Dijkstra's Shortest Path Algorithm Implementation

[Hassan Almosa]

May 2025

Contents

1 Building and Running Project and Tests

To build, compile with JDOM libraries, simply run script (`build.sh`):

```
./build.sh

// or manually compile
#!/bin/bash
mkdir -p bin
javac -cp lib/jdom-2.0.6/jdom-2.0.6.jar -d bin src/*.java
```

Listing 1: Build script (build.sh)

To run tests, after building, simply run script:

```
./run_tests.sh

// or manually and run indvidual tests
echo "data/graphs/graphSpecExample.graphml 0" | java -cp lib/jdom-2.0.6/jdom-2.0.6.jar:bin GraphShortestPathDriver
```

Listing 2: Updated test script commands

---

2 Graph Representation and Core Classes

2.1 Adjacency List vs Adjacency Matrix Decision

Initially, we weighed cons and pros of both implementation of adjacency representations and it was decided to go with an adjacency list to avoid $O(V^2)$ space complexity since our test entries range from 100-1000 edges/vertices [14].

2.2 The `Graph` Interface

First we needed to redefine the Graph interface building upon the structure used in Lab 3. The interface was edited with get weights call from the directed adjacency list but this was a bit confusing and led to bugs where I'd call the wrong method. Hence, a clean interface focused only on weighted graphs was needed.

```java
public interface Graph {
    void addVertex(Vertex vertex);
    void addEdge(Vertex source, Vertex target, int weight); // + weight
    List<Vertex> getVertices();
    // returns neighbours AND edge weight
    Map<Vertex, Integer> getAdjacentVertices(Vertex vertex);
    Vertex getVertex(String label);
    boolean hasVertex(Vertex vertex);
    int getVertexCount();
}
```

Listing 3: Refined `Graph.java` Interface

The method `getAdjacentVertices(Vertex vertex)` is key in here, it returns a `Map<Vertex, Integer>` where the int value is the weight of the edge to the neighbouring `Vertex`.

2.3 The Vertex Class

Each node in the graph is represented by a `Vertex` object.

```
public class Vertex implements Comparable<Vertex> {
    private String label;

    public Vertex(String label) {
        this.label = label;
    }
    public String getLabel() {
        return label;
    }
}
```

Listing 4: `Vertex.java` - Core Structure

Implementing `Comparable<Vertex>` allows sorting vertices by lexicographical order.

```
@Override
public int compareTo(Vertex other) {
    return this.label.compareTo(other.label);
}
```

Listing 5: `Vertex.java` - `compareTo` Method

2.4 Graph Implementation: `MyGraph`

Using adjacency lists for the implementation in `MyGraph.java`:

```
public class MyGraph implements Graph {
    private Map<String, Vertex> vertices;
    private Map<Vertex, Map<Vertex, Integer>> adjacencyList;

    public MyGraph() {
        this.vertices = new TreeMap<>();
        this.adjacencyList = new HashMap<>();
    }
}
```

Listing 6: `MyGraph.java` - Adjacency List Structure

A `TreeMap<String, Vertex> vertices` was used to store vertex objects, mapping their labels to the objects. Using `TreeMap` makes sure that if I were to iterate over `this.vertices.values()` directly, they would be processed in lexicographical order of labels. A `HashMap` would also work and offer $O(1)$ average-case lookups.

Adding a Vertex:

```
@Override
public void addVertex(Vertex vertex) {
    if (!vertices.containsKey(vertex.getLabel())) {
        vertices.put(vertex.getLabel(), vertex);
        adjacencyList.put(vertex, new HashMap<>());
    }
}
```

Listing 7: `MyGraph.java` - `addVertex` Method

If the vertex isn't already known by its label, it is added to the `vertices` map, and an entry for it is created in the `adjacencyList` with an empty map for its neighbours.

Adding an Edge:

```java
@Override
public void addEdge(Vertex source, Vertex target, int weight) {
    addVertex(source);
    addVertex(target);

    adjacencyList.get(source).put(target, weight);
}
```

Listing 8: `MyGraph.java - addEdge` Method

This makes sure both source and target vertices are in the graph (calling `addVertex` handles this). Then it updates the adjacency list for the pointed `source` vertex.

