

15 Shortest Paths: Part 1

In the shortest $s \rightarrow t$ path problem, we are given a directed graph $G = (V, E)$ and two vertices $s, t \in V$. Our goal is to find a minimum length path from s to t . We will focus on the directed graph version of the problem because any algorithm for shortest paths in directed graphs can easily be applied to undirected graphs if we simply replace each undirected edge with two directed edges, one in each direction.

The shortest $s \rightarrow t$ path problem has natural practical applications, for example in a maps app that needs to give the user efficient directions between their location and a desired destination. Interestingly the fastest algorithms for the shortest $s \rightarrow t$ path problem can also solve a more general problem: Single source shortest paths. In the single source shortest path problem, we are given a graph $G = (V, E)$ and a single vertex s . The goal is to find the shortest path from s to every other vertex.

There are many natural variants on the shortest paths problem. The most common one is to allow the input graph to have edge weights. In a weighted graph we define the length of a path as the sum of the weights of the edges in the path, rather than just the number of edges in the path. In this chapter we will see efficient algorithms for single source shortest paths in unweighted graphs as well as graphs with non-negative edge weights. In the next chapter we will look at how to handle negative edge weights and explore an even broader generalization of the shortest $s \rightarrow t$ path problem.

15.1 Shortest Paths in Unweighted Graphs: BFS

Intuitively, breadth first search is simple. First we label s itself with a distance of 0. Then we label all the neighbors of s with a distance of 1. Then we label all of the unlabeled neighbors of those neighbors with a distance of 2. Then we label all the unlabeled neighbors of those vertices with a distance of 3. We continue this process until we've labeled every vertex in the graph with its distance from s .

While this algorithm can give us the distance of each vertex from s , it's not immediately clear how to recover the actual shortest paths from s to each vertex. Returning an array of n paths could require $\Omega(n^2)$ space and thus $\Omega(n^2)$ time (suppose that G is just a long path and s is the first vertex. Then the shortest paths have length $1, \dots, n-1$, so by Theorem 3 their total length is $n(n-1)/2$). There is a more efficient approach, however, which takes advantage of the fact that shortest paths overlap a lot.

Note that shortest paths exhibit optimal substructure: To find the shortest path from s to t , we just need to choose which neighbor u of t comes immediately before t in the path, then find a shortest path from s to u . The shortest path from s to t will then be the shortest $s \rightarrow u$ path plus the $u \rightarrow t$ edge (using anything but the shortest path to get from s to u would just make our $s \rightarrow t$ path longer). Therefore, to keep track of all the shortest paths, we just need to remember the *predecessor* of each vertex: the vertex immediately before it in a shortest path. Storing a predecessor takes only $O(n)$ space. Given the predecessor list, we can read out the shortest path to any given t by following the predecessors until we get to s . If we view the predecessor of each vertex as its parent in a rooted tree, we get a tree rooted at s which we call a **shortest path tree**.

We give the following algorithm based on these ideas. `BREADTHFIRSTSEARCH` uses an array D to store the distance of each vertex from s and an array $pred$ to store the predecessor of each vertex.

```

BREADTHFIRSTSEARCH( $G = (V, E), s$ )
   $D, pred \leftarrow$  length  $n$  arrays
  For each  $v \in V$ :
     $D[v] \leftarrow \infty$ 
     $pred[v] \leftarrow \text{Null}$ 
   $D[s] \leftarrow 0$ 
  Frontier  $\leftarrow \{s\}$ 
  NextFrontier  $\leftarrow \emptyset$ 
  While Frontier  $\neq \emptyset$ :
    For each vertex  $u \in$  Frontier:
      For each vertex  $v \in N_{out}(u)$ :
        If  $D[v] = \infty$ :
           $D[v] \leftarrow D[u] + 1$ 
           $pred[v] \leftarrow u$ 
          Add  $v$  to NextFrontier
      Frontier  $\leftarrow$  NextFrontier
      NextFrontier  $\leftarrow \emptyset$ 
  Return  $D, pred$ 

```

Theorem 71. For every directed graph $G = (V, E)$ and vertex $s \in V$, BREADTHFIRSTSEARCH(G, s) returns arrays D and $pred$ such that for every $v \in V$, $D[v] = \text{dist}(v)$ and (if $\text{dist}(v) \neq \infty$ and $v \neq s$) a shortest $s \rightarrow v$ path ends with the edge $(pred[v], v)$. Furthermore, BREADTHFIRSTSEARCH(G, s) runs in time $O(m + n)$.

