group chat**. Members are encouraged to post questions or issues during the week, which will be addressed collectively on Saturdays to ensure that all members are informed before collaborative coding on Sundays. This schedule prevents miscommunication and ensures consistent progress.

2. Communication Agreement Penalty

If a group member repeatedly fails to adhere to the communication protocol (for example, not responding to messages or missing agreed meetings), this behavior will be reflected in **peer evaluation scores**. In severe cases, other team members will report the issue to the professor for further action.

3. Collaboration Tools

We will use **Visual Studio Code** as our primary development environment. For version control and collaboration, we will use **Git** and a shared repository to track all changes and contributions.

4. Conflict Resolution Protocol

All team members are encouraged to participate in the work. Initially, tasks will be divided according to personal interest and then balanced for fairness. In case of any disputes or disagreements, we will first attempt to resolve them respectfully within the group through discussion during meetings or via group chat. If the issue remains unresolved, we will seek the assistance of the TA or the course instructor.

5. Additional notes

Our team values fairness, transparency, and accountability. All contributions and discussions will be open and inclusive to ensure a positive and productive group dynamic.

Part 2: Single source shortest path algorithms

2.1 Code for variation of Dijkstra's algorithm

```
class SPAlgorithm:
    def calc_sp(self, graph, source, dest):
        raise NotImplementedError("This method should be overridden by subclasses")

class Dijkstra(SPAlgorithm):
    def __init__(self, k=10000):
        self.k = k

    def calc_sp(self, graph:Graph, source, dest):
        k = self.k
        distance = {node:float('inf') for node in graph.adj}
        distance[source] = 0

        path = {node: [] for node in graph.adj}
        path[source] = [source]

        relax_count = {node: 0 for node in graph.adj} #counter

        priority_queue = [(0, source)]  # (distance, node)

        while priority_queue:
            dist_u, u = heapq.heappop(priority_queue) #pop the node with the smallest distance
            if relax_count[u] >= k: #if the node has been relaxed k times, skip
                continue

            for v in graph.adj[u]:
                weight = graph.get_adj_nodes(u)[v]['weight']
                if distance[v] > distance[u] + weight:
                    distance[v] = distance[u] + weight
                    path[v] = path[u] + [v]
```

```
                    heapq.heappush(priority_queue, (distance[v], v))
                    relax_count[v] += 1

        return distance, path
```

Listing 1: Dijkstra Algorithm

2.2 Code for variation of Bellman-Ford

```
class BellmanFord(SPAlgorithm):
    def __init__(self, k=10000):
        self.k = k

    def calc_sp(self, graph, source, dest):
        k = self.k
        distance = {node: float('inf') for node in range(len(graph.adj))}
        distance[source] = 0

        path = {node: [] for node in range(len(graph.adj))}
        path[source] = [source]

        for _ in range(k):  # at most k iterations
            update = False
            for u in graph.adj:
                for v in graph.adj[u]:
                    weight = graph.get_adj_nodes(u)[v]['weight']
                    if distance[u] + weight < distance[v]:
                        distance[v] = distance[u] + weight
                        path[v] = path[u] + [v]
                        update = True
            if not update:
                break

        return distance, path
```

