Lecture 11 Dijkstra's Algorithm, Amortized Analysis, Bellman-Ford Algorithm
Scribes: Himanshu Bhandoh (2015), Virginia Williams, and Anthony Kim (2016) **Date:** November 2, 2016

1 Implementation of Dijkstra's Algorithm

In the last lecture, we studied Dijkstra's algorithm that solves the single source shortest paths problem in directed graphs with nonnegative edge weights and proved its correctness. In this lecture, we study different implementations of Dijkstra's algorithm and analyze corresponding runtimes.

Let $G = (V, E)$ be a directed graph with n nodes and m edges with nonnegative weights. An edge $(x, y) \in E$ has weight $c(x, y)$. The key idea behind Dijkstra's algorithm is to maintain an estimate of the current shortest path ending at v for $v \in V$ over the iterations. More specificially, the algorithm computes an estimate of the distance of v from the source s, $d[v]$, such that -

1. At any point in time, $d[v] \geq d(s, v)$, and
2. When v is finished, $d[v] = d(s, v)$.

Algorithm 1: Dijkstra$(G = (V, E), s)$

```
d[v] ← ∞, ∀v ∈ V   // set initial distance estimates
// to maintain paths: set π(v) ←nil for all v, π(v) represents the predecessor of v
d[s] ← 0
F ← {v | ∀v ∈ V}   // F is the set of nodes that are yet to achieve final distance estimates
D ← ∅   // D is the set of nodes that have achieved final distance estimates
while F ≠ ∅ do
    x ← element in F with minimum distance estimate
    for (x, y) ∈ E do
        d[y] ← min{d[y], d[x] + c(x, y)} // "relax" the estimate of y
        // to maintain paths: if d[y] changes, then π(y) ← x
    F ← F \ {x}
    D ← D ∪ {x}
```

Note that the for loop is over all edges incident on node x. $\pi(\cdot)$ is used to store the shortest paths found and $\pi(v)$ represents the predecessor of v on the shortest path from s to v.

For an example run of Dijkstra's algorithm, please refer to the lecture slides or CLRS.

We implement Dijkstra's algorithm with a priority queue to store the set F, where the distance estimates are the keys. The initialization step takes $O(n)$ operations to set n distance estimate values to infinity and 0. In each iteration of the while loop, we make a call to find the node x in F with the minimum distance estimate (via, say, `FindMin` operation). Then, we relax each edge leaving x (via `DecreaseKey`). We remove node x (via `DeleteMin`) and add it to D. In total, there are n calls to `FindMin` and n calls to `DeleteMin` since nodes are never re-inserted into F. Similarly, there will be m calls to `DecreaseKey` to relax the edges since each edge will be relaxed at most once.

Depending on how quickly our priority queue can support `FindMin`, `DeleteMin`, and `DecreaseKey` operations, the total runtime of Dijkstra's algorithm is on the order of

$$n \cdot (T_{\texttt{FindMin}}(n) + T_{\texttt{DeleteMin}}(n)) + m \cdot T_{\texttt{DecreaseKey}}(n).$$

We consider the following implementations of the priority queue for storing F:

- Store F as an array:
  Each slot corresponds to a node and stores the distance $d[j]$ if $j \in F$, or NIL otherwise. `DecreaseKey` runs in $O(1)$ as nodes are indexed. `FindMin` and `DeleteMin` run in $O(n)$ as the array is not sorted and we have to go through the whole array. The total runtime is $O(m + n^2) = O(n^2)$.

- Store F as a red-black tree:
  All operations run in $O(\log n)$ time. We implement `DecreaseKey` by deleting and re-inserting with the new key. The total runtime is $O((m+n)\log n)$. If graph G is sparse with few edges, then the red-black tree implementation is faster than the array implementation. However, it can be slower when G is dense with $m = \Theta(n^2)$.

- Store F as a Fibonacci heap:
  Fibonacci heaps are a complex data structure which is able to support the operations `Insert` in $O(1)$, `FindMin` in $O(1)$, `DecreaseKey` in $O(1)$ and `DeleteMin` in $O(\log n)$ "amortized" time, over a sequence of calls to these operations. The meaning of amortized time in this case is as follows: starting from an empty Fibonacci heap, any sequence of operations that includes a `Insert`'s, b `FindMin`'s, c `DecreaseKey`'s and d `DeleteMin`'s take $O(a+b+c+d\log n)$ time. The total runtime is $O(m+n\log n)$.

