CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Dec

Scribe: Soham Chatterjee

sohamchatterjee999@gmail.com
Website: sohamch08.github.io

Contents

Chapter 1 Dijkstra Algorithm with Data Structures Page 3

1.1 Dijkstra Algorithm 3
1.2 Data Structure 1: Linear Array 5
1.3 Data Structure 2: Min Heap 5
1.4 Amortized Analysis 5
1.5 Data Structure 3: Fibonacci Heap 5
1.5.1 Inserting Node 5

Chapter 2 Kruskal Algorithm with Data Structure Page 6

2.1 Kruskal Algorithm 6
2.2 Data Structure 1: Array 6
2.3 Data Structure 2: Left Child Right Siblings Tree 6
2.4 Data Structure 3: Union Find 6
2.4.1 Analyzing the Union-Find Data-Structure 6

Chapter 1

Dijkstra Algorithm with Data Structures

Minimum Weight Path
Input: Directed Graph $G = (V, E)$, $s \in V$ is source and $W = \{w_e \in \mathbb{Z}_+ : e \in E\}$
Question: $\forall\, v \in V - \{s\}$ find minimum weight path $s \rightsquigarrow v$.

This is the problem we will discuss in this chapter. In this chapter we will often use the term 'shortest distance' to denote the minimum weight path distance. One of the most famous algorithm for finding out minimum weight paths to all vertices from a given source vertex is Dijkstra's Algorithm

1.1 Dijkstra Algorithm

We will assume that the graph is given as adjacency list. Dijkstra Algorithm is basically dynamic programming. Suppose $\delta(v)$ is the shortest path distance from $s \rightsquigarrow v$. Then we have the following relation:

$$\delta(v) = \min_{u:(u,v)\in E} \{\delta(u) + e(u, v)\}$$

And suppose for any vertex $v \in V - \{s\}$, $dist(v)$ be the distance from s estimated by the algorithm at any point. This is why Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with minimum shortest-path estimate and estimates the distances of neighbors of u. So here is the algorithm:

Algorithm 1: Dijkstra(G, s, W)

Input: Adjacency Matrix of digraph $G = (V, E)$, source vertex $s \in V$ and weight function $W = \{w_e \in \mathbb{Z}_+ : e \in E\}$
Output: $\forall\, v \in V - \{s\}$ minimum weight path from $s \rightsquigarrow v$

```
1 begin
2     S ⟵ ∅, U ⟵ V
3     dist(s) ⟵ 0, ∀ v ∈ V − {s}, dist(v) ⟵ ∞
4     while U ≠ ∅ do
5         u ⟵ min_{u∈U} dist(u) and remove u from U
6         S ⟵ S ∪ {u}
7         for e = (u, v) ∈ E do
8             dist(v) ⟵ min{dist(v), dist(u) + w(u, v)}
```

Here below we give an example of how the Dijkstra algorithm works:

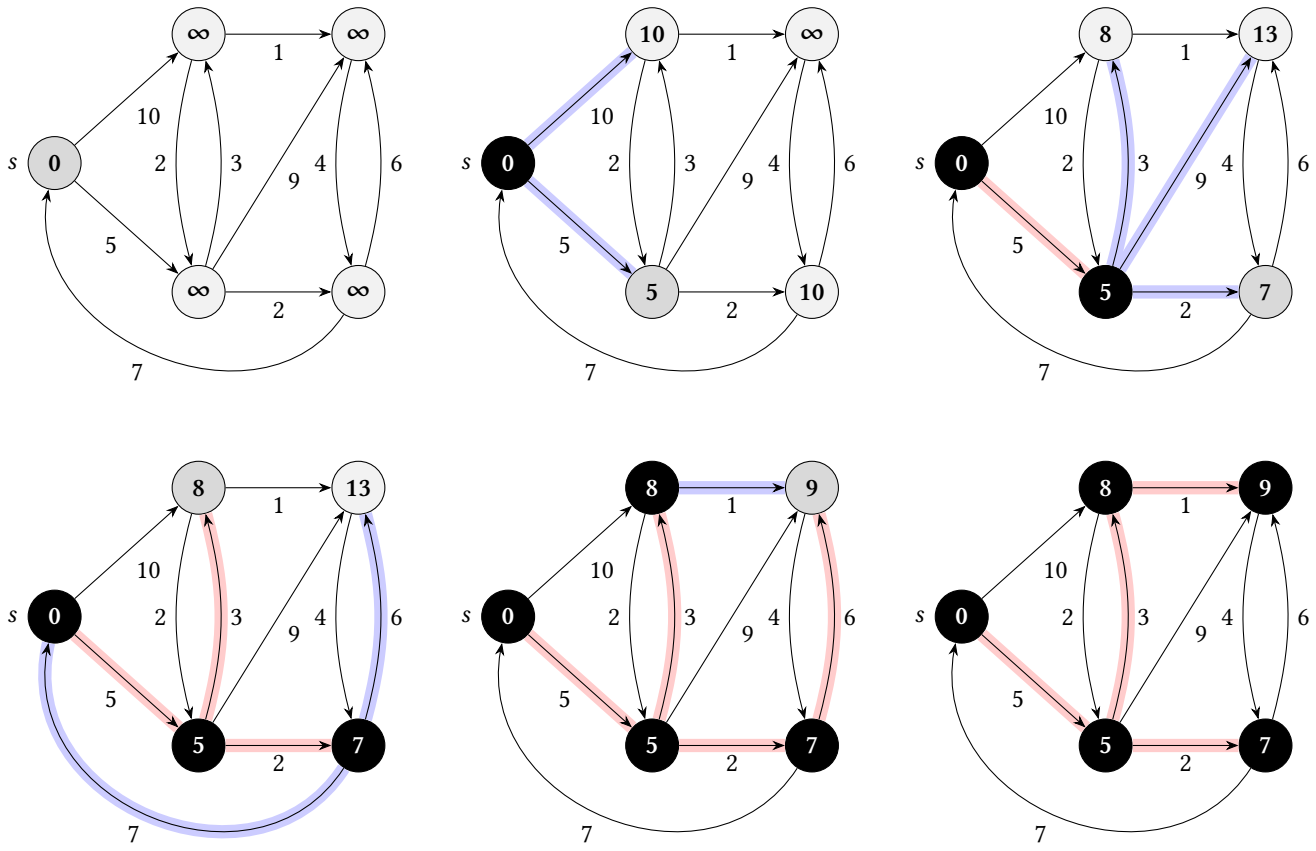


Figure 1.1: The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S and at any iteration of while loop the shaded vertex has the minimum value. At any iteration the red edges are the edges considered in minimum weight path from s using only vertices in S.

Suppose at any iteration t, let $dist_t(v)$ denotes the distance v from s calculated by algorithm for any $v \in V$ and $S^{(t)}$ denote the content of S at t^{th} iteration. In order to show that the algorithm correctly computes the distances we prove the following lemma:

Theorem 1.1.1

For each $v \in S^{(t)}$, $\delta(v) = dist_t(v)$ for any iteration t.

Proof: We will prove this induction. Base case is $|S^{(1)}| = 1$. S grows in size. Then only time $|S^{(1)}| = 1$ is when $S^{(1)} = \{s\}$ and $d(s) = 0 = \delta(s)$. Hence for base case this is correct.

Suppose this is also true for $t-1$. Let at t^{th} iteration the vertex $u \in V - S$ is picked. By induction hypothesis for all $v \in S^{(t)} - \{u\}$, $dist_t(v) = dist_{t-1}(v) = \delta(v)$. So we have to show that $dist_t(u) = \delta(u)$.

Suppose for contradiction the shortest path from $s \rightsquigarrow u$ is P and has total weight $= \delta(u) = w(P) < dist_t(u)$. Now P starts with vertices from $S^{(t)}$ by eventually leaves S. Let (x, y) be the first edge in P which leaves S i.e. $x \in S$ but $y \notin S$. By inductive hypothesis $dist_t(x) = \delta(x)$. Let P_y denote the path $s \rightsquigarrow y$ following P. Since y appears before u we have

$$w(P_y) = \delta(y) \leq \delta(u) = w(P)$$

Now

$$dist_t(y) \leq dist_t(x) + w(x, y)$$

since y is adjacent to x. Therefore

$$dist_t(y) \leq dist_t(x) + w(x, y) = \delta(y) \leq dist_t(y) \implies dist_t(y) = \delta(y)$$

Now since both $u, y \notin S^{(t)}$ and the algorithm picked up u we have $\delta(u) < dist_t(u) \leq dist_t(y) = \delta(y)$. But we can not have both $\delta(y) \leq \delta(u)$ and $\delta(u) < \delta(y)$. Hence contradiction. Therefore $\delta(u) = dist_t(u)$. Hence by mathematical induction for any iteration t, for all $v \in S^{(t)}$, $\delta(v) = dist_t(v)$. ■

Therefore by the lemma after all iterations S has all the vertices with their shortest distances from s and henceforth the algorithm runs correctly.

1.2 Data Structure 1: Linear Array

1.3 Data Structure 2: Min Heap

1.4 Amortized Analysis

1.5 Data Structure 3: Fibonacci Heap

Instead of keeping just one Heap we will no keep an array of Heaps. We will also discard the idea of binary trees. We will now use a data structure which will take the benefit of the faster time of both the data structure. I.e. The * is because in Fibonacci Heap the amortized time taken by EXTRACT-MIN is $O(\log n)$.