Listing 2: Bellman_Ford Algorithm

2.3 Experiment

```
def experiment():
    runs = 150
    node_size = [50, 100, 200]  #different graph size
    densities = [0.01, 0.05, 0.1] #control the density of the graph
    relax_time = [1, 5, 10, 20] #Maximum number of relaxations allowed per node

    for n in node_size:
        for density in densities:
            max_edges = n * (n - 1)  # max edges in a directed graph
            edge_count = int(density * max_edges)    # number of edges
```

```
11
12              for k in relax_time:
13                  dijkstra_runtimes = []
14                  bellman_runtimes = []
15
16                  for _ in range(runs):
17                      graph = create_random_graph(n, edge_count) #
    random graph(nodes, edges)
18
19                      # Dijkstra
20                      start = timeit.default_timer()
21                      dijkstra(graph, 0, k)
22                      stop = timeit.default_timer()
23                      dijkstra_runtimes.append((stop - start) * 1000)
      # ms
24
25                      # Bellman-Ford
26                      start = timeit.default_timer()
27                      bellman_ford(graph, 0, k)
28                      stop = timeit.default_timer()
29                      bellman_runtimes.append((stop - start) * 1000)
     # ms
30
31                  dijkstra_mean = sum(dijkstra_runtimes) / runs
32                  bellman_mean = sum(bellman_runtimes) / runs
33
34                  print(f"  Dijkstra mean runtime:     {dijkstra_mean
    :.6f} ms")
35                  print(f"  Bellman-Ford mean runtime: {bellman_mean
    :.6f} ms")
36
37                  draw_plot(dijkstra_runtimes,dijkstra_mean, "
    Dijkstra")
38                  draw_plot(bellman_runtimes,bellman_mean, "Bellman-
    Ford")
39
40          return 0
41  experiment()
```

Listing 3: Experiment

Part 3: All-pair shortest path algorithm

```
1  def allpair_dijkstra(G): # dijkstra complexity is O(E + VlogV) = O(
       E) for sparse graph, O(V^2) for dense graph, therefore expected
        complexity of allpair is O(V*(E + VlogV)) = O(VE) for sparse
       graph, O(V^3) for dense graph
2      # O(V^3) for dense graph, O(VE) for sparse graph
3      result = [[-1 for _ in range(len(G))] for _ in range(len(G))]
4      for src in G:
5          distance, _ = dijkstra(G, src, len(G))
6          for dst in G:
7              if src != dst:
8                  result[src][dst] = distance[dst]
9      return result
```

```
10
11 def allpair_bellman_ford(G): # bellman-ford complexity is O(VE) for
       sparse graph, O(V^2) for dense graph, therefore expected
   complexity of allpair is O(V*(VE)) = O(V^2E) for sparse graph,
   O(V^3) for dense graph
12     # O(V^2E) for sparse graph, O(V^3) for dense graph
13     result = [[-1 for _ in range(len(G))] for _ in range(len(G))]
14     for src in G:
15         distance, _ = bellman_ford(G, src, len(G))
16         for dst in G:
17             if src != dst:
18                 result[src][dst] = distance[dst]
19     return result
```

Listing 4: All-pair shortest path algorithm

Part 4: A* algorithm

4.1 Code

```
1  def euclidean_distance(station1_id, station2_id, coordinates_dict):
2      coord1 = coordinates_dict[station1_id]
3      coord2 = coordinates_dict[station2_id]
4      return math.sqrt((coord1[0] - coord2[0]) ** 2 + (coord1[1] -
       coord2[1]) ** 2)
5
6  class AStar(SPAlgorithm):
7      def __init__(self, heuristic):
8          self.heuristic = heuristic
9
10     def calc_sp(self, graph, source, dest):
11         heuristic = self.heuristic
12         open_list = [(0, source)]  #to record the node
13         closed_list = set() #to check visited node
14
15         g_values = {node: float('inf') for node in range(len(graph.
       adj))} #set every cost to infinity
16         g_values[source] = 0
17
18         parent = {source: None}
19
20         while open_list:
21             current_f, current_node = heapq.heappop(open_list) #
       variable to receive the smallest value in the list
22
23             if current_node == dest:
24                 path = []
25                 while current_node is not None:
26                     path.append(current_node)
27                     current_node = parent[current_node]
28                 return path[::-1]
29
30             closed_list.add(current_node)
31
32
```

```
            for neighbor_raw in graph.adj[current_node]:
                neighbor = int(neighbor_raw)

                if neighbor in closed_list:
                    continue

                if neighbor not in g_values:
                    g_values[neighbor] = float('inf')

                cost = graph.adj[current_node][neighbor]['weight']
                tentative_g = g_values[current_node] + cost

                if tentative_g < g_values[neighbor]:
                    g_values[neighbor] = tentative_g
                    f_value = tentative_g + heuristic(neighbor)
                    heapq.heappush(open_list, (f_value, neighbor))
                    parent[neighbor] = current_node

        return None  #if no path found
```

Listing 5: A* Algorithm

4.2 Report: Analysis of A* vs Dijkstra

1. The Dijkstra algorithm will traverse all possible paths to find the target, and its worst case performance will be poor. However, A* adds a heuristic function to prioritize the paths that are more likely to reach the destination based on the cost to the target, hence improving efficiency.

