CS204: Design and Analysis of Algorithms

220001008, 220001028, 220001029

February 12, 2024

1 Shortest Path Problem

The shortest path problem involves finding the minimum weight path between two nodes in a graph. In a weighted graph, where each edge has an associated numerical value (weight), the goal is to determine the path with the lowest cumulative weight from a specified source node to a target node.
Given a weighted and directed graph $G = (V, E)$ and $W : E \longrightarrow R$, the cost of a path $P_{(v_0,v_k)}$ from vertex v_0 to v_k is defined as

$$W(P_{(v_0,v_k)}) = \sum_{i=0}^{k-1} W(v_i, v_k)$$

Shortest Path $S(v_0, v_k)$ is defined as

$$S(v_0, v_k) = \begin{cases} \min W(P_{(v_0,v_k)} & \text{if } \exists \text{ a path between } \mathrm{v}_0 \text{ and } v_k \\ \infty & \text{otherwise} \end{cases}$$

2 How to solve the Shortest Path Problem ?

There are several ways to solve the shortest path problem in a graph. Here are some common approaches:

- **Dijkstra's Algorithm:** A greedy algorithm that finds the shortest path from a source node to all other nodes in non-negative weighted graphs.
- **Bellman-Ford Algorithm:** Suitable for graphs with negative edge weights, it finds the shortest paths from a single source node to all other nodes.
- **Floyd-Warshall Algorithm:** Computes the shortest paths between all pairs of nodes in a graph, handling both positive and negative edge weights.

The choice of algorithm depends on the specific characteristics of the graph, such as edge weights, graph density, and the presence of negative weights.

3 Dijkstra's Algorithm

Below is the implementation of the Dijkstra's Algorithm:

Mark initial distance from the source as infinite.
Create an empty priority_queue PQ. Each item of PQ is a pair $(\text{weight}, \text{vertex})$. Weight is used as the first item of the pair as the first item is by default used to compare two pairs.
Insert source vertex into PQ and set its distance as 0.
while PQ is not empty **do**
    Extract minimum distance vertex u from PQ.
    for all adjacent vertices v of u **do**
        if $\text{dist}[v] > \text{dist}[u] + \text{weight}(u, v)$ **then**
            Update distance of v: $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$.
            Insert v into PQ (even if v is already there).
        end if
    end for
end while
Loop through the dist[] array to print the shortest paths from source to all the vertices.

3.1 This is a Greedy algorithm

Dijkstra's algorithm is considered a greedy algorithm because it always chooses the vertex with the smallest distance from the source and then updates the distances of the adjacent vertices. This approach is considered greedy because it makes the locally optimal choice at each step with the hope of finding a globally optimal solution. The algorithm does not reconsider its choices, which is a characteristic of greedy algorithms.

3.2 How to check the correctness of Dijkstra's Algorithm ?

Dijkstra's algorithm for finding the shortest paths in a graph can be analyzed using loop invariants. A loop invariant is a property that holds true before and after each iteration of a loop. Here is how the concept of a loop invariant can be applied to Dijkstra's algorithm:

1. **Initialization:** Before the loop starts, the loop invariant holds true. In the context of Dijkstra's algorithm, this could be the correct initialization of distances and predecessors.

2. **Maintenance:** Assuming the loop invariant is true before an iteration, the algorithm maintains it during the iteration. In Dijkstra's algorithm, this corresponds to the correct relaxation step, updating distances and predecessors.

3. **Termination:** When the loop terminates, the loop invariant guarantees the correctness of the algorithm's output. In Dijkstra's algorithm, the loop terminates when all vertices are included in the shortest path tree.

Let's formalize the loop invariant for Dijkstra's algorithm:
Loop Invariant: At the start of each iteration of the main loop, the distances of all vertices in the priority queue are the shortest distances from the source vertex to those vertices.

By ensuring the loop invariant at each step, we can argue the correctness of Dijkstra's algorithm. This invariant guarantees that the algorithm correctly computes and updates the shortest paths during its execution.

3.3 Time and Space Analysis of Dijkstra's Algorithm

3.3.1 Time Complexity Analysis:

Let $|E|$ and $|V|$ be the number of edges and vertices in the graph, respectively. Then the time complexity is calculated:

Using adjacency matrix:

Adding all $|V|$ vertices to Q takes $O(|V|)$ time.

Removing the node with minimal *dist* takes $O(|V|)$ time, and we only need $O(1)$ to recalculate *dist*[u] and update Q. Since we use an adjacency matrix here, we'll need to loop for $|V|$ vertices to update the *dist* array.

The time taken for each iteration of the loop is $O(|V|)$, as one vertex is deleted from Q per loop.

Thus, the total time complexity becomes $O(|V|) + O(|V|) \times O(|V|) = O(|V|^2)$.

