Lecture 12

Input: a directed graph $G = (V, E)$ with non-negative edge weights and a special vertex s.

Date

Output: Two arrays: the distance array $d[1..n]$ and the parent array $\pi[1..n]$

Dijkstra's algorithm for single source shortest paths

1. Initialization: $d[s] = 0$ and $d[u] = \infty \ \forall u \in V - \{s\}$
   $\pi[u] = nil \ \forall u \in V$
2. $S = \phi$ and $Q = V$.
3. while $Q \neq \phi$ do
   {
   - extract the vertex u with min d-value from Q.
   - $S = S \cup \{u\}$.
   - relax all edges leaving u.

   }

We will now prove the correctness of Dijkstra's algo.
Let $\delta(s, u)$ denote the distance from s to u in G.

Claim. When Dijkstra's algo. terminates, we have
$d[u] = \delta(s, u) \ \forall u \in V$.

Proof. We will show that we have $d[u] = \delta(s, u)$ at the time when u is added to the set S. Hence this equality holds at all times thereafter.

For the purpose of contradiction, let u be the first vertex for which $d[u] \neq \delta(s, u)$ when u is added to S. We must have $u \neq s$ because when s is added to S, $d[s] = 0$ and $\delta(s, s) = 0$.

There is a shortest path from s to u in G. Let this be $s - x_1 - x_2 - \cdots - x_k - u$ where we will set $x_0 = s$ and $x_{k+1} = u$.

Just before adding u to S, we have $s \in S$ and $u \in V - S$.

So there must exist some consecutive pair x_i, x_{i+1} such that $x_i \in S$ and $x_{i+1} \in V-S$.

- Observe that $d[x_i] = \delta(s, x_i)$ since u is the first vertex for which at the time of adding to S, we had $d[u] \neq \delta(s, u)$.
- When we added x_i to S, we relaxed the edge (x_i, x_{i+1}). So $d[x_{i+1}] = d[x_i] + w(x_i, x_{i+1}) = \delta(s, x_{i+1})$.

So $d[x_{i+1}] = \delta(s, x_{i+1})$
$\underbrace{\leq \delta(s, u) < d[u]}$

By assumption, $d[u] \neq \delta(s, u)$. This means $d[u] > \delta(s, u)$ since $d[u]$ is the length of some path from s to u in G. (why?)

The inequality $d[x_{i+1}] < d[u]$ contradicts the fact that right now u is the vertex in $V-S$ with min d-value. Recall that $x_{i+1} \in V-S$. □

Note that $\delta(s, x_{i+1}) \leq \delta(s, u)$ crucially used the fact that all edge weights are non-negative.

Running time of Dijkstra's algorithm.
The while loop runs for n iterations. In each iteration we perform 1 extract-min operation and $\leq$ out-degree(u) many relax operations where u is the vertex extracted from Q in this iteration.

- In total we perform n extract-min operations and $\leq m$ decrease-key operations.

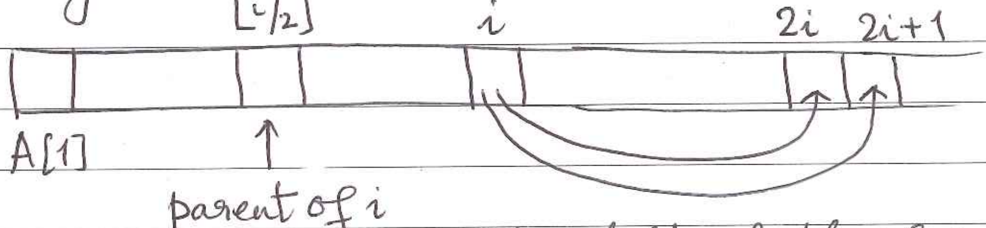
How do we maintain the set Q so that we can perform these operations efficiently?

Suppose we maintain Q as an array $A[1..n]$.
The vertices are numbered 1 to n.
So $d[v]$ is stored in $A[v]$.
- Then Decrease-Key takes $O(1)$ time. To decrease $d[v]$ from α to β, we simply assign $A[v] = \beta$.

However Extract-Min takes $\Theta(n)$ time since we need to search the entire array A to find the vertex u with min d-value.

So the total time taken by Dijkstra's algorithm with the above implementation is $O(m + n^2)$.

Another option: suppose we maintain Q as a min-heap. The heap data structure is an array object that can be viewed as a nearly complete binary tree.

