1 Dijkstra's runtime

Last time we introduced Dijkstra's algorithm and proved its the correctness. Today we'll look at how long it will take to run. We'll see that the runtime is highly-dependent on the data structure used to store F.

The initialization step takes $O(n)$ operations to set n distance estimate values to infinity and one to 0. Then, within the while loop, we make $O(n)$ calls to finding node with $\min(d[x])$ from F (this is called `FindMin`) as nodes are never re-inserted into F. Additionally, there will be $O(n)$ calls to deleting $min(d[x])$ node from F (`DeleteMin`). Finally, each edge will be relaxed at most once, each in constant time, for a total of $O(m)$ work. During these relaxations, there will be $O(m)$ potential decreases of distance estimates $d[u]$ (`DecreaseKey`).

Thus, the total runtime of Dijkstra's algorithm depends on how quickly our data structure can support each call to `FindMin`, `DeleteMin`, and `DecreaseKey`. That is, the runtime is on the order of

$$n \cdot (FindMin(n) + DeleteMin(n)) + m \cdot DecreaseKey(n).$$

Consider implementing Dijkstra using the following data structures.

- Store F as an array where each slot corresponds to a node and where $F(j) = d[j]$ if $j \in F$, else NIL.
  DecreaseKey runs in $O(1)$ as key is indexed.
  FindMin and DeleteMin run in $O(n)$ as the array is not sorted.
  The total runtime is $O(m + n^2) = O(n^2)$

- Store F as a Red-Black tree.
  DecreaseKey and FindMin run in $O(\log n)$ time. You implement DecreaseKey by deleting and reinserting with the new key.
  Total runtime is $O((m+n)\log n)$, which improves over the array implementation, except in the special case when $m = \Theta(n^2)$.

- 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 each operation. The meaning of amortized time in this case is as follows: starting from an empty Fibonacci Heaps data structure, any sequence of operations that includes a Inserts, b FindMins, c DecreaseKeys and d DeleteMins take $O(a+b+c+d\log n)$ time. Thus, the outcome is that Dijkstra's algorithm can be implemented to run in $O(m + n\log n)$ time.

2 Amortized Time

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 o on a data structure takes amortized $t(n)$ time if starting from an empty data structure, performing L operations takes $O(L \times t(n))$ time. This means, averaged over the L operations, the runtime of o is $O(t(n))$.

Each individual operation might take much more than $t(n)$ time, but this is compensated by many cheap operations (that take much less than $t(n)$ time).

Here, we'll analyze the amortized cost of adding 1 to an integer represented in binary (i.e. maintaining a *binary counter*).

Consider an b bit integer which starts at 0 (i.e. b 0s) and in each operation we add a single 1 to it. Some of these additions may take $\Omega(b)$ time. For example, to add a single 1, we may end up carrying b bits.

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

Other additions will clearly take $O(1)$ time.

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

All this said, we can show is that the amortized cost of adding 1 to an integer is $O(1)$, that is, even though some additions take linear time, on average, if we do n additions of 1 to the counter starting from the all 0s, each addition only costs $O(1)$ operations on average.

Claim 1. *The total time to add n 1s to a binary number is $O(n)$.*

We will use something often called the *piggybank* or *charging* method. Each nonzero bit will get a "credit" obtained from earlier addition operations that will then be used to pay for later carries.

That is, we will maintain the *invariant* that every 1 in the binary representation has a "credit", which we represent as $\oplus$, associated with it. If we start with an "empty" integer – that is 0 – then clearly all 1s have a credit as there are no 1s. Let x be the binary counter. We'll assume (inductively) that initially all the 1s of x have a credit. Every time 1 is added to x, we know the first addition will require constant work which the addition operation will be charged with. We actually "charge" the addition operation *two* credits added 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 with a bit in x. If the carry is 0, we do nothing and stop. If the carry to be added to the ith bit is 1 and the ith bit of x is 0 (i starts at 0), then one of the $\oplus$ credits of the carry 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 so we can stop the addition.

