CS2040S: Data Structures and Algorithms

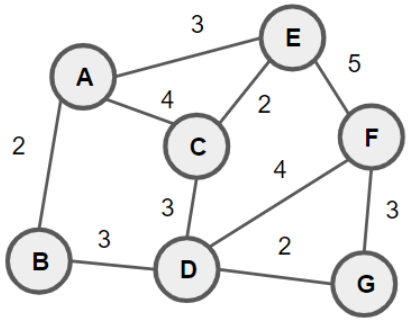
Discussion Group Problems for Week 12

For: April 6–April 10

Goals:

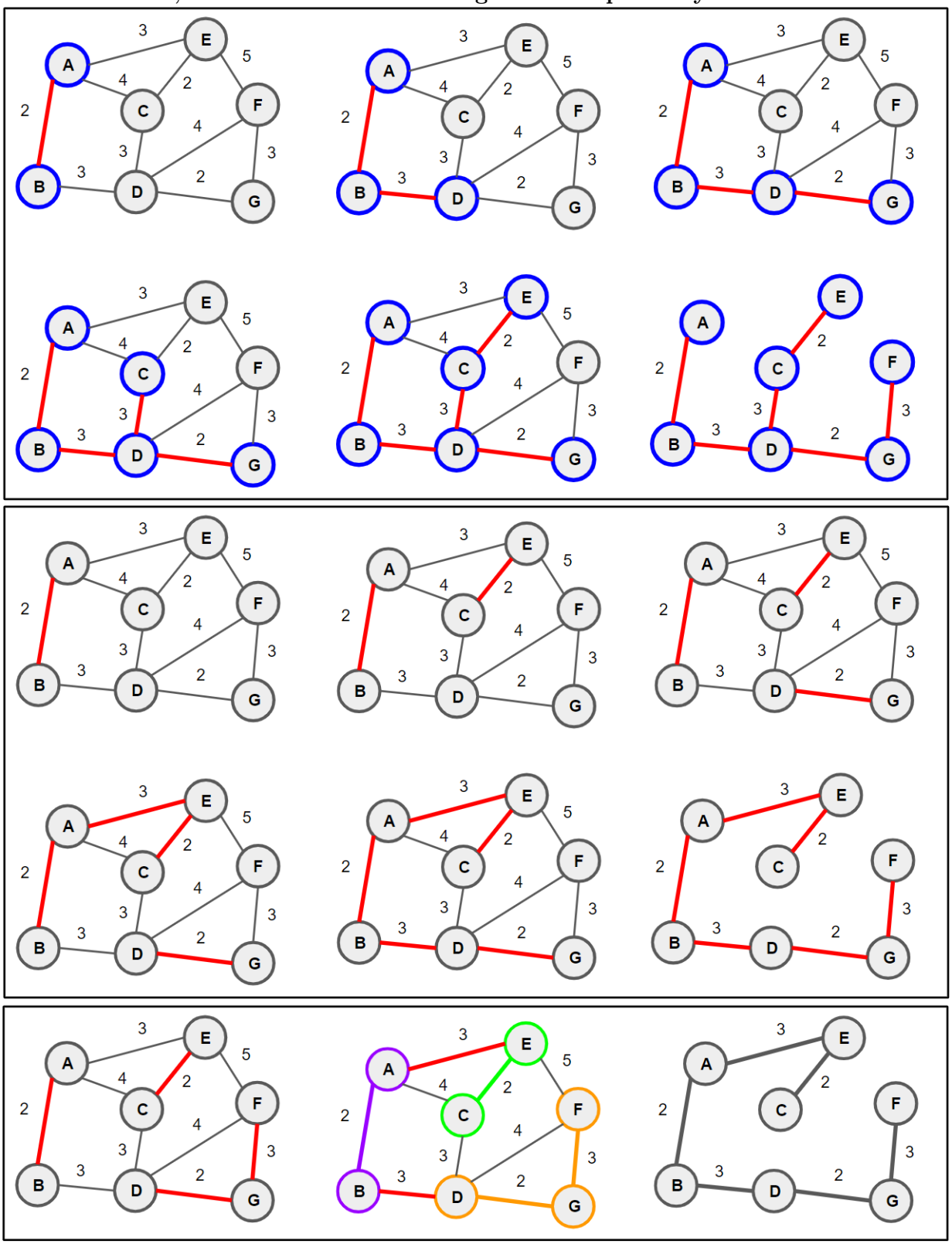
- MST
- Dynamic Programming

Problem 1.



Perform Prim's, Kruskal's, and Boruvka's MST algorithm on the graph above.

Solution: Prim, Kruskal and Burovka's algorithm respectively



Problem 2. Henry Hacker has invented a new divide-and-conquer algorithm for finding a minimum spanning tree! The algorithm is super simple! It only involves x steps:

- Find the median edge weight w_e.
- Partition the edges using the edges into those $\leq w_e$ and those $> w_e$ into two graphs G_a and G_b.
- Recursively find the MST for G_a and G_b, yielding trees T_a and T_b.
- Combine the edges from T_a and T_b, and recursively find the MST for the resulting graph.

After a little bit of thought, Henry realizes that he can speed it up by stopping the recursion early (i.e., not going down to a base case of 1). Instead, if the number of edges in the graph $E < 4V$, then the recursion terminates and he uses Prim's algorithm to find the MST.

How well do you think this algorithm works? (What would happen if it continued to a base case of 1 edge?)

Solution: The key observation is that after the two recursive calls, there are at most $2V$ total edges left, since each of the recursively constructed MSTs has at most V edges. Recall that we can find a median in $O(E)$ time, e.g., using the QuickSelect algorithm. (Let's not worry about the issue of randomization.). Thus the basic recurrence here is:

$$T(V, E) = 2T(V, E/2) + O(E) + T(V, 2V)$$

And remember, the base case says that we use Prim's algorithm if $E < 4V$, so notably the last recursive call is just a direct call to Prim's, so the recurrence is just:

$$T(V, E) = 2T(V, E/2) + O(E) + O(V \log V)$$