Since Fibonacci heap is an array of heaps there is a *rootlist* which is the list of all the roots of all the heaps in the Fibonacci heap. There is a *min-pointer* which points to the root with the minimum key. For each node in the Fibonacci heap we have a pointer to its parent and we keep 3 variables. The 3 variables are *degree*, *size* and *lost* where *lost* is a Boolean Variable. For any node x in the Fibonacci heap the $x.degree$ is the number of children x has, $x.size$ is the number of nodes in the tree rooted at x and $x.lost$ is 1 if and only if x has lost a child before. Why any node will lost a child that explanation we will give later. With this set up let's dive into the data structure.

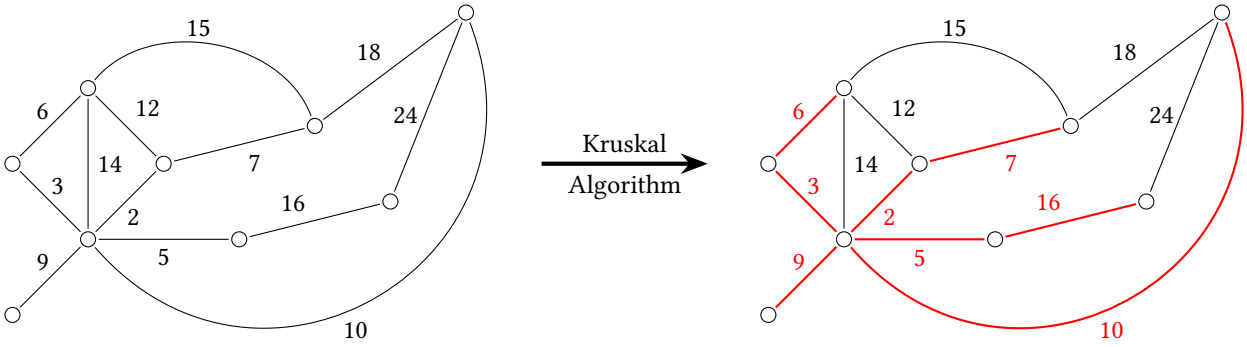
1.5.1 Inserting Node

To insert a node we call the FIB-INSERT function and in the function the algorithm initiates the node with setting up all the pointers and variables then add the node to the *rootlist*.

Chapter 2

Kruskal Algorithm with Data Structure

2.1 Kruskal Algorithm



2.2 Data Structure 1: Array

2.3 Data Structure 2: Left Child Right Siblings Tree

2.4 Data Structure 3: Union Find

2.4.1 Analyzing the Union-Find Data-Structure

We call a node in the union-find data-structure a *leader* if it is the root of the (reversed) tree.

Lemma 2.4.1
Once a node stop being a *leader* (i.e. the node in top of a tree). it can never become a leader again.

Lemma 2.4.2
Once a node stop being a leader then its rank is fixed.

Lemma 2.4.3
Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.

Lemma 2.4.4

When a node gets rank k than there are at least $\geq 2^k$ elements in its subtree.

Corollary 2.4.5

For all vertices v, $v.rank \leq \lfloor \log n \rfloor$

Corollary 2.4.6

Height of any tree $\leq \lfloor \log_2 n \rfloor$

Lemma 2.4.7

The number of nodes that get assigned rank k throughout the execution of the Union-Find data-structure is at most $\frac{n}{2^k}$.

Define $N(r)$ = #vertices with rank at least k. Then by the above lemma we have $N(r) \leq \frac{n}{2^k}$.

Lemma 2.4.8

The time to perform a single find operation when we perform union by rank and path compression is $O(\log n)$ time.

We will show that we can do much better. In fact we will show that for m operations over n elements the overall running time is $O((n+m)\log^* n)$

Lemma 2.4.9

During a single FIND(x) operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Lemma 2.4.10

At most $|Block(i)| \leq Tower(i)$ many FIND operations can pass through an element x which is in the i^{th} block (i.e. $\text{INDEX}_B(x) = i$) before $x.parent$ is no longer in the i^{th} block. That is $\text{INDEX}_B(x.parent) > i$.

Lemma 2.4.11

There are at most $\frac{n}{Tower(i)}$ nodes that have ranks in the i^{th} block throughout the algorithm execution.

Lemma 2.4.12

The number of internal jumps performed, inside the i^{th} block, during the lifetime of UNION-FIND data structure is $O(n)$.

Theorem 2.4.13

The number of internal jumps performed by the UNION-FIND data structure overall $O(n \log^* n)$.

Theorem 2.4.14

The overall time spent on m FIND operations, throughout the lifetime of a Union-Find data structure defined over n elements is $O((n+m)\log^* n)$.