Proof. We claim that the while loop in BREADTHFIRSTSEARCH satisfies the following invariant. After i iterations of the loop:

1. $D[v] = \text{dist}(v)$ for every vertex v with $\text{dist}(v) \leq i$.
2. $D[v] = \infty$ for every vertex with $\text{dist}(v) > i$.
3. Frontier contains exactly the vertices v with $\text{dist}(v) = i$.

We prove this by induction on i :

- Base Case: This is true at the start of the first iteration (i.e. after 0 iterations):
 1. We set $D[s]$ to 0 before the loop and s is the only vertex as distance 0.
 2. We set $D[v]$ to ∞ for all other vertices before the loop.
 3. At the start of the first iteration Frontier contains only s .
- Induction Hypothesis: After $i - 1$ iterations of the loop, every vertex v with $\text{dist}(v) \leq i - 1$ has $D[v] = \text{dist}(v)$, every vertex v with $\text{dist}(v) \geq i$ has $D[v] = \infty$, and Frontier contains exactly the vertices with $\text{dist}(v) = i - 1$.
- Induction Step: Let $i > 0$. Let v be a vertex with $\text{dist}(v) = i$. Let u be the vertex preceding v on a shortest $s \rightarrow v$ path. We note that $\text{dist}(u) = \text{dist}(v) - 1 = i - 1$. Thus, by our IH, at the start of iteration i of the loop, $D[u] = i - 1$, $D[v] = \infty$, and u is in Frontier. Also by our IH, we know that every vertex x in Frontier has $D[x] = i - 1$. Thus, if v is assigned a new value in D in this iteration, it will be assigned the correct value of $(i - 1) + 1 = i$. Since u is in the Frontier, we know that v will be assigned a new value in D this iteration, since it will be explored as a neighbor of u if it wasn't already explored as a neighbor of some other vertex in Frontier. Thus, v will be added to NextFrontier, which becomes Frontier at the end of the loop iteration. Therefore, at the end of iteration i , v will be in Frontier and have $D[v] = i$. Since v was an arbitrary vertex at distance i , this implies that at the end of iteration i , every vertex at distance i will have the correct value in D and will be in the Frontier.

It remains to show that no vertex v with $\text{dist}(v) > i$ will have its value in D updated in iteration i or get added to the Frontier. To show this, let v be a vertex assigned a value in D

during iteration i . Since we only assign a new D value to vertices whose current value in D is ∞ , v must not have been assigned a finite value in D previously. By our IH, this implies $\text{dist}(v) \geq i$. Additionally, to be visited in this iteration, v must be a neighbor of a vertex u which was in Frontier at the start of iteration i . By our IH, u must satisfy $\text{dist}(u) = i - 1$. Thus, we can obtain a path of length i to v by following the length $i - 1$ $s \rightarrow u$ path and then the (u, v) edge, so $\text{dist}(v) \leq i$. We conclude that $\text{dist}(v) = i$. Thus, the vertices we assign values in D and add to the frontier this iteration of the loop are exactly the ones at distance i , and all three parts of our loop invariant hold.

This loop invariant immediately implies that when `BREADTHFIRSTSEARCH` returns, $D[v] = \text{dist}(v)$ for every vertex. We note that when the algorithm assigns u to $\text{pred}[v]$, u is a neighbor of v with $D[u]$ (and thus $\text{dist}(u)$, by our invariant) equal to $D[v] - 1 = \text{dist}(v) - 1$. Therefore, there is indeed a shortest path to v which ends by following the $(\text{pred}[v], v)$, and so the pred array defines a valid shortest path tree.

Finally, we analyze the runtime. The work before the while loop takes $O(n)$ time (to initialize v and pred). By our loop invariant, each vertex v gets added to Frontier at most once (in particular, v is added to Frontier in iteration $\text{dist}(v)$ of the loop). When the algorithm examines a vertex v from Frontier it performs $O(1 + \deg_{\text{out}}(v))$ work iterating through the neighbors of v . The extra work of the while loop per iteration is constant, and the while loop runs for at most n iterations (A shortest path can include at most all of the vertices, so no vertex reachable from s can be at distance $\geq n$). Thus the total work of the while loop is $O(n + \sum_{v \in V} (1 + \deg_{\text{out}}(v))) = O(m + n)$, and so the whole algorithm runs in $O(m + n)$ time. $\square$

15.2 Shortest Paths with Non-Negative Edge Weights: Dijkstra's

Now we consider weighted graphs. In particular, graphs with *non-negative* edge weights (we'll see how to handle negative edge weights next chapter). The algorithm we will use for this problem is called Dijkstra's algorithm after Edsger Dijkstra who discovered it in the 1950s (possibly first, although several other researchers independently developed the same algorithm around the same time).