Since the base case happens before V edges, the recursion tree is going to have depth at most $\log(E/V)$, and it will have at most E/V leaves and $2E/V$ nodes. Thus the cost of finding the medians is at most $O(E)$ per level of the recursion tree, and hence $O(E \log(E/V))$. And the cost of all the Prim's Algorithms is $O(V \log V)$ per node in the recursion tree, and so is $O((2E/V)V \log V) = O(E \log V)$. So the total cost of the entire algorithm is just $O(E \log V)$, i.e., the same as Prim's and Kruskal's!
There are in fact benefits to this type of approach, e.g., for building cache-efficient MST algorithms (though you will want to do the merge more efficiently so you do not have to run Prim's every time).

Problem 3.

Henry Hacker has another idea for a faster MST algorithm!

Problem 3.a. Recently, he read about *Fibonacci Heaps*. A Fibonacci heap is a priority queue that implements insert, delete, and decreaseKey in $O(1)$ amortized time, and implements extract-Min in $O(\log n)$ amortized time, if there are n elements in the heap. If you run Prim's Algorithm on a graph of V nodes and E edges using a Fibonacci Heap, what is the running time?

Solution: In this case, each node is added at most once from the priority queue, which costs $O(V)$; each edge leads to one decreaseKey operation, which has cost $O(E)$; and each node is extracted once from the priority queue, which results in a cost of $O(V \log V)$. So the total cost is $O(E + V \log V)$.

Problem 3.b. Henry suggest ths following algorithm:

1. Run $O(\log \log V)$ Boruvka steps (and save all blue edges).
2. *Contract* each blue connected component, i.e., build a new graph where each connected component is represented by a single node, and edges represent edges between connected components.
3. Run Prim's algorithm with a Fibonacci Heap on the new graph.