2. I will design an experimental environment to compare the execution time, number of expanded nodes, and path length of Dijkstra and A* through random graphs. In the experiment, we use the control variable method to keep the starting and ending points consistent and the data structure consistent, change the size, density, and obstacle distribution of the graph, and repeat multiple times to find the average result.

3. If the heuristic function is randomly generated, then A* may perform worse because the distance to the goal is not reasonable. It may expand many irrelevant nodes and even be slower than Dijkstra. In contrast, Dijkstra will traverse all possible paths but will not be misled by the wrong heuristic function.

4. I would use A* in the case of GPS navigation, obstacle avoidance and navigation for drones/autonomous vehicles. In this case, big data comes in and a faster algorithm is needed to give the shortest path.

Part 5: Compare Shortest Path Algorithms

5.1 Code part

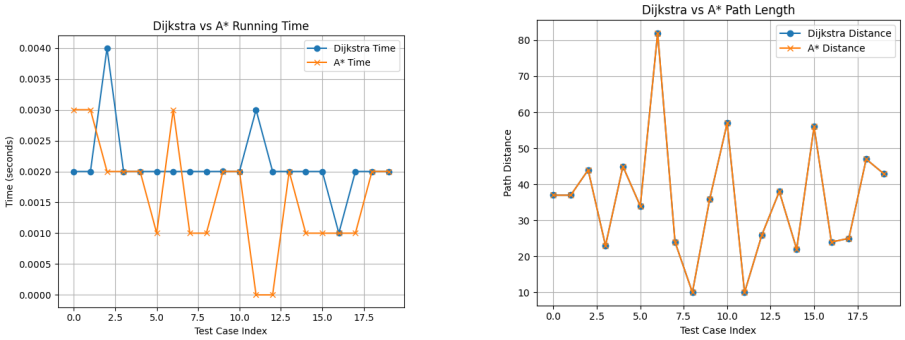
```
df1 = pd.read_csv('london_connections.csv')

df2 = pd.read_csv('london_stations.csv')

G = nx.Graph()

for _, row in df1.iterrows():
    G.add_edge(row['station1'], row['station2'], weight=row['time'
    ], line = row['line'])

station_coordinates = {
    row['id']: (row['latitude'], row['longitude']) for _, row in
    df2.iterrows()
}

def count_line_changes(path, graph):
    lines_used = []
    for i in range(len(path) - 1):
        u, v = path[i], path[i+1]
        line = graph[u][v]['line']
        lines_used.append(line)
    return len(set(lines_used))

def compare_algorithms(graph, source, target, coordinates_dict):

    # Dijkstra
    start_d = time.time()
    dist_d, path_d = dijkstra(graph, source, k=len(graph) - 1)
    end_d = time.time()
    dijkstra_time = end_d - start_d
    dijkstra_distance = dist_d.get(target, float('inf'))

    # A*
    heuristic = lambda current_id: euclidean_distance(current_id,
    target, coordinates_dict)
    start_a = time.time()
    path_a = A_Star(graph, source, target, heuristic)
    end_a = time.time()
    astar_time = end_a - start_a
    if path_a and len(path_a) > 1:
        astar_distance = 0
        for i in range(len(path_a) - 1):
            u = path_a[i]
            v = path_a[i + 1]
            astar_distance += graph[u][v]['weight']
    else:
        astar_distance = float('inf')

    line_count = count_line_changes(path_a, graph)

    return {
        "source": source,
        "target": target,
        "dijkstra_time": dijkstra_time,
        "astar_time": astar_time,
        "dijkstra_distance": dijkstra_distance,
        "astar_distance": astar_distance,
```

```
        "dijkstra_path": path_d.get(target, []),
        "astar_path": path_a,
        "line_changes": line_count
    }


all_stations = list(station_coordinates.keys())
results = []

for _ in range(20):
    s, t = random.sample(all_stations, 2)
    result = compare_algorithms(G, s, t, station_coordinates)
    results.append(result)
```