Retrieving Vertices:

```java
@Override
public List<Vertex> getVertices() {
    List<Vertex> vertexList = new ArrayList<>(vertices.values());
    Collections.sort(vertexList); // sort based on Vertex.compareTo
    return vertexList;
}
```

Listing 9: `MyGraph.java - getVertices` Method

This retrieves all `Vertex` objects, place them in an `ArrayList`, and sort them in lexographical order by label.

---

3 Core Concepts of Dijkstra's Algorithm

3.1 Key Data Structures for Dijkstra's (in my Java context)

- `Map<Vertex, Integer> distances`: Stores the current shortest known path distance from the `sourceVertex` to every other `Vertex`. Its intialised to `Integer.MAX_VALUE` for all vertices except the `sourceVertex` (distance 0).

- `Map<Vertex, Vertex> predecessors`: For each vertex and stores the vertex that comes before it on the shortest path from that source. This is how the path is actually reconstructed.

- **Priority Queue mechanism** (for Q) [4]: This is the "frontier." It stores vertices that have been reached but whose shortest paths haven't been finalised by by visiting. They are prioritised by their current path distance. My implementations explore an `ArrayList`, Java's `PriorityQueue`, and a custom `DijkstraHeap`.

- `Set<Vertex> visitedVertices` (for S): Keeps track of vertices for which the shortest path has been finalized.

3.2 Handling the "NO PATH" Problem

Representing and identifying unreachable vertices was a challenge. My initial thought was to check if a vertex's distance remained `Integer.MAX_VALUE`, the pitfall I had to avoid was potential integer overflows if `distances.get(current)` was `Integer.MAX_VALUE` and I added a positive weight to it. The relaxation step checks 'if (newDistance < distances.get(neighbour))' [14], which implicitly handles the initial `Integer.MAX_VALUE` correctly as any valid path distance will be less than it. The final determination of "NO PATH" is handled in the 'DijkstraResult' class where if `getDistance(target)` returns `Integer.MAX_VALUE`, that means that no path was found.

---

4 Implementation Journey: Three Approaches

My exploration of Dijkstra's algorithm lead me to three different implementations within `DijkstraShortestPath.java`. All approaches used the same `Graph` interface and `MyGraph` implementations.

4.1 Approach 1: Dijkstra with ArrayList (Foundational Naive Dijkstra Implementation)

The first version, `dijkstraWithArrayList`, uses an `ArrayList` to manage unvisited nodes relying on a linear scan to find the minimum distance vertex.

1. **Initialisation**:

   - `distances<Vertex, Integer>`: Stores shortest known distances (source is 0, others `Integer.MAX_VALUE`).
   - `predecessors<Vertex, Vertex>`: Tracks the path
   - `unvisited<Vertex>`: An `ArrayList` storing all vertices nodes.
   - `visited<Vertex>`: A `Set` for finalised visited vertices.

```java
Map<Vertex, Integer> distances = new HashMap<>();
Map<Vertex, Vertex> predecessors = new HashMap<>();
Set<Vertex> visited = new HashSet<>();
List<Vertex> unvisited = new ArrayList<>();

for (Vertex v : graph.getVertices()) {
    distances.put(v, Integer.MAX_VALUE);
    unvisited.add(v);
}
distances.put(source, 0);
```

Listing 10: ArrayList Approach: Initialization

2. **Main Loop**: Continues while `unvisited` is not empty.

   - **Select Minimum**: Linearly scans `unvisited` nodes to find `current` vertex with the smallest distance.

```java
Vertex current = null;
int minDistance = Integer.MAX_VALUE;
for (Vertex v : unvisited) {
    if (distances.get(v) < minDistance) {
        minDistance = distances.get(v);
        current = v;
    }
}
```

```
unvisited.remove(current);
visited.add(current);
```

Listing 11: ArrayList Approach: Finding Minimum

- **Relaxation**: For each `neighbour` of `current`: if a shorter path through `current` is found then update `distances` and `predecessors` [14].

```
Map<Vertex, Integer> neighbours = graph.getAdjacentVertices(current);
for (Map.Entry<Vertex, Integer> entry : neighbours.entrySet()) {
    Vertex neighbour = entry.getKey();
    int weight = entry.getValue();
    if (!visited.contains(neighbour)) {
        int newDistance = distances.get(current) + weight;
        if (newDistance < distances.get(neighbour)) {
            distances.put(neighbour, newDistance);
            predecessors.put(neighbour, current);
        }
    }
}
```

Listing 12: ArrayList Approach: Relaxation

4.1.1 Reflection

This basic implemntation approach is simple yet results in $O(V^2)$ space complexity, its inefficient for sparse graphs due to the $O(V)$ scan in each of V iterations.