What is the running time of this algorithm?

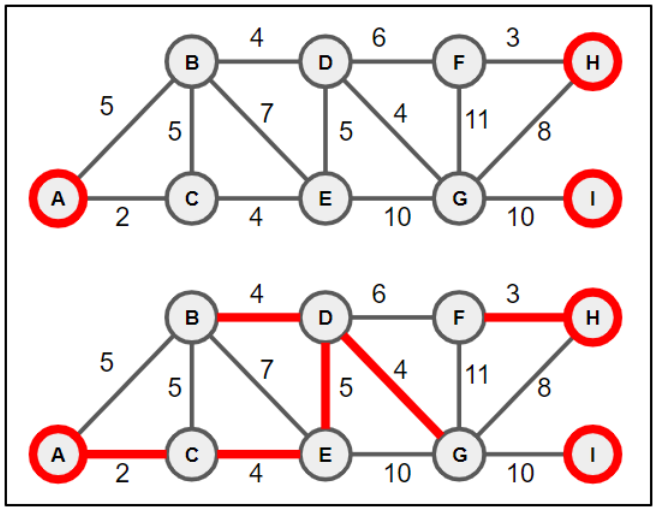
Solution: First, let's think about the Boruvka Steps. Each has $O(E)$ cost, so the total cost is $O(E \log \log V)$. Each Boruvka Step halves the number of connected components. So after $\log \log V$ Boruvka Steps, we have at most $V/\log V$ connected components left.
Second, we can do the contraction step in $O(E)$ time via a DFS. Each component is already labelled, so as we enumerate the edges, we can add edges to the new contracted graph.
Third, we are now running Prim's Algorithm on a graph with at most $V/\log V$ nodes and at most E edges. Using a Fibonacci Heap, this takes time at most $O(E + (V/\log V) \log V)) = O(E + V) = O(E)$ time. So this step takes only $O(E)$ time total!
Altogether, then, the most expensive part is the first step, and so the overall cost is $O(E \log \log V)$ time.

Problem 3.c. Which of these two new variants for finding an MST do you prefer? When is the version from Part (a) better thant he version from Part (b) and vice versa?

Solution: In a dense graph, when $E > V \log V$, then the first algorithm is better, as the running time is just $O(E)$, which is optimal. When the graph is sparse, i.e., $E < V \log V$, then the second algorithm may be better.
It isn't that complicated to use similar ideas to reduce the running time to $O(E \log^* V)$. A much more complicated version gets you $O(\alpha E)$, but no one knows how to do $O(E)$ with a deterministic algorithm.

Problem 4. (Power Plant)

Given a network of power plants, houses and potential cables to be laid along with its associated cost, find the cheapest way to connect every house to at least one power plant. The connections can be routed through other houses if required.



In the diagram above, the top graph shows the initial layout of the power plants (modelled as red nodes), houses (modelled as gray nodes) and cables (modelled as bidirectional edges with its associated cost). The highlighted edges shown in the bottom graph shows an optimal way to connect every house to at least one power plant. Come up with an algorithm to find the required minimum cost. You may assume that there exists at least one way to do so.

Solution: A way to solve this problem is to run Prim's algorithm with the priority queue initialized to contain all of the power plant nodes. This initial setup ensures that every node discovered later can be traced back to exactly one of the power plant nodes.
You can also run Kruskal's or Boruvka's algorithm with all the power plant nodes initialised to be in the same component.
Alternatively, you can insert an additional super node into the graph that is connected to each of the power plant nodes via edges of weight 0, then run any MST algorithm. All of these approaches will run in $O(E \log V)$, where E is the number of potential cables and V is the sum of the number of power plants and houses.