When the carry to be added to the ith 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 to the new carry for the $i+1$st position. This maintains the invariant that all 1s in x have a credit and all carries have two credits.

$$\begin{aligned} &001^{\oplus}1^{\oplus} \\ &\quad +1^{\oplus\oplus} \end{aligned}$$

A new carry is formed:

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

A new carry is formed:

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

A new carry is formed:

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

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

3 More Shortest Path Algorithms

Taking a step forward from Dijkstra's Algorithm, we will discuss more shortest paths algorithms. Specifically, the two algorithms we wish to cover are:

- **Bellman-Ford Algorithm.** Just like Dijkstra's Algorithm covered last week, the Bellman-Ford Algorithm too is a Single Source Shortest Paths (SSSP) algorithm. In Dijkstra's algorithm, the edge weights were conditioned to be all nonnegative. We are going to remove this condition and allow negative weights as well as non-negative edge weights to solve the problem using the Bellman-Ford Algorithm.
- **Floyd-Warshall Algorithm.** The Floyd-Warshall algorithm solves the All Pairs Shortest Paths (APSP) problem. Given a graph, we can use the algorithm to find the distance between every pair of vertices. This is one of the simplest algorithms to solve for APSP, and the current best algorithms for the problem are only slightly better in their runtime. (Williams' algorithm is currently the best and solves APSP on n nodes in $n^3/2^{\Theta(\sqrt{\log n})}$ time.

We will cover the first algorithm in this lecture and the second in the next lecture.

3.1 Negative Edge Weights

The problem with negative edge weights is that we may have a negative cycle (i.e. a cycle in the graph for which the sum of edge weights is negative). What really is the problem with negative cycles? Simply put, each additional loop traversal on the cycle lowers the overall cost incurred. Thus, one can keep looping in the cycle to ultimately get a distance of -∞. Hence, for such a case, we say that shortest path is not well defined since it is (negatively) infinite.

The shortest path above would start from the node s, loop around in 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 dealing with negative cycles, there is no problem with asking for the shortest paths in a graph with negative edge weights. There are many applications where allowing negative edge weights is important. Hence, the behavior of our desired algorithm should be as follows:

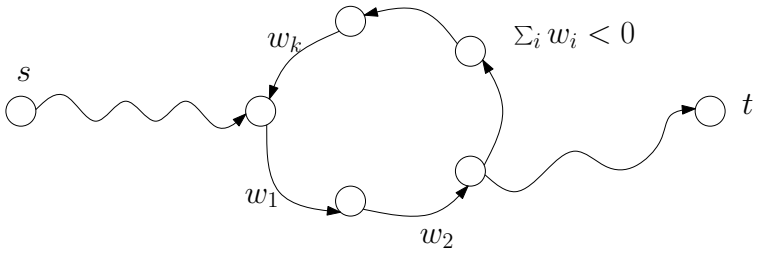


Figure 1: In this case, because there is a negative cycle along the $s - t$ path, the distance between s and t is not well-defined.

- The shortest paths algorithm tries to detect negative cycles and aborts if they exist.
- Otherwise, shortest paths can be computed correctly when there are no negative cycles.

4 Bellman-Ford Algorithm

As discussed earlier, the Bellman-Ford Algorithm solves the Single Source Shortest Paths (SSSP) problem i.e. given G and $s \in V$, compute $d(s,t), \forall\ t \in V$. Also, the algorithm detects negative cycles if they exist.

Algorithm 1: Bellman Ford Algorithm

```
∀v ∈ V, d[v] ← ∞ // set initial distance estimates
//optional: 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 estimate of v
        // optional - 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
```

To see the running time of the proposed algorithm, note that we repeatedly update our distance estimates $n-1$ times on all m edges in time $O(mn)$, and then run through all m edges to check for negative cycles in time of $O(m)$. Thus, the total runtime for the Bellman Ford algorithm is $O(mn)$.