Listing 6: A* main loop with parent tracking

5.2 Explanation

- A* average outperforms better than Dijkstra in terms of runtime, especially when the source and destination stations are far apart or require multiple transfers. That's because A* don't have to try all the possible path, it's way more efficient than Dijkstra to guide A* toward the destination.
- 1. Both algorithms perform similarly since the graph is simple
  2. A* is slightly faster, since the heuristic can leads to destination.
  3. A* significantly reduces time by avoiding unnecessary exploration, while Dijkstra scans broadly.



(a) Dijkstra vs A* Running Time

(b) Dijkstra vs A* Path Length

Figure 1: Two images side by side.

Part 6: Organize your code as per UML diagram

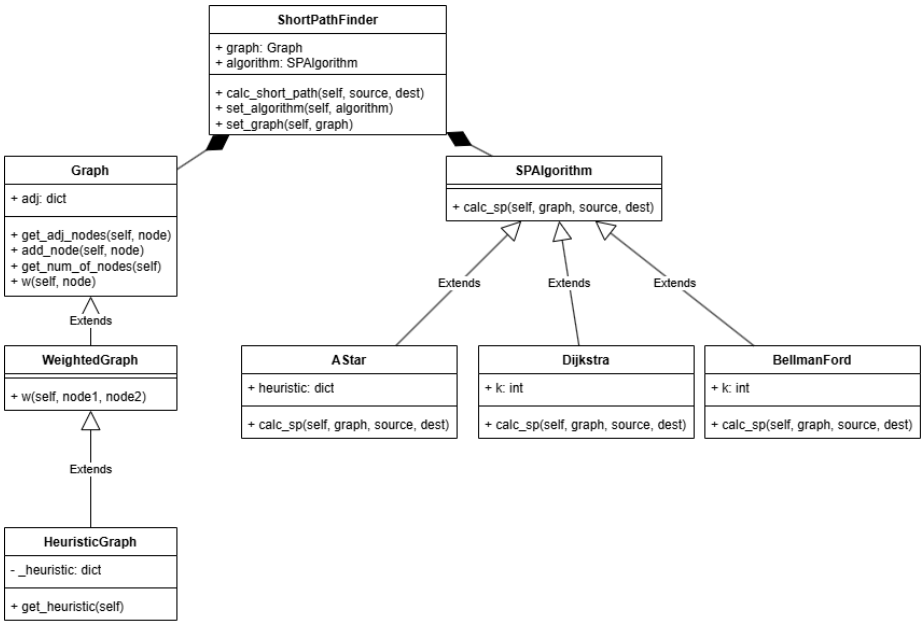


Figure 2: UML Diagram

- Overall, the design pattern of the OOP project is to respect the original codes, while trying to not break the instructions given by the UML diagram. For example, there is a problem that Dijkstra and Bellman-ford algorithm have different return values compared to A* algorithm, hereby, we identified the outputs by reading if it is a tuple and excluded the unnecessary element. There is also a problem when trying to fit in A* and the k times of relaxing in SPAlgorithm, therefore we designed these parameters as fields, and read those fields in the functions.

- If the node is designed to be a string, or any other basic Python type, we have two solutions. Firstly, our data structure of the adjacency list can search for it, because it is essentially a dictionary. Secondly, a field of names can be implemented into ShortPathFinder, aiming to convert the node names into corresponding unique index. If the user want to include more information in the node names, a tuple or a self-designed data structure is recommended, and it is also plausible by going through two kinds of method described earlier.