Problem 5. (Traffic Reduction)

In order to reduce traffic, the Singapore government has passed a new law: cars are not allowed to go in circles. If you drive around in a circle, you will be charged a fine. Your job is to deploy a set of cameras to monitor the roads and catch drivers that are violating the new law.

For this problem, you are given a roadmap of Singapore as a connected, undirected graph $G =$

(V, E). For each road (i.e., edge) e you are given the cost $w(e)$ of building a camera station that can detect when the same car passes twice. Your job is to find a minimal set of roads (i.e., edges) such that you can detect any car that makes a circle.

Solution: Find a maximal spanning tree. All the edges that are *not* in the maximal spanning tree are designated for deploying cameras. Given a spanning tree T, any additional edge will form a cycle. So if you deploy a camera on each non-MST edge, then you will catch anyone driving around in a circle. This is the cheapest such set of edges since in a maximal spanning tree, for each cycle, the lightest edge is a *red* edge that is not included in the MST.

Problem 6. Espionage

You are a spy, currently on a mission to obtain intel on the final examination that will be unleashed upon this world by the dreaded *Htes Treblig* within the next 25 days. At the moment, you have a message to send to HQ in the form of a **weighted tree** of N vertices, where all of the edge weights are positive integers.

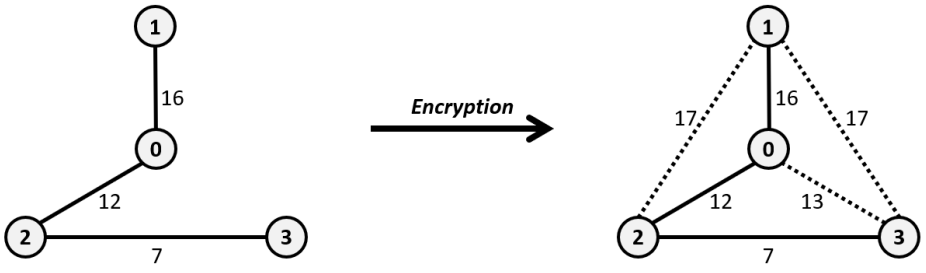
To ensure your message is not at risk of being intercepted, you need to encrypt your message. The way you decide to do this is by inserting additional edges *(with integer weights)* to your tree. The resulting graph, which represents the encrypted message, must satisfy the following properties:

- The resulting graph is a **complete graph** of N vertices (i.e. every pair of vertices in the graph must have exactly 1 edge between them).
- The initial tree is **the unique Minimum Spanning Tree** of the resulting graph.

Inserting edges with large weights will make the encryption process more complicated, so you would like to avoid them wherever possible. The cost of the encryption process is defined as the sum of the weights of the additional edges that are inserted into the graph during the encryption process.

Given the initial message, which consists of the number of vertices in the tree, N, and a list of $N-1$ edges of the tree, **output the minimum possible cost** of encrypting this message.

For example, let the initial message be the tree on the left, consisting of $N = 4$ vertices. The graph on the right represents the optimal way to encrypt the message with a total cost of $17+17+13 = 47$. Therefore, the expected output is "47".



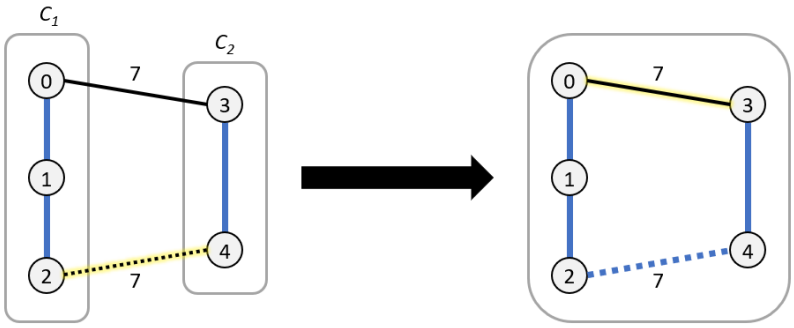
Initial Graph *Encrypted Graph*

Solution: The way to approach this problem is to take an MST algorithm and think about how to force it to select the edges from the initial tree to form the MST. Here, we will use Kruskal's algorithm. We need to ensure that Kruskal's will always select the edges from the initial tree and ignore all of the new edges inserted during encryption.