4.2 Approach 2: Dijkstra with Java's `PriorityQueue`

The `dijkstraWithPriorityQueue` method uses Java's built-in `PriorityQueue` to finding optimal minimum-distance between nodes.

1. **Data Structure**:
   - `distances`, `predecessors`, `visited` are similar.
   - `pq`: A `PriorityQueue<VertexDistance>`.
   - `VertexDistance`: A static nested class to pair a `Vertex` with its `distance` enabling comparison [14].

```
static class VertexDistance {
    Vertex vertex;
    int distance;
    VertexDistance(Vertex vertex, int distance) { /* … */ }
}
PriorityQueue<VertexDistance> pq =
    new PriorityQueue<>(Comparator.comparingInt(vd → vd.distance));
```

Listing 13: PriorityQueue Approach: VertexDistance Helper

2. **Initialisation**: Source vertex (wrapped in `VertexDistance`) added to `pq`.

```
distances.put(source, 0);
pq.offer(new VertexDistance(source, 0));
```

Listing 14: PriorityQueue Approach: Initialization

3. **Main Loop**: While `pq` is not empty.

   - **Extract Minimum**: `VertexDistance currentVD = pq.poll();`[14] Efficient ($O(logV)$) space complexity.
   - **Stale Entry Check**: If `currentVD.vertex` is already `visited`, skip (handles lack of direct `decreaseKey`) [14].

```java
VertexDistance currentVD = pq.poll();
if (visited.contains(currentVD.vertex)) {
    continue;
}
visited.add(currentVD.vertex);
```

Listing 15: PriorityQueue Approach: Extract Min Stale Check

   - **Relaxation**: If a shorter path to a `neighbour` is found, update `distances`, `predecessors`, and `pq.offer(new VertexDistance(neighbour, newDistance))`.

```java
if (newDistance < distances.get(neighbour)) {
    distances.put(neighbour, newDistance);
    predecessors.put(neighbour, currentVD.vertex);
    pq.offer(new VertexDistance(neighbour, newDistance));
}
```

Listing 16: PriorityQueue Approach: Relaxation

4.2.1 Reflection

This $O((V+E)\log V)$ approach is significantly faster for sparse graphs. The `VertexDistance` wrapper and re-adding entries to simulate `decreaseKey` are key practical aspects. [14] [3].

4.3 Approach 3: Dijkstra with Custom Heap (`DijkstraHeap`)

The third approach with `dijkstraWithCustomHeap` implementation, uses a custom built binary min-heap, named `DijkstraHeap`. This heap is designed as a static nested class within `DijkstraShortestPath.java`. In this implementation, it supports `decreaseKey` operation explicitly. This allows for more efficient updates to distances in the priority queue between vertices compared to re-adding entries, as we done with Java's `PriorityQueue`.

4.3.1 Data Structure

The `DijkstraHeap` uses two primary internal data structures and a helper class:

- **`List<HeapNode> heapList`**: An `ArrayList` that stores the actual heap elements.

- **`Map<Vertex, Integer> vertexToIndex`**: This map is key for efficiency of the `decreaseKey` operations. [14] It stores each `Vertex` currently in the heap and maps it to its current index in the `heapList`. This allows for $O(1)$ lookup of a node position, which is necessary before its key can be decreased and its position potentially adjusted.

- **HeapNode**: A nested class within DijkstraHeap. Each HeapNode object encapsulates a Vertex and its current distance (key) in the priority queue.

```java
static class DijkstraHeap {
    private List<HeapNode> heapList;
    private Map<Vertex, Integer> vertexToIndex;

    static class HeapNode {
        Vertex vertex;
        int distance; // priority of the vertex

        HeapNode(Vertex vertex, int distance) {
            this.vertex = vertex;
            this.distance = distance;
        }
    }

    public DijkstraHeap() {
        this.heapList = new ArrayList<>();
        this.vertexToIndex = new HashMap<>();
    }
}
```