Like BFS, Dijkstra's creates an array D of distance values and an array pred representing the parent relationships in a shortest path tree. Unlike in BFS, we may need to update the entries in D and pred multiple times. Throughout the algorithm we will maintain the invariant that $D[v] \geq \text{dist}(v)$ (i.e. $D[v]$ is an upper bound on the distance to v), and there exists an $s \rightarrow v$ path of length $D[v]$ which ends with the edge $(\text{pred}[v], v)$ (assuming $D[v] < \infty$). Initially these invariants are true because $D[s] = 0$ and $D[v] = \infty$ for every other vertex (infinity is a very safe upper bound on $\text{dist}(v)$).

For a given array of distance upper bounds D , we say that an edge (u, v) is *tense* if $D[v] > D[u] + w(u, v)$. If an edge is tense, we can improve our distance upper bounds by running the following simple procedure, which we refer to as **relaxing** the edge:

$\begin{aligned} &\text{RELAX}(e = (u, v)) \\ &D[v] \leftarrow D[u] + w(u, v) \\ &\text{pred}[v] \leftarrow u \end{aligned}$
--

We note that relaxing an edge preserves our invariants for D and pred : If there is an $s \rightarrow u$ path of length $D[u]$, then we can obtain an $s \rightarrow v$ path of length $D[u] + w(u, v)$ by simply following the $s \rightarrow u$ path then following the edge (u, v) .

Note that we can view the step in `BREADTHFIRSTSEARCH` where we update $D[v]$ and $\text{pred}[v]$ as relaxing the edge (u, v) if we view our unweighted graph as a graph where all edge weights are 1. For arbitrary edge weights, however, we cannot simply use the expanding frontier idea from BFS. We need to be open to revisiting a vertex when we discover a path that uses more edges but has shorter total length.

The core idea of Dijkstra's algorithm is to repeatedly relax all of the out edges of the vertex with minimum $D[v]$ among those whose out edges we haven't relaxed yet. In order to efficiently find the vertex with minimum $D[v]$ we will use a data structure called a priority queue. You should have

seen priority queues before in CSCE 221, but it's okay if you don't remember them. The important thing to know is that a priority queue contains (element, priority) pairs and efficiently supports the operations INSERT for adding an element to the queue, DECREASEKEY for reducing the priority value of an existing element, and EXTRACTMIN, which removes and returns the element with the smallest priority value. We will use $D[v]$ as the priority value for v , so that EXTRACTMIN will get the vertex from our queue with minimum $D[v]$. We give the algorithm below:

```

DIJKSTRA'S( $G = (V, E), s$ )
   $D, pred \leftarrow$  length  $n$  arrays
  For each  $v \in V$ :
     $D[v] \leftarrow \infty$ 
     $pred[v] \leftarrow \text{Null}$ 
   $D[s] \leftarrow 0$ 
   $Q \leftarrow$  a priority queue containing  $(s, 0)$ 
  While  $Q$  is not empty:
     $u \leftarrow Q.\text{EXTRACTMIN}()$ 
    For each vertex  $v \in N_{out}(u)$ :
      If  $(u, v)$  is tense:
        RELAX( $u, v$ )
      If  $v \notin Q$ 
         $Q.\text{INSERT}(v, D[v])$ 
      Else:
         $Q.\text{DECREASEKEY}(v, D[v])$ 
  Return  $D, pred$ 

```

Theorem 72. *For every weighted directed graph $G = (V, E)$ with non-negative edge weights and every vertex $s \in V$, DIJKSTRA'S(G, s) returns arrays D and $pred$ such that for every $v \in V$, $D[v] = \text{dist}(v)$ and (if $\text{dist}(v) \neq \infty$ and $v \neq s$) a shortest $s \rightarrow v$ path ends with the edge $(pred[v], v)$.*

We will sketch the proof of Theorem 72, see CLRS or Erickson for a full proof (you will not be expected to know the details of this proof).

