CS-1203 – Monsoon 2023 – Assignment 3

Kudakwashe Chakanyuka

kudakwashe.chakanyuka_ug25@ashoka.edu.in

November 9, 2023

Question 2

Question: According to Dijkstra's algorithm, will this idea work for a directed graph?

To begin, yes, Dijkstra's algorithm can be used for directed graphs and the algorithm itself is designed to work with both directed and undirected graphs. A few important considerations when applying Dijkstra's algorithm to directed graphs are as follows:

1. Edge Direction:
- In a directed graph, edges have a direction, meaning they go from one vertex to another. Dijkstra's algorithm works as long as the edge weights represent the cost of traveling from one vertex to another, regardless of direction.

2. Graph Representation:
- Dijkstra's algorithm is often implemented using an adjacency matrix or an adjacency list to represent the graph. This is done to ensure that the graph representation accounts for the direction of edges.

3. Initialization:
- There is a need to initialize the distance values for each vertex to infinity and the distance of the source vertex to itself to 0. This therefore remains the same as in the undirected case.

4. Update Conditions:
- When updating the distance values during the relaxation step, there is need to consider the direction of the edges. We only update the distance if the current path through the vertex being relaxed is shorter than the previously known distance.

Moving on to how Dijkstra's algorithm works for a directed graph:

1. - Initialization:
- Setting the distance values to infinity for all vertices and 0 for the source vertex.
- Creating a priority queue or using a suitable data structure to keep track of vertices and their current distances.

2. Main Loop:

3. Repeating the following procedures until all vertices are processed:

a. Extracting the vertex with the minimum distance from the priority queue.
b. For each neighboring vertex of the extracted vertex, we therefore update its distance if the current path through the extracted vertex is shorter.

4. Termination:
- Once all vertices are processed or the destination is reached, the algorithm then terminates.
- Dijkstra's algorithm can be labeled as greedy, it always chooses the vertex with the minimum distance at each step. This also works well for graphs without negative edge weights and if negative edge weights are present that is when the user can consider using the Bellman-Ford algorithm.

Question 2

Question: Time complexity $O(n^2)$, auxiliary space O(n). How the time complexity can be improved? What are the relevant data structures?

The time complexity of Dijkstra's algorithm can be improved from $O(V^2)$ to O((V + E) log V) by using a priority queue or a min-heap data structure to efficiently select the vertex with the minimum distance.

The improvements are as follows:

1. Using Priority Queue/Min-Heap:
- Instead of iterating through all vertices to find the minimum distance vertex in each iteration, the user can use a priority queue or a min-heap to efficiently extract the minimum distance vertex.
- This reduces the time complexity of finding the minimum from O(V) to O(log V).

2. Adjacency List Representation:
- Using an adjacency list instead of an adjacency matrix. For sparse graphs, where the number of edges E is much less than V^2, this can significantly reduce the number of operations needed to iterate over neighbors.
- With these improvements, the overall time complexity becomes O((V + E)log V) where V is the number of vertices and E is the number of edges.

Relevant Data Structures:

1. Priority Queue:
- A priority queue is a data structure that maintains a set of elements with associated priorities and allows efficient extraction of the element with the minimum priority. In the context of Dijkstra's algorithm, the priorities are the distances from the source vertex.

2. Min-Heap:
- A min-heap is a specialized form of a priority queue where the element with the minimum key is always at the root. This allows for efficient extraction of the minimum element.

3. Adjacency List:
- Representing the graph as an adjacency list is more efficient than using an adjacency matrix, especially for sparse graphs. Each vertex maintains a list of its neighbors along with the corresponding edge weights.

To conclude, from the above-mentioned procedures, we can therefore achieve the improved time complexity for Dijkstra's algorithm which also makes it more suitable for large graphs where the number of edges is much smaller than the number of vertices.