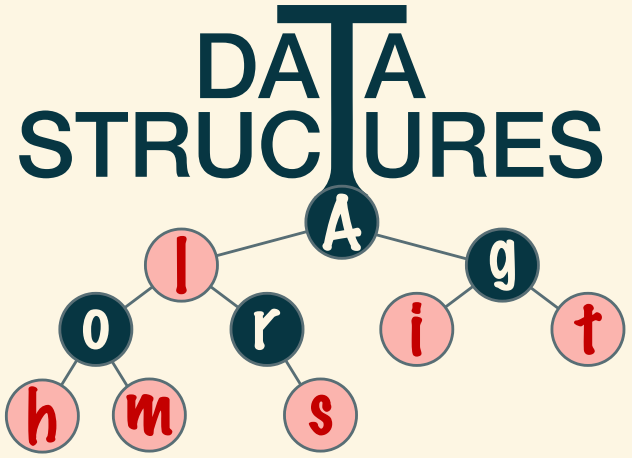
Home



WEEK 10: SINGLE-SOURCE SHORTEST PATHS (LAB)

Kelvin Goh

2026-03-12

Home

INTRO

Home

LEARNING OBJECTIVES

By the end of this lab session, you should be able to:

1. **Understand single-source shortest path algorithms:** Explain how Dijkstra's (priority queue, greedy) and Bellman-Ford (relaxation iterations) work, compare their time complexities, and identify when to use each based on graph properties (negative weights, cycles).
2. **Implement Dijkstra's algorithm:** Implement Dijkstra's in C++ using priority queues and maps, handle string-based vertex names with adjacency lists, track predecessors, and detect negative edge weights.
3. **Apply shortest path algorithms:** Use Bellman-Ford to solve pathfinding problems in game development and model grid-based problems as graphs to find optimal paths.

Home

OVERVIEW OF LAB ACTIVITIES

This lab consists of three main activities:

Home

ACTIVITY 1: MULTIPLE CHOICE QUESTIONS

Home

EXERCISE: UNDERSTANDING DIJKSTRA'S AND BELLMAN-FORD ALGORITHMS

This activity consists of multiple choice questions to test your understanding of **single-source shortest path algorithms**, specifically **Dijkstra's algorithm** and **Bellman-Ford algorithm**.

Home

INSTRUCTIONS

- Complete the xSITe quiz questions covering Dijkstra's and Bellman-Ford algorithms
- Questions may include:
  - Selecting the correct algorithm for a given scenario
  - Identifying the order of vertex processing
  - Calculating shortest distances after specific iterations
  - Understanding time complexity and space complexity
  - Recognizing when negative cycles exist

Submission (xSITe Quizzes, 10 marks). Complete the quiz "Week 10 Lab Exercise 1" on xSITe.

Home

KEY CONCEPTS TO REVIEW

Before attempting the quiz, make sure you understand:

Home

ACTIVITY 2: SINGLE-SOURCE SHORTEST PATH WITH DIJKSTRA'S ALGORITHM

Home

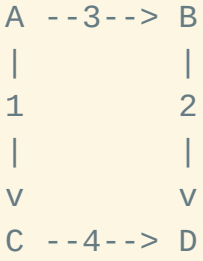
PROBLEM DESCRIPTION

You are given a **weighted directed graph** represented as an adjacency list, where each vertex is identified by a string name. Your task is to implement **Dijkstra's algorithm** to find the **shortest distances** from a given source vertex to all other vertices in the graph.

This is a classic **single-source shortest path** problem where:

- Each vertex is identified by a string (e.g., "A", "B", "start", "end")
- Edges have non-negative weights
- We need to find the shortest path from a source vertex to all reachable vertices

Consider the following graph:



Home

HYPOTHETICAL SCENARIO: LOGISTICS ROUTING IN NORTHERN SINGAPORE (SHORTEST PATH CALCULATION)

A courier hub in **Woodlands (A)** needs to deliver a parcel to a customer in **Bukit Panjang (D)**. The values on the map represent the actual distance in **kilometers (km)** between the locations.

Using the starting point of **Woodlands (A)**, calculate the shortest distance to each town:

- **Distance to Woodlands (A): 0 km** (Origin)
- **Shortest distance to Yishun (B): 3 km** (Direct via Woodlands (A) → Yishun (B))
- **Shortest distance to Kranji (C): 1 km** (Direct via Woodlands (A) → Kranji (C))
- **Shortest distance to Bukit Panjang (D): 5 km** (path A → C → D with cost 1 + 4 = 5, or A → B → D with cost 3 + 2 = 5)

Result: In this specific graph, both the Yishun route and the Kranji route result in a total distance of **5 km**.

Home

CONSTRAINTS

- $1 \leq |V| \leq 1000$, where $|V|$ is the number of vertices
- $0 \leq |E| \leq 10000$, where $|E|$ is the number of edges
- Edge weights are integers (typically non-negative, but **negative weights may appear in test cases** — since Dijkstra's greedy approach assumes no negative edges, your implementation must detect them and return an empty map to indicate the input is invalid for this algorithm).
- Vertex names are strings (alphanumeric)
- The graph may be disconnected (some vertices may be unreachable from the source)

Home

EXERCISE

Home

EXERCISE (CONT.)