Let v_i be the vertex that DIJKSTRA'S extracts from Q in iteration i of the while loop, and let $V_i := \{v_1, v_2, \dots, v_i\}$. The correctness of Dijkstra's flows from a few core observations:

- Each vertex which is reachable from s eventually gets inserted into the priority queue (This follows by induction on the length of the shortest path to the vertex, using the observation that an edge (u, v) where $D[u]$ is finite and $D[v]$ is ∞ will always be tense).
- After each iteration of the while loop, for each vertex v , $D[v]$ is the length of the shortest path from s to v using only vertices in $V_i \cup \{v\}$. (We can prove this inductively together with the following points)
- When we extract v from Q , $D[v] = \text{dist}(v)$. (Since all edge weights are non-negative, the shortest path to v can only use vertices whose distance from s is $\leq \text{dist}(v)$)
- Once a vertex is extracted from the priority queue it never gets reinserted (This follows from the previous point. If $D[v] = \text{dist}(v)$, then no edge to v can become tense again, because that would imply a shorter path to v)
- $\text{dist}(v_i) \leq \text{dist}(v_j)$ if $i \leq j$ (This follows from the previous two points and the fact that we always remove the vertex with minimum $D[v]$)

Theorem 73. *If G has non-negative edge weights, then Dijkstra's can be implemented to run in time $O(m + n \log n)$.*

Proof. By the observations above, each vertex gets extracted from the priority queue at most once, so DIJKSTRA'S makes $O(n)$ calls to EXTRACTMIN and $O(n)$ calls to INSERT. Since an edge can only be relaxed when we remove its first vertex from the queue, this also means that we relax each

edge at most once, and thus we make $O(m)$ calls to DECREASEKEY. The additional work done by the algorithm can be accomplished in $O(m + n)$ time, so the overall runtime is $O(n)$ times the runtime of EXTRACTMIN plus $O(n)$ times the runtime of INSERT plus $O(m)$ times the runtime of DECREASEKEY.

With the binary heap implementation of priority queues that you wrote in CSCE 221, all three of these operations run in $O(\log n)$ time, so we get an overall runtime of $O(n \log n + m \log n)$. We can do better, however. If we implement our priority queue using a Fibonacci heap, then INSERT and DECREASEKEY both run in $O(1)$ time (amortized), while EXTRACTMIN remains $O(\log n)$. Thus, using this implementation, the runtime for Dijkstra's becomes $O(m + n \log n)$. This is not a data structures course, so we will not discuss how to implement a binary heap, let alone a Fibonacci heap, but you may freely assume a Fibonacci heap implementation of priority queues and therefore assume that DIJKSTRA'S runs in $O(m + n \log n)$ time. $\square$

15.3 Shortest Paths with Constraints: Graph Transformations

Many times in the real world, we face variations of the shortest path problem with additional constraints. That is, there is some property P that we need our $s \rightarrow t$ path to satisfy, and we want to find the shortest $s \rightarrow t$ path from among those with property P . For example, in addition to their length, edges may have a cost, and we want to find the shortest path that isn't too expensive. Or perhaps we need to stop for gas at some point along a trip, so we want to find the shortest route from s to t which passes a gas station.

We could solve problems like this by developing a new algorithm which is some variant of BFS or Dijkstra's and proving that it correctly computes the shortest path with our desired property. Typically an easier approach, however, is to modify our input graph into a format where we can directly apply a graph algorithm we know. This way, we only need to analyze the correctness and runtime of making our modification to the graph, we don't need to worry about a complicated induction proof like the ones we used to prove the correctness of our algorithms in the previous two sections.

One particularly useful technique for modifying graphs is called *graph layering* and it consists of creating multiple copies (layers) of the vertices from the original graph, and strategically choosing which edges should go between vertices in the same layer and which should go between vertices in different layers.

For example, consider the gas station constraint mentioned above. Suppose we are given a list L of n locations and an $n \times n$ array where $A[i, j]$ stores the time to drive from $L[i]$ to $L[j]$ (these times are all non-negative). We are also given a list $S \subseteq L$ of $k \leq n$ locations that are gas stations, as well as a starting location s and an ending location t . We want to find the shortest path from s to t that stops at a gas station.