Listing 17: DijkstraHeap.java - Core Structure and HeapNode

4.3.2 Key Heap Operations

The DijkstraHeap implements standard binary min-heap operations, adapted to work with HeapNode objects and the vertexToIndex map. The implementation involves index tracking updates to both heapList and vertexToIndex.[6]

insert(Vertex vertex, int distance): A new HeapNode gets created and added to the end of heapList. The vertexToIndex map is updated with the new vertex and its index. Then, heapifyUp is called to restore the heap property by bubbling the new element up to its correct position.

```java
public void insert(Vertex vertex, int distance) {
    if (vertexToIndex.containsKey(vertex)) {
        return;
    }
    HeapNode newNode = new HeapNode(vertex, distance);
    heapList.add(newNode);
    int currentIndex = heapList.size() - 1;
    vertexToIndex.put(vertex, currentIndex);
    heapifyUp(currentIndex);
}
```

Listing 18: DijkstraHeap.java - insert Method (Conceptual)

extractMin(): The node with the minimum distance (the root of the heap, at index 0) is extracted. To maintain the heap structure, the last element in heapList is moved to the root. The vertexToIndex map is updated for the moved element and the extracted element is removed from it. heapifyDown is then called on the root to restore the heap. [14] The Vertex from the extracted root node is then returned.

```java
public Vertex extractMin() {
    if (isEmpty()) {
```

```java
        throw new NoSuchElementException("Heap is empty");
    }
    HeapNode minNode = heapList.get(0);
    Vertex minVertex = minNode.vertex;

    HeapNode lastNode = heapList.remove(heapList.size() - 1);
    vertexToIndex.remove(minVertex);

    if (!heapList.isEmpty()) {
        heapList.set(0, lastNode);
        vertexToIndex.put(lastNode.vertex, 0);
        heapifyDown(0);
    }
    return minVertex;
}
```

Listing 19: `DijkstraHeap.java` - `extractMin` Method (Conceptual)

`decreaseKey(Vertex vertex, int newDistance)`: The `vertexToIndex` map is used to find the current index of the `vertex` in `heapList`. If the `newDistance` is less than the current distance the node's distance get updated. `heapifyUp` is then called from the updated node's index to restore the heap property [14].

```java
public void decreaseKey(Vertex vertex, int newDistance) {
    if (!vertexToIndex.containsKey(vertex)) {
        /
        return;
    }
    int index = vertexToIndex.get(vertex);
    HeapNode node = heapList.get(index);

    if (newDistance < node.distance) {
        node.distance = newDistance;
        heapifyUp(index);
    }
}
```

Listing 20: `DijkstraHeap.java` - `decreaseKey` Method (Conceptual)

Heapify Operations (`heapifyUp` and `heapifyDown`):

- `heapifyUp(int index)`: Starting from the `index`, if the node at `index` is smaller than its parent, they get swapped. This process continues recursively all the way up to the root until the heap property is restored or the root is reached. `vertexToIndex` is updated during swaps [14].
- `heapifyDown(int index)`: Starts from `index`, if the node is larger than one of its children, it gets swapped with the lower child. This continues down the heap until the node is smaller than both its children. `vertexToIndex` is updated during swaps [14].

A helper `swap(int i, int j)` method is used by heapify operations to exchange two elements in `heapList` and update their index indices in `vertexToIndex` [14].

```java
private void swap(int i, int j) {
    HeapNode temp = heapList.get(i);
    heapList.set(i, heapList.get(j));
    heapList.set(j, temp);

    vertexToIndex.put(heapList.get(i).vertex, i);
```

```java
    vertexToIndex.put(heapList.get(j).vertex, j);
}

private void heapifyUp(int index) {
    int parentIndex = (index - 1) / 2;
    while (index > 0 && heapList.get(index).distance < heapList.get(parentIndex).
    distance) {
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}
// heapifyDown is similarly implemented by comparing with children and swapping.
```

Listing 21: `DijkstraHeap.java` - `swap` and `heapifyUp` (Conceptual)

4.3.3 (Algorithm Steps Summary)