2 Amortized Time

Note the runtimes listed for the operations of Fibonacci heaps are not worst-case runtimes. Instead, they are, what we call *amortized* runtimes. We say an operation on a data structure takes amortized $t(n)$ time if starting from an empty data structure, performing the operation L times takes $O(L \times t(n))$ time in total. This means the runtime of the operation is $O(t(n))$ when averaged over the sequence of L instances of the operation. Each individual operation call may take much more than $t(n)$ time, but this is compensated by many cheap operation calls (that take much less than $t(n)$ time).

We analyze the amortized cost of incrementing a *binary counter* by one when the count is represented in binary. Consider a b-bit counter which starts at 0 (i.e. b 0's). In each increment operation, we update the counter's bits correspondingly by flipping some bits from 0 to 1, or vice versa.

Some of the increment operations may take $\Omega(b)$ time. For example, an increment operation can require carrying b bits:

$$\begin{aligned} &11111111 \\ &\quad +1 \\ &= 100000000 \end{aligned}$$

Other increment operations can take $O(1)$ time:

$$\begin{aligned} &10000000 \\ &\quad +1 \\ &= 10000001 \end{aligned}$$

All this said, we can show the amortized cost of the increment operation on a binary counter is $O(1)$. Even though some increments take time linear in the number of bits, if we do n increment operations to the counter starting from the all 0s, each operation takes $O(1)$ time on average.

Claim 1. *The total time to increment a binary counter n times is $O(n)$.*

We use what is known as the *accounting* method to prove this claim. Each nonzero bit in the binary counter will get a "credit" obtained from earlier increment operations that will then be used to pay for later expensive operations. More specifically, we will maintain the *invariant* that every 1 in the binary representation has a "credit", which we represent as $\oplus$, associated with it.

Let x be the binary counter. If we start with an "empty" integer – that is 0 – then clearly all 1's have a credit as there are no 1's. Assume that all the 1's of x have a credit at the start of an increment operation. In each increment operation, we know the first addition will require constant work for which the addition operation will be charged with. We actually "charge" the addition operation *two* credits, represented as $\oplus\oplus$, to the new 1 to be added:

$$\begin{aligned} x = 1^{\oplus}1^{\oplus}0 \\ + \\ 1^{\oplus\oplus} \end{aligned}$$

Now we start adding. We will maintain the invariant that any "carry" bit will have two $\oplus$ credits. For completeness, we'll call the original 1 to be added to x a "carry" as well.

Now, at each point we are adding a carry bit to a bit in x. If the carry bit is 0, we do nothing and stop. If the carry bit to be added to the i-th bit is 1 and the i-th bit of x is 0 (Note i starts at 0), then one of the $\oplus$ credits of the carry bit is used to store 1 in $x[i]$ and the other remains on this new 1 as $\oplus$:

$$\begin{aligned} & 1^{\oplus}1^{\oplus}0 \\ & \quad +1^{\oplus\oplus} \\ = 1^{\oplus}1^{\oplus}1^{\oplus} \end{aligned}$$

At this point, the carry for the $i+1$-st slot is 0 and we can stop the addition.

When the carry bit to be added to the i-th bit is 1 and $x[i]$ is 1, however, we will get a non-zero carry bit for the $i+1$-st position. In this case, we will use one $\oplus$ from the 1 stored in $x[i]$ to pay for storing a 0 in $x[i]$ (doing the carry addition), and we'll move the two $\oplus$s of the carry bit to the new carry bit for the $i+1$-st position. This maintains the invariant that all 1s in x have a credit and all carries have two credits.

For example, consider an increment operation on the binary counter $x = 0111$:

$$\begin{aligned} & 01^{\oplus}1^{\oplus}1^{\oplus} \\ & \quad +1^{\oplus\oplus} \end{aligned}$$

A new carry bit is formed:

$$\begin{aligned} & 1^{\oplus\oplus} \\ = 01^{\oplus}1^{\oplus}0 \end{aligned}$$

A new carry bit is formed:

$$\begin{aligned} & 1^{\oplus\oplus} \\ = 01^{\oplus}00 \end{aligned}$$

A new carry bit is formed:

$$\begin{aligned} & 1^{\oplus\oplus} \\ =& 0000 \\ = 1^{\oplus}000 \ , \end{aligned}$$

arriving at $x = 1000$.