One way to solve this problem would be to use Dijkstra's algorithm to calculate the shortest path from s to each gas station and then to calculate the shortest path from each gas station to t . If there are many gas stations, however, then this approach may be rather slow. We might need to run Dijkstra's $\Omega(n)$ times (remember that running Dijkstra's once can get us the shortest $s \rightarrow g$ path for every gas station $g \in S$, but it does not get us the shortest path from each gas station to t)

Instead, we can solve this problem with only one call to DIJKSTRA'S if we modify the input graph. We create a weighted directed graph G as follows:

- The vertex set is $L \times \{0, 1\}$. That is, each vertex is a pair of a location and an integer. Vertices of the form $(\ell, 0)$ will represent being at location ℓ before we've made our stop for gas. Vertices of the form $(\ell, 1)$ will represent being at location ℓ after we've stopped for gas
- The edge set contains all edges of the form $((L[i], x), (L[j], x))$ where $1 \leq i, j \leq n$, $x \in \{0, 1\}$, and $i \neq j$. The weight of the edge $((L[i], x), (L[j], x))$ is $A[i, j]$. That is, all the original edges of our graph appear in both layers. Additionally, our edge set contains all edges of the form $((S[i], 0), (S[i], 1))$ where $1 \leq i \leq k$. These edges have weight 0. These additional edges allow us to freely reach layer 1 when we're at a gas station.

To find the shortest $s \rightarrow t$ path that stops for gas, we simply need to find the shortest $(s, 0) \rightarrow (t, 1)$ path in this graph G . Since the only places we can transition from layer 0 to layer 1 in G are at gas stations, every $(s, 0) \rightarrow (t, 1)$ path must stop for gas, so we only need the one call to Dijkstra's.

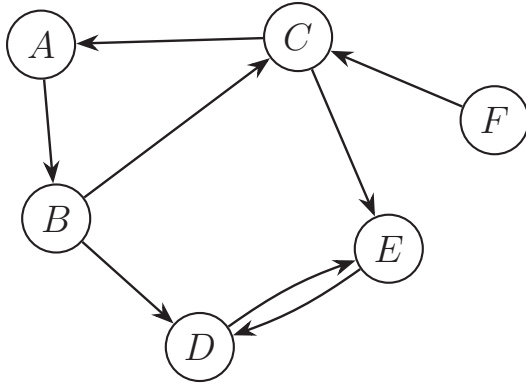
Whenever we use a graph transformation like this we have to be careful when analyzing the runtime. We cannot simply say that this algorithm runs in $O(m \log n)$ time because that's the runtime of Dijkstra's. After all, the runtime of Dijkstra's depends on the number of vertices and edges in the graph that we run it on, not on the size of the inputs to our original problem. We need to figure out how the size of our constructed graph relates to the size of our input(s). We also need to take into account the time to construct the new graph.

In this case, our new graph has $2n$ vertices and $2n(n-1) + k$ edges (each layer has $n(n-1)$ edges within the layer, and there are k edges between the layers). Since $k \leq n$ we can say that the number of vertices is $O(n)$ and the number of edges is $O(n^2)$. Thus, running Dijkstra's on this graph takes time $O(n^2 + n \log n) = O(n^2)$. Constructing the graph can also be done in $O(n^2)$ time, since we only need to perform constant work per vertex and edge, so the overall runtime of this approach is $O(n^2)$.

Sidenote: We could also solve this particular problem without graph layering while still using only a constant number of calls to Dijkstra's. The key idea is to create the reverse graph G^R (see Section 14.4) and run Dijkstra's starting from t in G^R to get the shortest path from each vertex to t .

15.4 Comprehension Questions

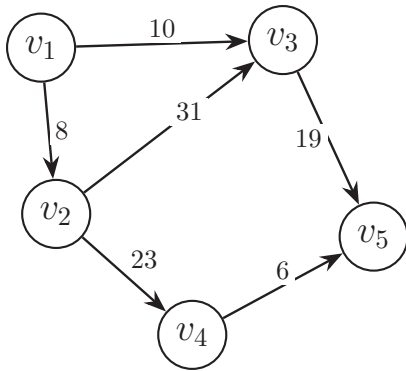
1. After running BREADTHFIRSTSEARCH on the graph below starting from vertex A , what will $pred[D]$ be?
2. After 1 iteration of the while loop of BREADTHFIRSTSEARCH on this graph, Frontier is $\{B\}$. What will Frontier be after the next iteration of the loop?