Focus on a single edge e of weight K in the initial tree. When Kruskal's algorithm selects e to be in the MST, it will union two disjoint connected components. Let these two components be called C_1 and C_2, containing S_1 and S_2 vertices respectively.

We know that among all the edges in the initial tree, e is the only edge connecting C_1 and C_2 *(otherwise there would be a cycle in the initial tree)*. This means there are $(S_1 \times S_2 - 1)$ edges that we need to insert between C_1 and C_2.

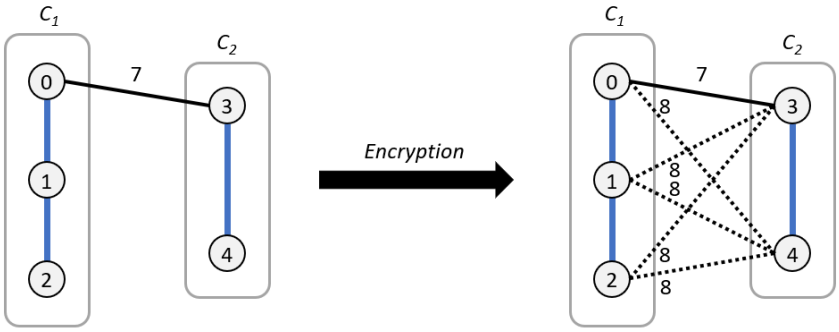
Notice that none of these new edges can have a weight $\leq K$. Otherwise, it is possible for Kruskal's algorithm to process one of the new edges before e. If that happens, then that edge will be selected to be part of the MST instead of e.



Processing edge (2—4)
2 and 4 are in different components.
Add edge (2—4) to the MST.

Processing edge (0—3)
0 and 3 are in the same components.
Ignore edge (0—3)

On the other hand, as long as the new edges have a weight $> K$, then Kruskal's algorithm will definitely process these edges after processing e, where all of the new edges will be ignored since they are already in the same connected component. Therefore, to minimize the total sum of edge weights, we should assign a weight of $(K + 1)$ to all $(S_1 \times S_2 - 1)$ additional edges.



Encryption cost of edges between C_1 and C_2 = $(3 \times 2 - 1) \times (7 + 1)$

Solution: (Continued from previous page)
This gives us the following algorithm:

1. Initialize a variable $cost = 0$
2. Perform Kruskal's Algorithm using the edges of the initial tree as the edge list.
   (a) Let the current edge being processed be (u, v) with weight w.
   (b) Obtain the sizes of the sets containing u and v respectively in the Union-Find Disjoint Set. Let these sizes be S_u and S_v respectively.
   (c) Add $(S_u \times S_v - 1) \times (w + 1)$ to $cost$.
3. Output $cost$.

With N vertices and $(N - 1)$ edges in the initial tree, this algorithm runs in $O(E \log V) = O(N \log N)$ time.

Problem 7. Road Trip
Relevant Kattis Problems:

- https://open.kattis.com/problems/adventuremoving4
- https://open.kattis.com/problems/roadtrip
- https://open.kattis.com/problems/highwayhassle

You are going on a road trip. You are going to get in your trusty car and drive to Panglossia, where you will spend a nice vacation by the beach.

You have already found the best route from your home to Panglossia (using Dijkstra's Algorithm). Next, you need to determine where you can buy petrol along the way. On the road between your home and Panglossia, there are a set of n petrol stations, the last of which is in Panglossia itself. (We will assume that when you reach the last station, your trip is complete.) By searching on the internet, you find for each station: (i) its location on the road; (ii) the cost of petrol. That is, the input to the problem is n stations $s_0, s_1, \ldots, s_{n-1}$, along with $cost(s_i)$, which specifies the cost of petrol at station s_i, and $d(s_i)$, which specifies the distance from station s_{i-1} to station s_i.