4. **Implementation requirements**:
    - Use **Dijkstra's algorithm** with a priority queue (min-heap)
    - Initialize all distances to `INT_MAX` except the source (distance = 0)
    - Track visited vertices to avoid reprocessing
    - Update `predecessors` map when relaxing edges
    - Handle the case where negative edge weights are detected (return empty map)
    - Attached are 4 Test files to test your algorithm.

Submission (Gradescope, 20 marks). Submit your completed `dijkstra.cpp` to the "Week 10 Lab Exercise 2" assignment on Gradescope.

Home

KEY POINTS & IMPLEMENTATION DETAILS

- **Initialization:** All distances = `INT_MAX`, source = 0, all vertices unvisited, predecessors = empty strings.
- **Priority Queue:** Use `std::priority_queue` with `std::greater` for min-heap. Stores `(distance, vertex_name)` pairs, always processes minimum distance first.
- **Relaxation:** For each edge, check negative weights (return empty map if found). If `dist[current] + weight < dist[neighbor]`, update distance and predecessor, push to queue.
- **Lazy Deletion:** Multiple entries per vertex may exist in queue; skip already visited vertices to avoid update/remove operations.
- **Edge Cases:** Negative weights return empty map; unreachable vertices remain `INT_MAX`; disconnected graphs handled naturally.

Home

ACTIVITY 3: GAME PATHFINDING WITH SHORTEST PATH ALGORITHMS

Home

PROBLEM DESCRIPTION

You are designing a **pathfinding system** for a 2D grid-based game. The game world consists of a grid where each cell has a **terrain cost** (movement cost). The player character starts at position (s_r, s_c) and needs to reach the target position (t_r, t_c).

The character can move in **4 directions** (up, down, left, right) to adjacent cells. The cost of moving from one cell to an adjacent cell is the **terrain cost of the destination cell**. Your task is to find the **minimum total cost** to move from the start to the target.

This is a classic **single-source shortest path** problem where:

- Each grid cell is a **vertex**
- Edges connect adjacent cells (4-neighbour connectivity)
- Edge weights are the terrain costs of destination cells
- We need to find the shortest path from source (s_r, s_c) to target (t_r, t_c)

Home

PROBLEM DESCRIPTION (CONT.)

Consider a 3×4 game grid with terrain costs:

```
   0  1  2  3
 ------------
0| 1  5  2  1
1| 3  1  4  2
2| 1  2  1  3
```

Starting at $(0, 0)$ with cost 1, and target at $(2, 3)$ with cost 3.

Example path: $(0, 0) \to (0, 1) \to (1, 1) \to (2, 1) \to (2, 2) \to (2, 3)$

Path cost: $1 + 5 + 1 + 2 + 1 + 3 = 13$ (includes start cell cost)

Note: The optimal path might be different! Use Bellman-Ford to find it.

Home

CONSTRAINTS

- $1 \leq rows, cols \leq 100$
- $0 \leq \text{terrain\_cost}[i][j] \leq 1000$
- Start and target positions are always valid (within grid bounds)
- The grid is guaranteed to be connected (all cells are reachable)

Home

EXERCISE

1. **Download:** Get `game_pathfinding.zip` from xSITe. The starter file `game_pathfinding.cpp` contains the stub:

```cpp
int shortestPathCost(int rows, int cols,
                     const vector<vector<int>> &terrain,
                     int start_r, int start_c,
                     int target_r, int target_c);
```

2. **Input format**:
   - First line: `rows cols`
   - Next line: `start_r start_c target_r target_c`
   - Next `rows` lines: each line has `cols` integers representing terrain costs
3. **Output format**: Return the minimum total cost to move from start to target (including the cost of the start cell).

Home

EXERCISE (CONT.)

4. **Implementation requirements**:
    - Use **Bellman-Ford algorithm**
    - Map (row, col) to a 1D index: `idx = row * cols + col`
    - Handle 4-directional movement (up, down, left, right)
    - Use appropriate data structures (using relaxation for Bellman-Ford)
    - Attached are 3 test files for testing your algorithm.

Submission (Gradescope, 30 marks). Upload completed `game_pathfinding.cpp` to Gradescope "Week 10 Lab Exercise 3".

Home

KEY POINTS

- **Bellman-Ford:** Relax all edges $(|V| - 1)$ times. Slower but handles negative weights and detects cycles.
- **Graph:** Vertices = grid cells (row, col); edges = 4-neighbour connections; edge weight = terrain cost of destination cell.

CONCLUSION

Home

WRAP-UP

By the end of this lab you should be able to:

1. Construct graphs for different use cases
2. Use appropriate data structures to implement single source shortest paths algorithms
3. Implement **Dijkstra's or Bellman-Ford algorithm** to find shortest paths from a source vertex
4. Apply graph algorithms to solve real-world problems (shortest pathfinding in games)

Home

OUTLOOK

This lab covered both Minimum Spanning Trees and Single-Source Shortest Paths as fundamental graph algorithms. The remaining weeks cover:

Dynamic Programming and Greedy Algorithms

Versatile optimization techniques for solving complex problems