CIS 575. Introduction to Algorithm Analysis
Material for April 12, 2024

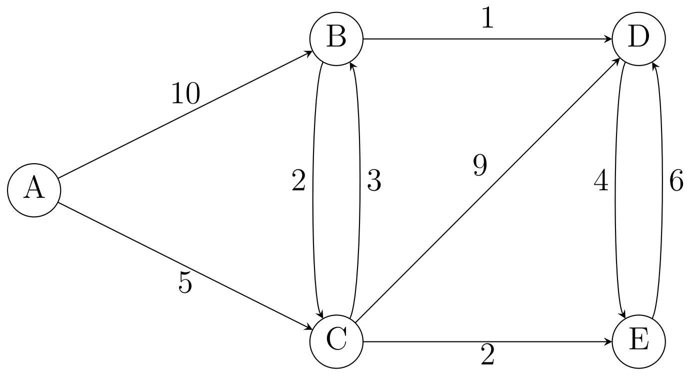
Single-Source Shortest Paths: Dijkstra's Algorithm

The topic of this note is presented in *Cormen*'s Section 22.3.

1 Single Source Shortest Paths

We are given a directed graph, where each edge has a length, and a *specific* source node. Our goal is, for each node u, to find the length of the shortest path from the source to u. (We may assume that all nodes are reachable from the source.)

As our running example, we shall use a slightly modified version of the graph from Figure 22.6 in *Cormen*:



With A the source, it is not hard to see that the shortest path to B has length 8 (through C), the shortest path to C has length 5 (directly), the shortest path to D has length 9 (through C and B), and the shortest path to E has length 7 (through C). We shall now present an algorithm for computing these lengths: **Dijkstra's algorithm**[1].

> As for Floyd's algorithm, we can *not* allow the presence of any *cycle* with negative length, as then no shortest path (involving the nodes in that cycle) would exist. But whereas Floyd's algorithm allows edges with negative length, to apply Dijkstra's algorithm we must require that **all edges have non-negative length**. If we need to compute single source shortest paths for graphs where some edges have negative length, we will have to use other (and in general slower) algorithms, such as the Bellman-Ford algorithm.

[1]We have already seen another algorithm discovered by Edsger Dijkstra: *The Dutch National Flag*.

1.1 Dijkstra's Algorithm

The key idea is to maintain a set S of nodes, and a table d, such that:

1. for a node v with $\mathbf{v} \in \mathbf{S}$ then $\mathbf{d(v)}$ is the **smallest length of a path from the source to** v, and there will be a shortest path where all nodes belong to S

2. for a node v with $\mathbf{v} \notin \mathbf{S}$ then $d(v)$ is the **smallest length of a path from the source to** v **where all nodes except** v **belong to** S (hence $d(v)$ is an upper approximation of the length of the shortest path from the source to v).

Let us illustrate how that works for our example. Initially, we have $S = \{A\}$, and obviously $d(A) = 0$. For any other node v, the only path from A to v where all nodes except v belong to $\{A\}$ consists of an edge from A to v. Hence, d is given by the table

S	$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$
$\{A\}$	0	10	5	∞	∞

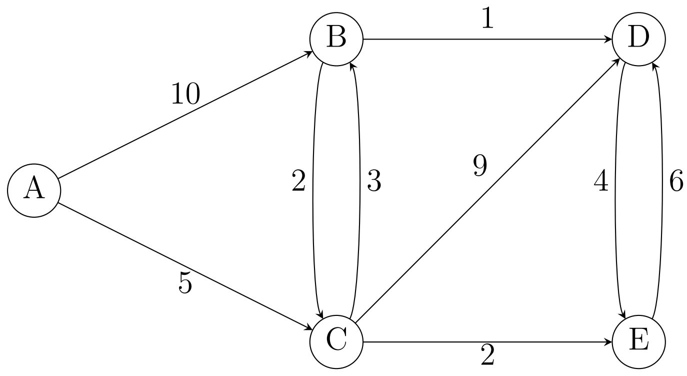
Based on that table, *without looking at the actual graph*, we can make a key inference: no path from A to C can have length less than 5. For assume that π is a path from A to C; there are two cases:

1. All nodes in π belong to S, except for C. But by the specification of d, we know that no such path can be shorter than 5.

2. There exists $v \notin S$ and $v \neq C$ such that we can write $\pi = \pi_1\pi_2$ with π_1 a path from A to v where all nodes except v belong to S, and with π_2 a path from v to C.

   But no matter whether v is B or D or E, we see from the above table that π_1 cannot have length less than 5. And by our requirements that *no lengths are negative*, this shows that π cannot have length less than 5.