Your car's tank holds L liters total. You begin your trip at home with a full tank of petrol. Your car uses exactly 1 liter per kilometer. Along the way, you must ensure that your car always has petrol (though you may arrive at a station just as you run out of petrol). Note that you do not need to fill your tank at a station; you can buy just as much petrol as you choose. Your job is to determine how much petrol to buy at each station along the way so as to minimize the cost of your trip.

You may assume, for simplicity, that L and all the distances are integers, i.e., all the quantities are integers.

Here are two examples, a simple one and a more complicated one.

Example 1: Your tank holds 6 liters, and there are three stations:

```
5km: $1
6km: $2
7km: $4
```

In this case, the best solution costs one dollar, and you purchase one liter of petrol at the first station at a cost of 1 per liter.

Example 2: Your tank holds 20 liters, and there are 10 stations:

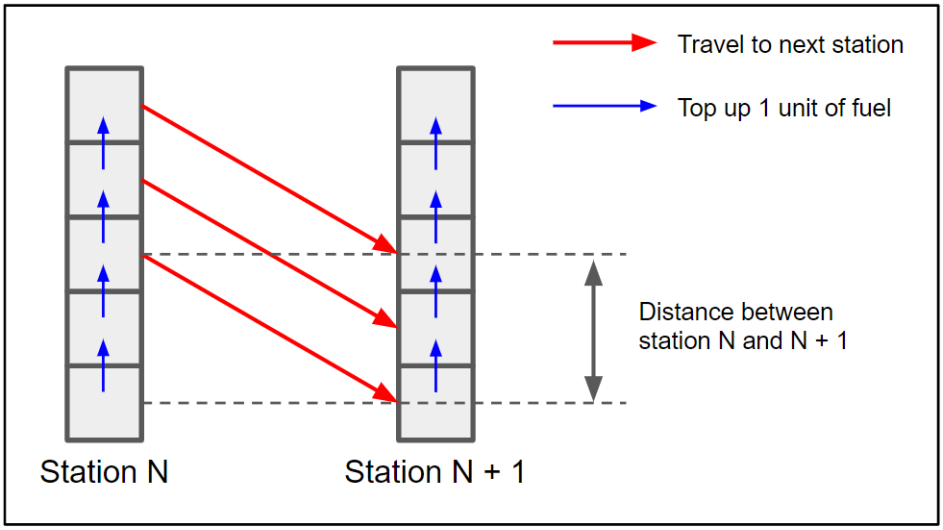
```
7:  $3
17: $5
20: $2
23: $1
39: $5
48: $4
66: $9
83: $9
88: $4
92: $1
```

In this case, the best solution[1] is to purchase petrol as follows, for a total cost of 327 dollars:

```
0
0
3
20
5
20
15
5
4
0
```

A hint. To solve this problem, think about how to calculate the cost $C(v_j, k)$, i.e., the minimum cost to get from station v_j to the destination, assuming you start at station v_j with k liters of petrol.

[1] According to the unvalidated belief of myself...



Solution: This is a typical dynamic programming style problem where the subproblems involve computing $C(v_j, k)$ for each station v_j and each value of k. You can observe that if you already have solved the problem for all $j' > j$, then it is not hard to compute $C(v_j, k)$: if you start with k liters and it is k' kilometers to the next station, then for all i, lookup the (already computing) $C(v_{j+1}, k - k' + i)$, which tells you the minimum cost for arriving at v_{j+1} with $k - k' + i$ liters of petrol. You can add to this the cost per liter at v_j times i, and choose the i that is cheapest.
Then, working in reverse until you get back to v_1, you will eventually discover the cheapest way to reach your destination.