CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Nov

Scribe: Soham Chatterjee

soham.chatterjee@tifr.res.in
Website: sohamch08.github.io

Contents

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:

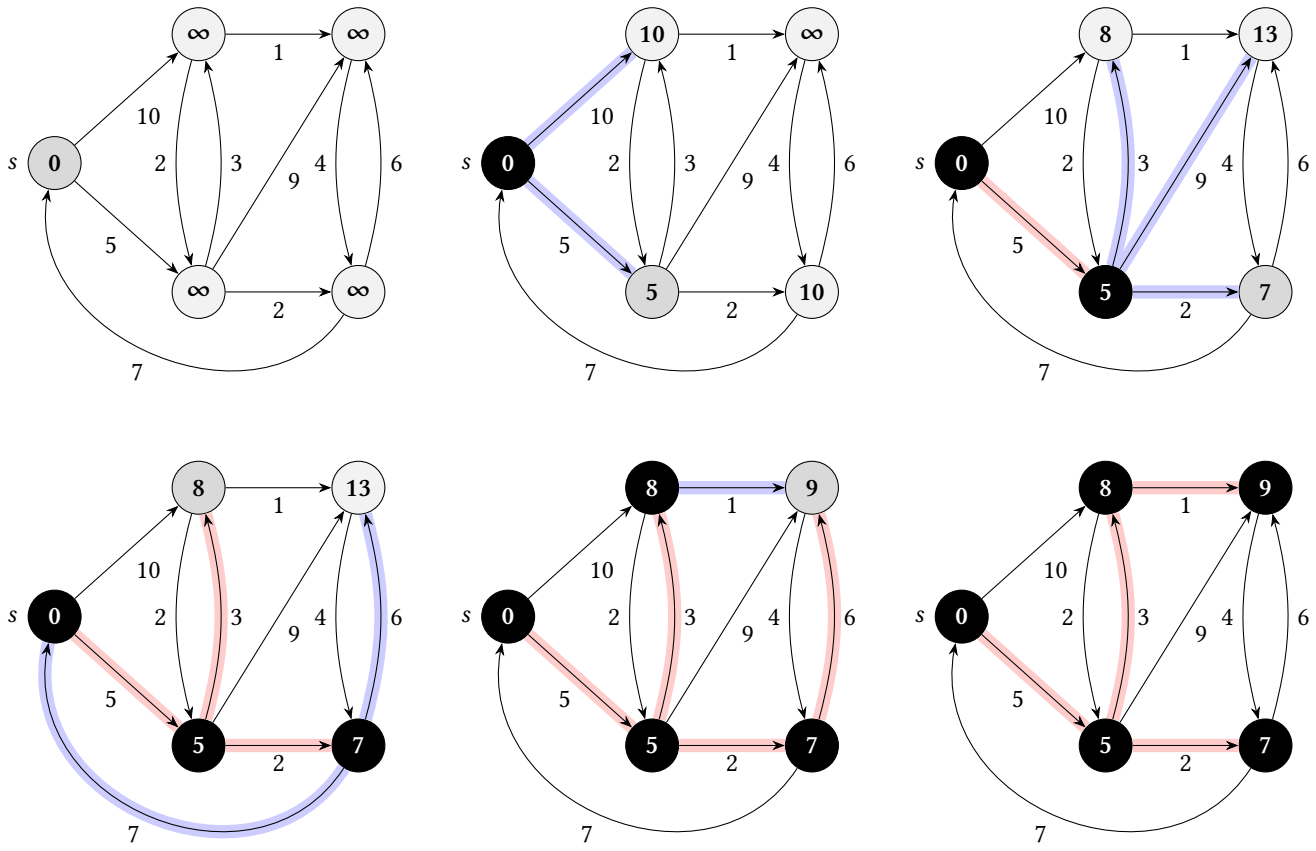

Figure 1.1: The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S and at any iteration of while loop the shaded vertex has the minimum value. At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

	EXTRACT-MIN	DECREASE-KEY
Linear Array	$O(n)$	$O(1)$
Min-Heap	$O(\log n)$	$O(\log n)$
Fibonacci Heap	$O(\log n)^*$	$O(1)^*$

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.
- $x.lost$ is 1 if and only if x has lost a child before.

Why any node will lose 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*.