A priori, it probably isn't obvious why this algorithm is correct or why it detects negative cycles. In order to perform the correctness analysis, we look at an equivalent formulation of the algorithm:

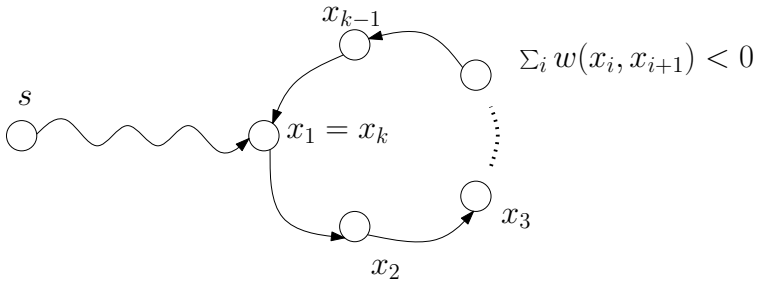
Note that the only difference between the two algorithms is that in the second version, we maintain the distance estimates at each iteration. In practice, we would never maintain such a table (because it makes our space requirement go from $O(n)$ to $O(n^2)$), but keeping track of these intermediate estimates makes the analysis easier. We will prove for correctness of this algorithm.

Claim 2. *If there is a negative cycle reachable from s, then for some $(u,v) \in E$, $d_{n-1}(v) > d_{n-1}(u) + w(u,v)$, i.e. Bellman-Ford detects negative cycles.*

Proof. Let C be a negative cycle reachable from s, where C contains the nodes $x_k = x_1, x_2, \ldots, x_{k-1}$ with edges (x_i, x_{i+1}) for $i \in \{1, \ldots, k-1\}$ (see Figure 4).

Algorithm 2: Equivalent formulation of the Bellman-Ford Algorithm

$\forall v \in V$ *and* $\forall k, d_k[v] \leftarrow \infty$ // set initial distance estimates
$d_k[s] \leftarrow 0$ // set distance to start node trivially as 0
for k *from* $1 \rightarrow n-1$ **do**
  for $(u, v) \in E$ **do**
    $d_k[v] \leftarrow \min\{d_{k-1}[v], d_{k-1}[u] + w(u, v)\}$ // update estimate of v
// Negative Cycle Step
for $(u, v) \in E$ **do**
  if $d_{n-1}[v] > d_{n-1}[u] + w(u, v)$ **then**
    return "Negative Cycle"; // negative cycle detected
return $d_{n-1}[t] \; \forall \; t \in V$



The sum of the weights of edges in the cycle is negative, i.e. $\sum_{i=1}^{k-1} w(x_i, x_{i+1}) < 0$. Now, consider $\sum_{i=1}^{k-1} d[x_i]$. As x_1 is the same as x_k, we have that $\sum_{i=1}^{k-1} d[x_i] = \sum_{i=1}^{k-1} d[x_{i+1}]$.

We will now attempt to prove Claim 2 by contradiction. Suppose that $\forall i \in \{1, \ldots, k-1\}$, $d(x_{i+1}) \leq d(x_i) + w(x_i, x_{i+1})$. Adding over inequalities formed over all i values, $\sum_{i=1}^{k-1} d[x_{i+1}] \leq \sum_{i=1}^{k-1} d[x_i] + \sum_{i=1}^{k-1} w(x_i, x_{i+1})$. As shown above, we can replace $\sum_{i=1}^{k-1} d[x_{i+1}]$ by $\sum_{i=1}^{k-1} d[x_i]$. The inequality thus becomes $\sum_{i=1}^{k-1} d[x_i] \leq \sum_{i=1}^{k-1} d[x_i] + \sum_{i=1}^{k-1} w(x_i, x_{i+1})$. Reducing the inequality, $0 \leq \sum_{i=1}^{k-1} w(x_i, x_{i+1})$. However, the weight of negative cycle $\sum_{i=1}^{k-1} w(x_i, x_{i+1})$ should be less than 0, which is a contradiction. Hence, the algorithm will detect the negative cycle and Claim 2 is proven. □

With the next Claim we show that if the graph has no negative cycles, then Bellman-Ford returns the correct distances.

Claim 3. *If G has no negative cycles then $d_{n-1}(v) = d(s, v)$ $\forall v \in V$*

Proof. By induction on k, we will prove that $d_k(v)$ = minimum weight of a path from s to v on $\leq k$ edges.

If a shortest path has a cycle, we can remove it and improve its weight. The reason for this is simple. Since we are considering the case where there are no negative weights, we can argue that by traversing on a cycle we can never reduce the weight of the path from node s to node v. If the sum of weights of edges on the cycle is positive, we would be able to reduce the weight of the path by removing the cycle from our path. If this sum is zero, the cycle can be ignored. Hence, we can claim that the shortest path, without loss of generality, is simple.