Using adjacency list:

It takes $O(|V|)$ time to construct the initial priority queue of $|V|$ vertices.

With adjacency list representation, all vertices of the graph can be traversed using BFS. Therefore, iterating over all vertices' neighbors and updating their *dist* values over the course of a run of the algorithm takes $O(|E|)$ time.

The time taken for each iteration of the loop is $O(|V|)$, as one vertex is removed from Q per loop.

The binary heap data structure allows us to *extract-min* (remove the node with minimal *dist*) and update an element (recalculate *dist*[u]) in $O(\log |V|)$ time.

Therefore, the **time complexity** becomes $O(|V|) + O(|E| \times \log |V|) + O(|V| \times \log |V|)$, which is $O((|E| + |V|) \times \log |V|)$

3.3.2 Space Complexity Analysis:

Distances Array: The distances array (`dist[]`) stores the current shortest distances from the source vertex to all other vertices. It requires $O(V)$ space.

Priority Queue: The priority queue is used to efficiently extract the minimum distance node during each iteration. Depending on the implementation, the space complexity of the priority queue can vary. A binary heap-based priority queue requires $O(V)$ space, while a Fibonacci heap can use $O(V + E)$ space.

Visited Array: A boolean array (`visited[]`) is often used to keep track of visited vertices. It requires $O(V)$ space.

Predecessor Array: A predecessor array (`prev[]`) stores the predecessor of each vertex in the shortest path. It also requires $O(V)$ space.

Total Space Complexity: Combining the space requirements of these data structures, the overall space complexity of Dijkstra's algorithm with a binary heap-based priority queue is $O(V)$, while with a Fibonacci heap-based priority queue, it is $O(V + E)$.

3.4 Example 1

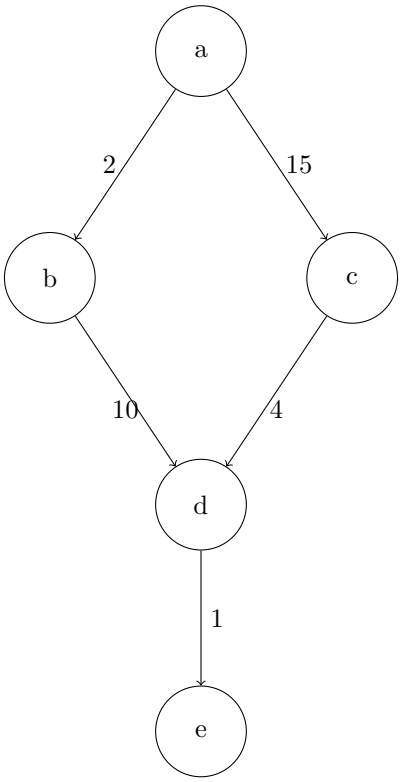
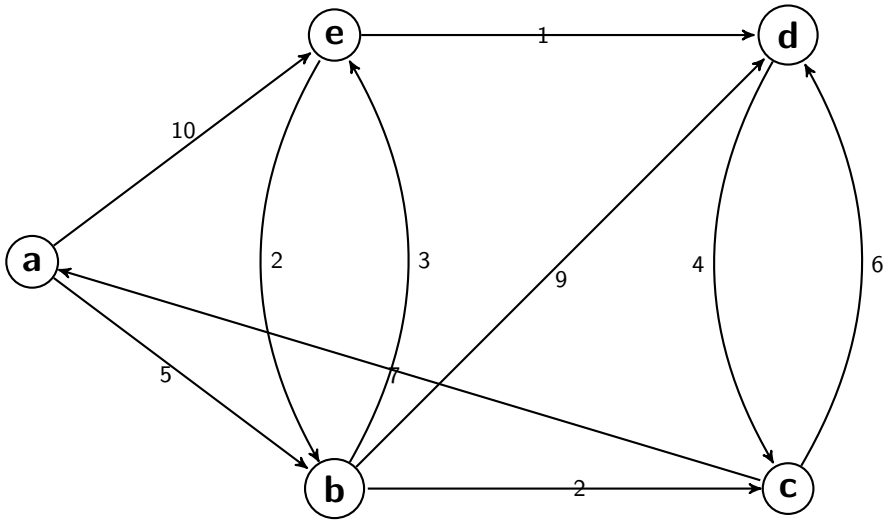


Figure 1: Directed Graph

	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	2	15	∞	∞
2	0	2	15	12	∞
3	0	2	15	12	13
4	0	2	15	12	13

Table 1: Dijkstra's Algorithm Iterations

3.5 Example 2



Directed Graph

	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	5	∞	∞	10
2	0	5	7	14	8
3	0	5	7	13	8
4	0	5	7	9	8

Table 2: Dijkstra's Algorithm Iterations