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 $W : E \longrightarrow R$ (R represents Real Numbers), the cost of a path $P_{(v_0,v_k)}$ from vertex v_0 to v_k is defined as

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

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

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

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:

Input: Graph G, starting vertex s
Output: Shortest distances from s to all other vertices
1 Make array for storing distances of each vertex from the starting vertex
  Make array for storing the parent of each of the vertices
  for *each vertex* $v \in G.V$ **do**
2 | $v.d \leftarrow$ distance between s (source vertex) and v
  | $v.d \leftarrow \infty$
  | v.pie $\leftarrow$ null (parent)
3 $s.d \leftarrow 0$ Distance from source vertex to itself
  Priority Queue $Q \leftarrow G.V$
  s.pie $\leftarrow \phi$
  while Q *is not empty* **do**
4 | $u \leftarrow Q$.extractMin()
  | $s \leftarrow s \cup \{u\}$
  | Mark u as visited
5 | **for** *each neighbor* v *of* u **do**
6 | | **if** v *does not belong to* s **then**
7 | | | **if** $v.d > u.d + w(u, v)$*(edge weight between vertices* u *and* v*)* **then**
8 | | | | $v.d \leftarrow u.d + w(u, v)$
  | | | | v.pie $\leftarrow u$
9 **return** *Shortest distances array*

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.

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 :

Initialization: Initializing the distances and priority queue takes $O(|V|)$ time

Main Loop:

1. **ExtractMin Operation**:Extracting the minimum distance vertex from the priority queue takes $O(log(|V|))$ time
2. **Neighbour Exploration**:For each neighbor of the current vertex, the algorithm explores the neighbors and updates their tentative distances. In the worst case, each edge is considered once, resulting in $O(|V|)$ time for the loop.

The main loop iterates V times (once for each vertex), and in each iteration, there is a $O(log(|V|))$ operation (extracting the minimum distance vertex) and a $O(V)$ operation (neighbor exploration).

Total Time Complexity: The total time complexity of the main loop is $O(V.(log(|V|) + V))$. and the time complexity can be approximated as $O(|V|^2)$

Using adjacency list:

We can observe that the statements in the inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has `extractMin()` operation which takes $O(\log V)$ time. So overall time complexity is $O((E+V)\cdot \log V) = O(E \log V)$.

Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to $O(E + V \log V)$ using Fibonacci Heap. The reason is that Fibonacci Heap takes $O(1)$ time for `extractMin` operation while Binary Heap takes $O(log(|V|))$ time.

In summary, Dijkstra's algorithm using an adjacency list representation is efficient with a time complexity that scales well, particularly in graphs where the number of edges is significantly less than the square of the number of vertices.

3.3.2 Space Complexity Analysis:

$|V|$: total number of vertices

Distances Array: The distances array 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 $O(log(|V|))$ 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 $O(|V|)$

3.4 Example 1

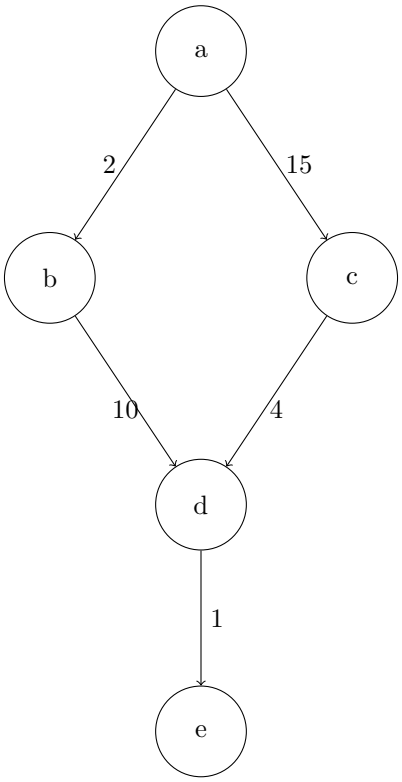
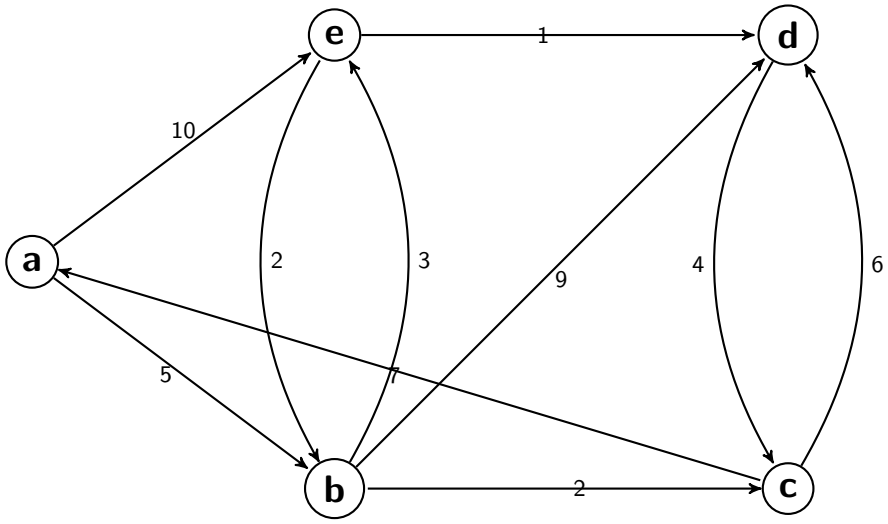


Figure 1: Directed Graph

	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