Graph Processing and Knapsack Problem

Ryan Davis

Ryan.Davis3@Marist.edu

December 7, 2024

This document provides an analysis and implementation of graph processing and knapsack problems. The graph section covers the representation and processing of weighted directed graphs, with Single Source Shortest Path (SSSP) calculations using the Bellman-Ford algorithm. The knapsack problem uses a greedy approach for optimizing spice allocation to maximize value. Each section discusses the algorithm's implementation, efficiency, and theoretical time complexity.

Contents

1 Graph Processing

1.1 Graph Representation

The graph is represented using vertices and edges, where each vertex maintains a list of outgoing edges.

```java
 1 static class Vertex {
 2     String id;
 3     List<Edge> edges;
 4 
 5     Vertex(String id) {
 6         this.id = id;
 7         this.edges = new ArrayList<>();
 8     }
 9 
10     void addEdge(Vertex to, int weight) {
11         edges.add(new Edge(this, to, weight));
12     }
13 }
14 
15 static class Edge {
16     Vertex from, to;
17     int weight;
18 
19     Edge(Vertex from, Vertex to, int weight) {
20         this.from = from;
21         this.to = to;
22         this.weight = weight;
23     }
24 }
```

Listing 1: Vertex and Edge Classes

Explanation:

- **Line 1:** The ‘Vertex‘ class represents a node in the graph.
- **Line 2:** Declares the ‘id‘ field to store the unique identifier for the vertex.
- **Line 3:** Declares a list to store all edges originating from the vertex.
- **Lines 5-7:** The constructor initializes the vertex’s ID and creates an empty list of edges.
- **Lines 9-10:** The ‘addEdge‘ method adds an edge from the current vertex to another, with a specified weight.
- **Line 12:** The ‘Edge‘ class represents a connection between two vertices.
- **Line 13:** Declares the ‘from‘ and ‘to‘ fields to store the source and destination vertices of the edge.
- **Line 14:** Declares the ‘weight‘ field to store the weight of the edge.
- **Lines 16-18:** The constructor initializes the source, destination, and weight of the edge.

Overview: This implementation uses a ‘Vertex‘ class to store identifiers and outgoing edges and an ‘Edge‘ class to represent weighted connections between vertices.

2 Single Source Shortest Path (SSSP)

The SSSP algorithm computes shortest paths from a source vertex to all other vertices in a graph.

```java
public static void runSSSP(Map<String, Vertex> graph, String sourceId) {
    Map<String, Integer> distances = new HashMap<>();
    Map<String, String> predecessors = new HashMap<>();

5     for (String vertexId : graph.keySet()) {
6         distances.put(vertexId, Integer.MAX_VALUE);
7         predecessors.put(vertexId, null);
8     }
9     distances.put(sourceId, 0);
10
11    for (int i = 1; i < graph.size(); i++) {
12        for (Vertex vertex : graph.values()) {
13            for (Edge edge : vertex.edges) {
14                int newDist = distances.get(vertex.id) + edge.weight;
15                if (newDist < distances.get(edge.to.id)) {
16                    distances.put(edge.to.id, newDist);
17                    predecessors.put(edge.to.id, vertex.id);
18                }
19            }
20        }
21    }
22
23    for (Vertex vertex : graph.values()) {
24        for (Edge edge : vertex.edges) {
25            if (distances.get(vertex.id) + edge.weight < distances.get(edge.to.id)) {
26                System.out.println("Graph contains a negative-weight cycle.");
27                return;
28            }
29        }
30    }
31 }
```

Listing 2: SSSP Algorithm

Explanation:

- **Line 1:** The ‘runSSSP‘ method takes a graph and a source vertex ID as input.
- **Lines 2-3:** Two maps are initialized: one for distances from the source vertex and one for tracking predecessors.
- **Lines 5-7:** All vertices are initialized with an infinite distance, except the source vertex, which is set to 0.
- **Lines 9-15:** The Bellman-Ford algorithm performs $|V| - 1$ relaxation passes over all edges in the graph. If a shorter path is found, the distance and predecessor are updated.
- **Lines 17-21:** After relaxation, the algorithm checks for negative-weight cycles by attempting to relax the edges once more.

Overview: The Bellman-Ford algorithm ensures accurate shortest path calculations even for graphs with negative weights. By maintaining maps for distances and predecessors, the algorithm efficiently reconstructs paths while detecting negative-weight cycles.