All carry propagations of additions are for free because they are paid for by the credits accumulated in previous additions of 0's and 1's. There are $O(n)$ credits overall, two for each increment operation. Thus, the total runtime is $O(n)$. The credit system allows you to pay for later long operations by depositing credits from previous short operations. Some operations are long, but over all n increment operations, the total work is $O(n)$. It follows that the increment operation takes amortized $O(1)$ time.

3 Negative Edge Weights

Note that Dijkstra's algorithm solves the single source shortest paths problem when there are no edges with negative weights. While Dijkstra's algorithm may fail on certain graphs with negative edge weights, having a negative cycle (i.e., a cycle in the graph for which the sum of edge weights is negative) is a bigger problem for any shortest path algorithm. When computing a shortest path between two vertices, each additional traversal along the cycle lowers the overall cost incurred and an arbitrarily small distance can be reached after looping around the cycle multiple times. In this case, the shortest path to a node on the cycle is not well defined since it is (negatively) infinite.

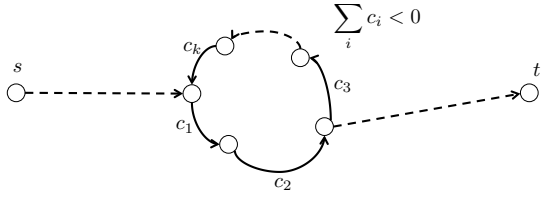

Figure 1: Assume there is a negative cycle along the $s - t$ path. The distance between s and t is not well-defined.

For example, consider the graph in Figure 1. The shortest path from s to t would start from the node s, loop around the negative cycle an infinite number of times and eventually reach destination t. The shortest path would, hence, be of infinite length and is not well-defined.

Besides the negative cycles, there are no problems in computing the shortest paths in a graph with negative edge weights. In fact, there are many applications where allowing negative edge weights is important.

4 Bellman-Ford Algorithm

In this section, we study the Bellman-Ford algorithm that solves the single source shortest paths problem on graphs with edges with potentially negative weights. Given a directed graph $G = (V, E)$ with edge weights given by $c(x, y)$ for $(x, y) \in E$, we want to compute the shortest path distances $d(s, v)$ from source s for all $v \in V$. More specifically, the Bellman-Ford algorithm:

- Detects a negative cycle if it exists and is reachable from s, or
- Computes the shortest path distances $d(s, v)$ for all $v \in V$.

Note $\pi(\cdot)$ is used to store the shortest paths found and $\pi(v)$ represents the predecessor of v on the shortest path from s to v.

For an example run of the Bellman-Ford algorithm, please refer to the lecture slides or CLRS.

The total runtime of the Bellman-Ford algorithm is $O(mn)$. In the first for loop, we repeatedly update the distance estimates $n - 1$ times on all m edges in time $O(mn)$. In the second for loop, we go through all m edges to check for negative cycles in time of $O(m)$.

We prove the correctness of the Bellman-Ford algorithm in two steps:

Claim 2. *If there is a negative cycle reachable from s, then the Bellman-Ford algorithm detects and reports "Negative Cycles".*

Proof. For the sake of contradiction, suppose there exists a negative cycle C reachable from the source s and the Bellman-Ford algorithm does not report "Negative Cycles". Assume C contains nodes $v_1, v_2, \ldots, v_k$ with edges (v_i, v_{i+1}) for $i = 1, \ldots, k$ such that $\sum_{i=1}^{k} c(v_i, v_{i+1}) < 0$, where $v_{k+1} = v_1$. See Figure 2. Let $d[\cdot]$ be the distance estimates determined in the first for loop of the algorithm.