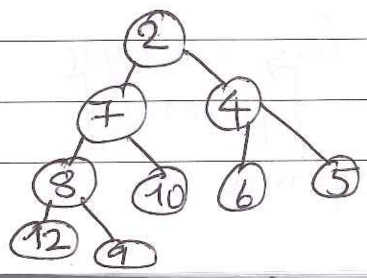


The left child of i is stored in $A[2i]$ and the right child of i in $A[2i+1]$.

The min-heap property is that $A[parent[i]] \leq A[i]$.
- in our problem, building the heap is easy. Recall that at the beginning, $d[s] = 0$ and $d[u] = \infty \ \forall u \in V - \{s\}$.

So s will be the root of the heap.
Other vertices are put in arbitrarily.

Extract-Min (S): min = $A[1]$
Put $A[\text{heap-size}]$ at the root and let the value float down.
Takes $O(\log n)$ time
height of the root.



So a min-heap implements Extract-Min much more efficiently than an array.
- But what about Decrease-Key operation?

Decrease-Key (v, k) decreases v's d-value to k.
- first update v's d-value to k.
  (however this may violate the min-heap property)
- if v's d-value is less than its parent's (parent in the heap) d-value then
  - find a path in the heap from v's location to the root to find a proper place for this newly decreased d-value
  - this takes $O(\log n)$ time
- Note that we assume v is accessed by the location i in A where it currently sits

So using a min-heap, we get a running time of $O((m+n)\log n)$ for Dijkstra's algorithm. We can assume $m \geq n-1$, So this is an $O(m \log n)$ algorithm.

- Let us compare both the options.

	Extract-Min	Decrease-Key
Array:	$\Theta(n)$	$O(1)$
Min-heap:	$O(\log n)$	$O(\log n)$
Can we have the best of both worlds?	$O(\log n)$	$O(1)$

We want a data structure for maintaining the set S of vertices, each with an associated d-value so that we can implement m Decrease-Key opns. and n Extract-Min opns. in $O(m + n\log n)$ time.

We will implement each Decrease-Key operation in $O(1)$ amortized time and each Extract-Min operation in $O(\log n)$ amortized time.

What is amortization?

- Let us see an example. Let A be a string of n bits all set to 0.

$A[1..n]$: array of size n.

Treat this array as a binary number and add 1 to this number m times. In fact, let us make this problem even harder: each addition operation starts at some specified $A[j]$ and scans through the higher order bits until the carry-over process stops.

Worst case time per addition is $\Theta(n)$.

What is the amortized time per addition?

$$= \frac{\text{total time taken}}{\text{number of additions}}$$

Suppose whenever a "0" turns into a "1", we charge this operation 2 units of cost: 1 unit to pay for this operation and 1 unit credit which this "1" keeps with itself to pay for turning itself into "0" during a future addition.

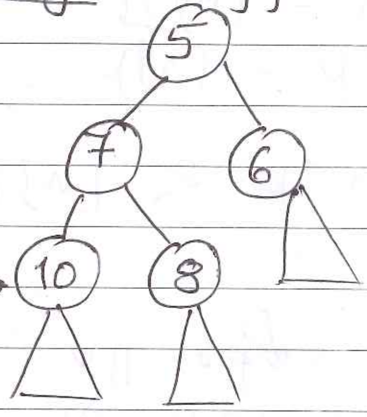
Any addition operation turns a single "0" into a "1". We charge this operation 2 units of cost. Thus the total work done by all additions ≤ 2(number of additions)

$\therefore$ Amortized cost per addition ≤ 2.

Our goal now is to perform m Decrease-Key opns. and n Extract-Min opns. in $O(m + n\log n)$ time.

We will now give an outline of how we will perform m Decrease-Key operations and n Extract-Min operations in $O(m + n\log n)$ time.

<u>Decrease-Key</u>: Suppose our min-heap is as follows.



Let us say we now have to decrease this value to 4.

<u>Idea</u>: Instead of traversing the path from this node to the root, why not just cut-off this subtree and start a new tree?

So heaps are no longer balanced binary trees. We have a collection of min-heap ordered trees now.

- In order to find the node with min d-value, we have to check the root of every tree. So as to perform Extract-Min operation in $O(\log n)$ time, we need to ensure that there are $O(\log n)$ number of trees.

- Each Decrease-Key operation creates a new tree. So we need to clean-up our data structure to maintain an upper bound of $O(\log n)$ on the number of trees.

- This clean-up subroutine will be called whenever we perform Extract-Min.

Hence it won't be the case that every Extract-Min takes $O(\log n)$ time. However similar to the example of binary addition, m Decrease-Key opns. + n Extract-Min opns. will take $O(m + n\log n)$ time.