Algorithm 2: FIB-CREATE-NODE(v)

```
1 x.degree ⟵ 0
2 x.parent ⟵ None
3 x.child ⟵ None
4 x.lost ⟵ 0
5 x.key ⟵ v
6 return x
```

Algorithm 3: FIB-INSERT(F, v)

```
1 x ⟵ CREATE-NODE(v)
2 if F.min == None then
3     F.rootlist ⟵ [x]
4     F.min ⟵ x
5 else
6     F.rootlist.append(x)
7     if x.key < F.min.key then
8         F.min ⟵ x
```

All of this can be done in $O(1)$ time. Therefore, to insert a node in the Fibonacci heap it takes $O(1)$ time.

1.5.2 Union of Fibonacci Heaps

To unite to Fibonacci heaps F_1 and F_2 we simply concatenate the root lists of F_1 and F_2 and then determine the new minimum node. All the operations here can be done in constant time. Hence, Fib-Union takes $O(1)$ time.

Algorithm 4: Fib-Union(F_1, F_2)

```
F ⟵ Make-Fib-Heap
F.min ⟵ F_1.min
F.rootlist ⟵ F_1.rootlist ++ F_2.rootlist
if F_2.min < F_1.min then
    F.min ⟵ F_2.min
return F
```

1.5.3 Extracting the Minimum Node

The Fib-Extract-Min function extracts the minimum node from the Fibonacci heap F and then rearranges the heap array. It works by first making a root node out of each of the minimum node's children and removing the minimum node from the rootlist. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

Algorithm 5: Fib-Extract-Min(F)

```
z ⟵ F.min
if z ≠ None then
    for x ∈ z.child do
        F.rootlist.append(x)
        x.parent ⟵ None
    Remove z from F.rootlist
    if z == z.right then
        F.min ⟵ None
    else
        F.min ⟵ z.right consolidate(F)
return z
```

Algorithm 6: Fib-Heap-Link(H, y, x)

```
Remove y from F.rootlist
y.parent ⟵ x
y.lost ⟵ 0
```

Algorithm 7: Consolidate(F)

```
Initialize array A[0, ..., D(n)] with None elements.
for x ∈ F.rootlist do
    d ⟵ x.degree
    if A[d] == None then
        A[d] ⟵ x
    while A[d] ≠ None do
        y ⟵ A[d]
        if y.key < x.key then
            Exchange x with y
        Fib-Heap-Link(F, y, x)
        A[d] ⟵ None
        d ⟵ d + 1
    A[d] ⟵ x
F.min ⟵ None
for i = 0 to D do
    if A[i] ≠ None then
        if F.min == None then
            F.rootlist ⟵ [A[i]]
            F.min ⟵ A[i]
        else
            F.rootlist.append(A[i])
            if A[i].key < F.min.key then
                F.min ⟵ A[i]
```

Here $D(n)$ denotes the maximum degree a node can have after Consolidate. The procedure Consolidate uses an auxiliary array of size A of size D which we will choose later. For each $i \leq D(n)$ it keeps a heap of degree i. And if it finds two heaps of same degree then it makes the one with higher key to be the child of the other one. The function Fib-Heap-Link does this process of linking two heaps of same degree.

Of course in order to allocate array we have to know how to calculate the upper bound for $D(n)$ on the maximum degree. We will show an upper bound of $O(\log n)$ in subsection 1.5.5.

Now in FIB-EXTRACT-MIN in each iteration of the outer for loop or inner while loop it operates on one heap in $F.rootlist$. Hence it takes $O(D(n) + \#\text{heaps in } F.rootlist)$ time.

1.5.4 Decreasing Key of a Node

In this section we will show how to decrease a key of a node in a Fibonacci heap in $O(1)$ amortized time. The FIB-DECREASE-KEY function decreases the key value of the target node then if the min-heap order the node is in is violated then we use the CASCADING-CUT function to restore the min-heap property again. These two functions operates like the following:

Algorithm 8: FIB-DECREASE-KEY(F, x, k)

```
1 if k > x.key then
2     return Error
3 x.key ⟵ k
4 y ⟵ x.parent
5 if y ≠ None and x.key < y.key then
6     CUT(F, x, y)
7     CASCADING-CUT(F, y)
8 if k < F.min.key then
9     F.min ⟵ x
```

Algorithm 9: CASCADING-CUT(F, y)

```
1 if y.parent ≠ None then
2     if y.lost == 0 then
3         y.lost ⟵ 1
4     else
5         CUT(F, y, y.parent)
6         CASCADING-CUT(F, y.parent)
```

Algorithm 10: CUT(F, x, y)

```
1 Remove x from y.child
2 y.degree ⟵ y.degree − 1
3 F.rootlist.append(x)
4 x.parent ⟵ None
5 x.lost ⟵ 0
```

After decreasing the key of the target node if the min-heap order has been violated then we start by cutting the link between x and its parent by adding it to the rootlist. Let x is a node in F. At some time x was a root. Then x was linked to another node. Suppose at some time two children of x were removed by cuts. As soon as second child has been lost we cut x from its parent and make it a new root. But we are not done yet. Since x might be the second child cut from its parent. So we have to check for its parent. Therefore, we recursively run CASCADING-CUT on its parent till we reach the root or cut the first child from a node.