1. **Initialisation**:
   - For every vertex `v` in the graph: `distances.put(v, Integer.MAX_VALUE); predecessors.put(v, null);`
   - `distances.put(sourceVertex, 0);`
   - Add `sourceVertex` (with its distance 0) to the priority queue structure.
2. **Main Loop**: While the priority queue structure is not empty:
   a. **Extract Minimum**: Remove the vertex `u` from the priority queue that has the smallest distance value.
   b. **Mark as Visited**: If `u` has already been visited, skip it. (This handles cases where `PriorityQueue` might have stale entries if not using a direct decrease key). Otherwise add `u` to `visitedVertices`.
   c. **Relax Edges**: For each neighbour `v` of `u` (with edge weight `w_uv`):
      - If `v` has not been visited:
        – Calculate `newDistance = distances.get(u) + w_uv;`
        – If `newDistance < distances.get(v)`:
          ∗ This is a shorter path to `v` Update it:
          ∗ `distances.put(v, newDistance);`
          ∗ `predecessors.put(v, u);`
          ∗ Update `v`'s entry in the priority queue with `newDistance`. (How this is done is by readding to Java's PQ, or calling 'decreaseKey' of the custom heap)[14].

4.3.4 Reflection

The custom `DijkstraHeap` approach, like the Java `PriorityQueue` method, achieves time complexity of $O((V + E)\log V)$ for Dijkstra's algorithm. The advantage of a custom heap with an explicit `decreaseKey` operation is the avoidance of "stale" entries in the priority queue [14]. When using Java's `PriorityQueue` the updates are handled by adding new entries, and potentially that would lead to multiple entries for the same node.

The `vertexToIndex` map is the cornerstone of an efficient `decreaseKey` operation, allowing the position of a vertex in the heap to be found in $O(1)$ time after when the actual key gets updated and the `heapifyUp` operation would take $O(\log V)$ time. Without this map, finding the vertex to update its key would take $O(V)$ time, making a negative trade off [14].

The main trade-off is the implementation complexity. Designing and implementing a custom heap significantly requires more caution than using a pre-built library class like `java.util.PriorityQueue`. This approach is often preferred when performance is crucial and the overhead of stale entries in a standard priority queue (or the lack of a decrease-key operation) becomes a bottleneck issue.[2]

---

5 Conclusion

The choice of data structure for Q and the method of selecting the minimum-distance vertex significantly affect the algorithm's efficiency. All three implementations successfully solve the single-source shortest path problem for graphs with non-negative weights and passed the graphML parsing tests.

6 References

[1] GeeksforGeeks. Dijkstra's Algorithm for Adjacency List Representation (Greedy Algo - 8). Retrieved from https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/

[2] GeeksforGeeks. Dijkstra's Shortest Path Algorithm (Greedy Algo - 7). Retrieved from https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[3] Baeldung. Dijkstra's Algorithm in Java. Retrieved from https://www.baeldung.com/java-dijkstra

[4] Spencer, J. Dijkstra's Algorithm Lecture Notes. Retrieved from https://cs.nyu.edu/~spencer/fundalg17/dijkstra.pdf

[5] W3Schools. Dijkstra's Algorithm in Data Structures. Retrieved from https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php

[6] GeeksforGeeks. Binary Heap. Retrieved from https://www.geeksforgeeks.org/binary-heap/

[7] Scaler. Dijkstra's Algorithm in Java. Retrieved from https://www.scaler.com/topics/dijkstras-algorithm-java/

[8] Codecademy. Dijkstra's Algorithm - Java Cheatsheet. Retrieved from https://www.codecademy.com/learn/graph-data-structures-java/modules/dijkstras-algorithm-java/cheatsheet

[9] Software Testing Help. Dijkstra's Algorithm in Java. Retrieved from https://www.softwaretestinghelp.com/dijkstras-algorithm-in-java/

[10] Alwayswannaly. Dijkstra's Algorithm Implementation Essentials. Retrieved from https://medium.com/@alwayswannaly/dijkstras-algorithm-implementation-essentials-c23db1d687b6

[11] DataCamp. Dijkstra's Algorithm in Python. Retrieved from https://www.datacamp.com/tutorial/dijkstra-algorithm-in-python

[12] Zhao, H. Dijkstra's Algorithm Java Implementation. Retrieved from https://github.com/zhaohuabing/shortest-path-weighted-graph-dijkstra-java/blob/master/src/main/java/com/zhaohuabing/Graph.java

[13] Princeton University. IndexMinPQ.java – Algorithms, 4th Edition. Retrieved from https://algs4.cs.princeton.edu/24pq/IndexMinPQ.java.html

[14] Gen AI LLM Assistants, Explanations, Coding Question, Idea Generation, Code Comments, Debugging, Document Formatting - OpenAI, ChatGPT 4.1; Google, Gemini 2.0 Flash; Google, Gemini 2.5 Pro Preview.