Algorithm 2: Bellman-Ford Algorithm

```
d[v] ← ∞, ∀v ∈ V   // set initial distance estimates
// to maintain paths: set π(v) ←nil for all v, π(v) represents the predecessor of v
d[s] ← 0 // set distance to start node trivially as 0
for i from 1 → n − 1 do
    for (u, v) ∈ E do
        d[v] ← min{d[v], d[u] + w(u, v)}   // update the distance estimate for v
        // to maintain paths - if d[v] changes, then π(v) ← u
// Negative Cycle Step
for (u, v) ∈ E do
    if d[v] > d[u] + w(u, v) then
        return "Negative Cycle"; // negative cycle detected
return d[v] ∀ v ∈ V
```

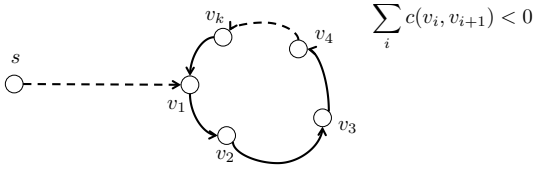

Figure 2: A negative cycle reachable from source s

Since C is reachable from s, there is a path from s to v_1 and to all nodes on C. In particular, there exist simple paths, i.e., paths without cycles, of at most $n-1$ edges to the nodes of C. In the first for loop, the edges on each such simply path get relaxed in order and consequently, $d[v_i]$ will be some finite number less than ∞ for $i = 1, \ldots, k$. Since the Bellman-Ford algorithm does not report "Negative Cycles" in the second for loop, it must be that $d[v_{i+1}] \leq d[v_i] + c(v_i, v_{i+1})$ for $i = 1, \ldots, k$. Adding the inequalities, we obtain

$$\sum_{i=1}^{k} d[v_{i+1}] \leq \sum_{i=1}^{k} d[v_i] + \sum_{i=1}^{k} c(v_i, v_{i+1}) \ .$$

As we are summing over the cycle C, the terms $\sum_{i=1}^{k} d[v_{i+1}]$ and $\sum_{i=1}^{k} d[v_i]$ are equal and can be cancelled. It follows that $0 \leq \sum_{i=1}^{k} c(v_i, v_{i+1})$. This contradicts that C is a negative cycle.

□

In the next claim, we show that if the graph has no negative cycles reachable from the source, then the Bellman-Ford algorithm returns the correct shortest path distances.

Claim 3. *If G has no negative cycles reachable from s, then $d[v] = d(s, v), \forall v \in V$.*

Proof. Let $d_k(v)$ be the value of $d[v]$ after k iterations of the first for loop. We prove by induction the statement that $d_k(v)$ is equal to the minimum distance of a path from s to v with at most k edges. Then, we will have $d_{n-1}(v) = d[v]$ for all node v at termination. Since we can assume that shortest paths have at most $n-1$ edges without loss of generality, the claim follows.

We argue that if there is a path from s to v, then there exists a shortest path from s to v has at most $n-1$ edges. If a shortest path has a cycle, the cycle cannot be negative and we can remove it and improve its total distance. If the cycle has a positive weight, removing the cycle will strictly improve the shortest

path's distance. If the cycle has zero weight, we can ignore the cycle. Hence, we can assume that shortest paths are simple, that is, do not have cycles.

Base Case: When $k = 0$, the distance estimates have been just initialized. So, $d_0(v) = \infty$ if $v \neq s$. Furthermore, $d_0(s) = 0 = d(s, s)$, which is the minimum distance of length-0 paths from s to s. The statement is satisfied for the base case.

Inductive Step: Assume that $d_{k-1}(v)$ is equal to the minimum distance of a $s \to v$ path on at most $k-1$ edges for all v.

Consider $v \neq s$. Let P be a shortest simple $s \to v$ path on at most k edges. Let u be the node just before v on P, and let Q be the sub-path of P from s to u. The path Q would have at most $k-1$ edges and is a shortest path from s to u with at most $k-1$ edges, since sub-paths of shortest paths are also shortest paths. By the inductive hypothesis, Q has distance $d_{k-1}(u)$.

In the k-th iteration, we update $d_k(v)$ such that $d_k(v) \leq d_{k-1}(u) + w(u, v) = w(Q) + w(u, v) = w(P)$. Since whenever $d_k(v)$ is finite, it actually corresponds to the distance of some path from s to v on at most k edges, in particular, it has to be at least as large as the distance of the shortest path from s to v on at most k edges. Thus, $d_k(v) \geq w(P)$. After the k-th iteration, $d_k(v) = w(P)$, and the inductive step follows.

The induction is complete, and the claim is proved. □