3. True or false? While running BREADTHFIRSTSEARCH, if $pred[v] \neq Null$, then $D[v] = dist(v)$.
4. Suppose that we are computing shortest paths with Dijkstra's algorithm and we are about to relax the edge (u, v) . Currently $D[u] = 10$, $D[v] = 15$, and $w(u, v) = 3$. Which of these will change after we relax (u, v) , and what will the new value be?
5. What priority queue data structure can we use to get Dijkstra's algorithm to run in $O(m + n \log n)$ time? What runtime does it give us for EXTRACTMIN?
6. Suppose that we are trying to compute shortest paths from A in the graph below (not necessarily with Dijkstra's) and we have the current distance upper bounds:

$$D[v_1] = 0, D[v_2] = 8, D[v_3] = 39, D[v_4] = 31, D[v_5] = 37.$$

Which edges in the graph are currently tense?



15.5 Exercises

1. In this problem, we consider a variation of the shortest path problem. You are given a graph where some of the edges are blue and others are red, and your goal is to find the shortest path which doesn't use too many red edges. More formally, suppose that you are given as input a (undirected, unweighted) graph $G = (V, E)$, a set $R \subseteq E$ of red edges, a positive integer k , and two vertices $s, t \in V$. Give an algorithm which returns a minimum-length path from s to t among those which use at most k red edges, or correctly reports that there is no s to t path in G which uses at most k red edges. The runtime of your algorithm should be $O(k(m+n))$.
2. You are given an directed weighted graph (G, w) with positive edge weights where each edge e has an associated color $c(e) \in \{\text{red, green, blue}\}$. You are also given two vertices $s, t \in V(G)$. We say that a walk $v_1, \dots, v_k$ in G is *tricolor* if it alternates between red, green, and blue edges—i.e. for every $1 \leq i < k$,
 - If $i \bmod 3 = 1$, then (v_i, v_{i+1}) is red
 - If $i \bmod 3 = 2$, then (v_i, v_{i+1}) is green
 - If $i \bmod 3 = 0$, then (v_i, v_{i+1}) is blue

Give an efficient algorithm to find the shortest tricolor walk from s to t in G . If there is no tricolor walk from s to t , your algorithm should correctly report this.

3. Suppose you are interested in finding the shortest paths from a single vertex s in a strongly connected weighted, directed graph (G, w) , where the edge weights are arbitrary integers (could be positive, negative, or zero). A suspicious-looking stranger comes up to you and says that they've already solved this problem. To prove it, they show you a spanning tree T for G rooted at s , which they claim is a shortest path tree. You don't trust this person, so you'd like to check their work, in a way that's faster than simply computing the shortest path tree yourself. Give an algorithm which takes as input G, w, s and T , and determines in $O(m)$ time (where m is the number of edges in G) whether or not T is in fact a shortest paths tree for G rooted at s .
4. A *vertex-weighted graph* is a pair (G, w) where G is a (directed or undirected) graph, and w is a function assigning a weight to each *vertex* of G (w can be represented by an array or a dictionary). The weight (also called the length) of a path in a vertex-weighted graph is the sum of the weights of the vertices in the path.

Show how to solve the shortest paths problem in vertex-weighted graphs. In particular, for each of the following restrictions on (G, w) , give an algorithm which takes as input a vertex weighted graph (G, w) and a vertex s and returns a shortest paths tree rooted at s . Each of your algorithms should be efficient as possible, asymptotically (you should analyze the big- Θ runtime of your algorithms, but you do not need to prove that your algorithm's runtime is the best possible)

- (a) First, suppose that (G, w) is a *directed* vertex-weighted graph where every vertex weight is *positive*.
 - (b) Now suppose that (G, w) is an *undirected* vertex-weighted graph where every vertex weight is *positive*.
5. In the land of Algorithmica, all roads are toll roads. At the end of any trip, you must make a cash payment to the government equal to the total tolls of the roads that you traveled on. John has a map of Algorithmica showing n cities connected by m toll roads. The map also lists the toll $t(r)$ for each road r , which is a positive integer number of cents. John would like to find the cheapest route between city s and city t . Unfortunately, John does not have any pennies, and the toll collectors are very picky and only accept exact change. Give an efficient algorithm to find the cheapest route from s to t whose cost is a multiple of 5.