3 Knapsack Problem

The knapsack problem is solved using a greedy algorithm to maximize value while adhering to capacity constraints.

```
1 public static void processSpice(String filename) {
2     List<Spice> spices = new ArrayList<>();
3     List<Knapsack> knapsacks = new ArrayList<>();
4
5     spices.sort((a, b) -> Double.compare(b.unitPrice, a.unitPrice));
6
7     for (Knapsack knapsack : knapsacks) {
8         int remainingCapacity = knapsack.capacity;
9         double totalValue = 0;
```

```java
10         List<String> contents = new ArrayList<>();
11 
12         for (Spice spice : spices) {
13             int takeQuantity = Math.min(remainingCapacity, spice.quantity);
14             totalValue += takeQuantity * spice.unitPrice;
15             remainingCapacity -= takeQuantity;
16 
17             if (takeQuantity > 0) {
18                 contents.add(takeQuantity + " scoop(s) of " + spice.name);
19             }
20         }
21         System.out.println("Knapsack worth: " + totalValue + " quatloos.");
22     }
23 }
```

Listing 3: Knapsack Implementation

Explanation:

- **Lines 1-2:** Initializes lists to store spices and knapsacks.
- **Line 4:** Sorts spices in descending order of unit price for optimal value selection.
- **Lines 6-17:** For each knapsack, iterates through spices, selecting as much as possible without exceeding capacity, and updates the total value and remaining capacity.
- **Line 18:** Outputs the total value and contents of the knapsack.

Overview: This greedy algorithm efficiently solves the fractional knapsack problem by prioritizing items with higher unit value. Sorting ensures that the most valuable items are considered first.

4 Time Complexity Analysis

4.1 Graph Processing (Bellman-Ford Algorithm)

Best Case: $O(E)$ The best-case time complexity of the Bellman-Ford algorithm is $O(E)$. This occurs when:

- No relaxation is required because the initial distances to vertices are already the shortest paths from the source vertex.
- The algorithm performs a single pass through all edges to verify the correctness of distances.
- Efficiency is particularly evident in sparse graphs with fewer edges, as the algorithm only processes each edge once.

In this scenario, the time complexity is directly proportional to the number of edges, as the algorithm terminates after efficiently traversing all edges without performing further relaxations.

Average Case: $O(V \cdot E)$ The average-case time complexity remains $O(V \cdot E)$ because:

- The algorithm requires multiple passes ($|V| - 1$) through all edges to ensure all shortest paths are relaxed.
- Practical scenarios often mirror the worst case, especially in dense graphs with many edges.
- Performance depends on both the number of vertices (V) and edges (E).

Worst Case: $O(V \cdot E)$ The worst-case time complexity also stands at $O(V \cdot E)$, which occurs when:

- The algorithm needs $|V| - 1$ passes to relax all edges and one additional pass to check for negative weight cycles.
- Negative weight cycles are present, necessitating the algorithm to iterate through all vertices and edges to detect these cycles.
- Dense graphs with a high number of vertices and edges exacerbate the computational load.

The worst-case scenario highlights the algorithm's resilience in handling complex graphs, but it also underscores the potential for inefficiency in dense or negatively weighted graphs.

Summary of Bellman-Ford Time Complexities:

- **Best Case:** $O(E)$
- **Average Case:** $O(V \cdot E)$
- **Worst Case:** $O(V \cdot E)$

4.2 Knapsack Problem

Sorting Spices: Sorting n spices based on their unit price (value per unit weight) in descending order uses TimSort. This has a time complexity of:

$$O(n \log n)$$

- **Best Case:** $O(n)$, if the spices are already sorted.
- **Average Case:** $O(n \log n)$, as TimSort performs optimally on average.
- **Worst Case:** $O(n \log n)$, when the spices are in reverse order.

Knapsack Filling: For each knapsack, the algorithm iterates over the sorted list of n spices. If there are k knapsacks, the overall complexity of filling the knapsacks is:

$$O(k \cdot n)$$

- **Best Case:** $O(n)$, if only one knapsack exists and it can be filled with the first item.
- **Average Case:** $O(k \cdot n)$, as it iterates over all spices for all knapsacks.
- **Worst Case:** $O(k \cdot n)$, if k is large, and all spices are required to fill the knapsacks.

Total Time Complexity for Knapsack Problem: The total time complexity is the sum of sorting the spices and filling the knapsacks:

$$O(n \log n) + O(k \cdot n) = O(n \log n + k \cdot n)$$

- **Best Case:** $O(n + n) = O(n)$, if the spices are pre-sorted and only one knapsack is used.
- **Average Case:** $O(n \log n + k \cdot n)$, for typical scenarios with multiple knapsacks and unordered spices.
- **Worst Case:** $O(n \log n + k \cdot n)$, if the spices are in reverse order and there are many knapsacks.