Base Case: $k = 0$

Here, $d_0(v) = \infty$ if $v \neq s$

$d_0(s) = 0 = d(s, s)$ = minimum weight of length 0 path from s to s. The claim is trivially satisfied in the base case.

Inductive Step: Assume that $d_{k-1}(v)$ = minimum weight of $s \rightarrow v$ path on $\leq k-1$ edges $\forall v$.

Consider $v \neq s$. Let P be a shortest simple $s \rightarrow v$ path on $\leq k$ edges. Let u be node just before v on P, and let Q be the path from s to u . The path Q would have $\leq k-1$ nodes and is a shortest path on $\leq k-1$

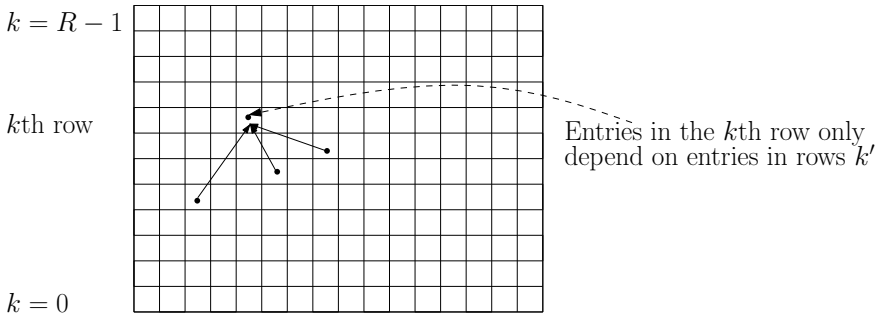


Figure 2: A general Dynamic Programming Table. Our algorithms work by filling up the table from bottom to top, where a given level only depends on levels below it.

edges from s to u as subpaths of shortest paths are also shortest. By the inductive hypothesis, Q has weight $d_{k-1}(u)$.

Now, in iteration k, we update $d_k(v) \leq d_{k-1}(u)+w(u,v) = w(Q)+w(u,v) = w(P)$. Since whenever $d_k(v)$ is finite, it actually corresponds to the weight of some path from s to v on at most k edges, in particular, it has to be at least as big as the weight of the *shortest* path from s to v on at most k edges. Thus $d_k(v) \geq w(P)$, and thus after the kth iteration, $d_k(v) = w(P)$. This proves the induction step for iteration k.

Hence, $d_{n-1}(v) = d(s,v)$, and we have proved the correctness of the Bellman-Ford Algorithm.. □

5 Dynamic Programming.

The Bellman-Ford algorithm is a dynamic programming algorithm. Dynamic programming is a basic paradigm in algorithm design. It is a method of solving problems that proceeds in a bottom up manner to build a table that provides us with the desired solution.

The first step for solving a Dynamic Programming problem is to analyze the problem and find its structure. For instance, in the Bellman-Ford algorithm, we discovered that the paths of length at most k can be evaluated by leveraging the paths of length $k-1$ and the edge weights. Finding the problem structure helps give us the recurrence relation between "intermediate solutions", similarly to divide-and-conquer. Unlike divide-and-conquer, however, the intermediate solutions can be reused many times. Instead of recomputing them, we use the recurrence relation to build a table that stores intermediate solutions, often from the bottom up, obtaining the final result from some of the entries of the table. In the case of the Bellman-Ford Algorithm, we get the final result from the top row.

The row numbered $k = 0$ in the figure represents the initialization of the table for the Bellman-Ford algorithm i.e. $d_0 = \infty \ \forall v \neq s$ and $d_0(s) = 0$. Each column represents a node in the graph, while each row represents the values of $d_k[v]$ for the particular k value. The key observation here is that any slot of table T in row k only depends on slots in rows $< k$. So, we fill up row k after row $k-1$ by computing some function on the slots below. We can thus simply look-up values to evaluate and build our rows in a bottom up manner. For instance, in the case of the Bellman-Ford algorithm, the number of slots we lookup to populate the slot corresponding to $d_k[v]$ is the in-degree of node v.