Notice at each run of CASCADING-CUT the *lost* bit of a node is getting reset. Therefore, the total time taken by FIB-DECREASE-KEY is $O(1 + \#\text{lost bits reset})$.

1.5.5 Bounding the Maximum Degree

To prove that the amortized time of FIB-EXTRACT-MIN and FIB-DELETE is $O(\log n)$ we must show that upper bound of the maximum degree of any node after CONSOLIDATE function is $O(\log n)$. In particular, we will show its $\lfloor \log_\phi n \rfloor$ where ϕ is the golden ratio.

Lemma 1.5.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, \ldots, y_k$ denote the children of x in the order in which they were linked to x from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, \ldots, k$.

Proof: Obviously $y_1.degree \geq 0$. The only function that adds a child to a node is the function CONSOLIDATE. Now for $i \geq 2$, y_i was linked to x when all of $y_1, \ldots, y_{i-1}$ were children of x, and therefore we must have had $x.degree \geq i - 1$. Because node y_i is linked to x only if $x degree = y_i.degree$ we must also have $y_i.degree \geq i - 1$. Since then node y_i has lost at most one child, since it would have been cut from x by CASCADING-CUT if it had lost two children. We conclude that $y_i.degree \geq i - 2$. ∎

Lemma 1.5.2

Let x be a node in a Fibonacci heap and let $k = x.degree$. Then

$$size(x) \geq F_{k+2} \geq \phi^k$$

Proof: We will prove this using induction. For $k = 0$, $F_2 = 1$ so this is obviously true. For $k = 1$ there is one child of x. Hence, $size(x) = 2 = F_3$. Suppose this is true for $1, \ldots, k-1$. Let $y_1, \ldots, y_k$ are the children of x in the order in which they were linked to x. By the above lemma we have $y_1.degree \geq 0$ and $y_i.degree \geq i-2$ for all $i = 2, \ldots, k$. Hence, by Induction hypothesis we have $size(y_i) \geq F_{i-2}$ for all $i = 2, \ldots, k$. Therefore,

$$size(x) \geq 1 + \sum_{i=1}^{k} size(y_k) \geq 2 + \sum_{i=2}^{k} F_k = 1 + \sum_{i=0}^{k} F_k = F_{k+2} \geq \phi^k$$

Hence, we have the lemma. ■

Corollary 1.1

The maximum degree of any node in Consolidate, $D(n) = O(\log n)$.

1.5.6 Time Complexity Analysis of Dijkstra

Now we will calculate the amortized time of Dijkstra algorithm. Before that we will calculate the amortized cost of the data structure. Let in an algorithm Fib-Extract-Min was called t times. Therefore, total cost of all t many Fib-Extract-Min calls is $O(t \log n +$ total #heaps created). Now heaps are created because of Fib-Extract-Min functions and Fib-Decrease-Key function. We know Fib-Extract-Min were called t times and each time it created $O(\log n)$ heaps. Hence, in total Fib-Extract-Min created $O(t \log n)$ heaps. Therefore, time taken by the t many Fib-Extract-Min calls is $O(t \log n +$ #Fib-Decrease-Key calls).

Now suppose in an algorithm k times the function Fib-Decrease-Key function were called. Hence, this takes $O(k +$ #total number of lost bits reset$) = O(k +$ #total number of lost bits rset) time. Now the *lost* bits are set only by the Fib-Decrease–Key. Therefore, #total number of lost bits rset = #Fib-Decrese-Key was called. Therefore, the total time taken by all the Fib-Decrese-Key calls is $O(k)$.

Hence, in an algorithm if t times Fib-Extract-Min was called and k times Fib-Decrese-Key was called then total time taken by Fib-Extract-Min is $O(t \log n + k)$ and total time taken by Fib-Decrese-Key is $O(k)$. Therefore, amortized time taken by Fib-Extract-Min is $O(\frac{t}{k} \log n)$ and by Fib-Decrese-Key is $O(1)$.

Now in the Dijkstra algorithm Fib-Extract-Min is called n times and Fib-Decrese-Key is called $O(m)$ times where n is the number of vertices in the graph and m is the number of edges in the graph. Hence the amortized cost of Fib-Extract-Min is $O(\log n)$ and Fib-Decrease-Key is $O(1)$.