We conclude it is safe to add C to S, with $d(C)$ still 5. But now we may need to update other d-entries, so as to take into account paths whose second last node is C. For the reader's convenience, let us again display the example:



We see that in particular, B can now be reached by a path where all nodes except B are in S and which has length 8. Also updating other entries, we arrive at:

S	$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$
$\{A, C\}$	0	8	5	14	7

Among the nodes not in S, we see that E is the one with the smallest d-value. Reasoning as above, we can then infer that no path from A to E can be shorter than 7, and that we can therefore safely add E to S, while updating d (at D only):

S	$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$
$\{A, C, E\}$	0	8	5	13	7

Among the nodes not in S, we see that B is the one with the smallest d-value. We can then again infer that no path from A to B can be shorter than 8, and that we can therefore safely add B to S, while updating d (at D only):

S	$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$
$\{A, B, C, E\}$	0	8	5	9	7

Since D is the only node not yet in S, we can add it, and we are done. We see that the lengths of the shortest paths are as we had expected.

Algorithm We may implement Dijkstra's algorithm as follows:

```
foreach node $X$
    if $X$ is the source
        $d[X] \leftarrow 0$
    else
        $d[X] \leftarrow \infty$
$S \leftarrow \emptyset$
while there is a node not in $S$
    let $Y \notin S$ be a node such that $d[Y] \leq d[Z]$ for all $Z \notin S$
    $S \leftarrow S \cup \{Y\}$
    foreach node $Z \notin S$
        if there exists edge from $Y$ to $Z$ with length $q$
            if $d[Y] + q < d[Z]$
                $d[Z] \leftarrow d[Y] + q$
                record that shortest path to $Z$ goes through $Y$
```

Observe that in the first iteration of the `while` loop, Y will be the source node.

Let us now justify why we can safely add Y to S (generalizing the argument we provided for the specific example). The situation is that

- We can *assume* that $d[Y]$ is the length of the shortest path from the source to Y where all nodes except Y are in S
- We must *prove* that $d[Y]$ is the length of the shortest path from the source to Y, and that there is a shortest path where all nodes are in $S \cup \{Y\}$.

Our non-trivial obligation is to prove that if π is a path from the source to Y then π must have length $\geq d[Y]$. We shall show that by a case analysis:

- For the case when all nodes in π except Y are in S, this follows from our assumption.
- For the case when π contains a node Z with $Z \neq Y$ and $Z \notin S$, let Z be the first such node. We can thus decompose π into a path π_1 from the source to Z where all nodes

except Z belong to S, and a path π_2 from Z to Y. We may now infer the desired inequality through the chain

$$\begin{aligned}
\text{length of } \pi &\geq \text{(as no edges with negative length)} \\
\text{length of } \pi_1 &\geq \text{(due to the property of table } d\text{)} \\
d[Z] &\geq \text{(as } d[Y] \text{ was minimum among nodes not in } S\text{)} \\
d[Y] &
\end{aligned}$$

Running Time The structure of the algorithm is very much like Prim's algorithm[2]. In particular, we need to repeatedly extract the node with the smallest associated value; for that purpose, we may use linear search in the array, or a priority queue which (as we saw earlier in this course) allows us in logarithmic time to *(i)* retrieve a node u with $d[u]$ mimimum and subsequently delete it, and *(ii)* for some w decrease the value of $d[w]$.

For simplicity, let us assume that the graph is implemented by adjacency lists. With n the number of nodes and a the number of edges, we can then analyze the algorithm:

		priority queue	linear search
n times	find & extract minimum	$\Theta(\lg(n))$	$\Theta(n)$
a times	update entries	$\Theta(\lg(n))$	$\Theta(1)$
		$\boldsymbol{\Theta(a \lg(n))}$	$\boldsymbol{\Theta(n^2)}$

1.2 Single/All Shortest Paths

Dijkstra's algorithm computes, for a single source, the shortest paths to all (other) nodes.

1. It is easy to adapt Dijkstra's algorithm to compute, for a single destination, the shortest paths from all (other) nodes: just reverse the direction of the edges.

2. We can use Dijkstra's algorithm to solve the *all pairs shortest path* problem: repeatedly apply it to each node in the graph. This can be implemented to run either in time $\Theta(n^3)$, just as Floyd's algorithm, or (when using priority queues) in time $\Theta(an\lg(n))$ which is faster for sparse graphs.

   It could be an interesting project to implement both Dijkstra's algorithm and Floyd's algorithm, and compare how long each takes to find *all pairs shortest path*.

3. Assume (as is often the case in real life) that we are interested only in the shortest path from a given source to a given destination. One may believe this specific problem could be solved much faster than the general problems listed above, but this turns out *not* to be the case. Intuitively, this make sense: to compute the shortest distance from Manhattan to Dodge City, we'll need to along the way also compute the shortest path from Manhattan to Salina, the shortest path from Manhattan to Wichita (to make sure it wouldn't pay off to go through there), etc.

We may summarize the above considerations:

destination \ source	single	all
single	no faster	Dijkstra (reverse edges)
all	Dijkstra	Floyd, or repeated Dijkstra

[2]Keep in mind, however, that Dijkstra's and Prim's algorithm differ in key aspects: the former is for directed graphs while the latter is for undirected graphs; the former computes shortest paths while the latter finds a minimum spanning tree; the former records the distance from the unique source while the latter records the distance from the current tree.