CS2040S Tutorial Week 10 - Quick solutions
AY2019/20 Semester 1

1. Easy Money

(a) **Problem summary:** You are given a list of V currencies and the exchange rates between the currencies. Let $R(x, y)$ be the exchange rates between currencies x and y, where 1 unit of currency x can be exchanged for $R(x, y)$ units of currency y. You may assume that any two currencies can be directly for each other (i.e. $R(x, y)$ exists for all currencies x and y).

You will be given multiple queries. Each query consists of two currencies, x and y. You are to output a sequence of exchanges that allows you to obtain the maximum number of units of y given 1 unit of x. Assume that arbitrage is impossible.

Perform some pre-processing so that each query can be answered as quickly as possible.

Solution: We can transform this problem into finding the shortest path between any two vertices in a graph.

Let $x \to a_1 \to a_2 \to \cdots \to a_k \to y$ be a sequence of exchanges from currencies x to y. The overall exchange rate is given as such:

$$Rate_{overall} = R(x, a_1) \times R(a_1, a_2) \times \cdots \times R(a_k, y)$$

Now, suppose we take the negative logarithm of both sides.

$$\begin{aligned} -\log(Rate_{overall}) &= -\log\left(R(x, a_1) \times R(a_1, a_2) \times \cdots \times R(a_k, y)\right) \\ &= (-\log R(x, a_1)) + (-\log R(a_1, a_2)) + \cdots + (-\log R(a_k, y)) \end{aligned}$$

By taking the logarithm of both sides, we have transformed the right hand side from a product to a sum. And by negating both sides, the goal of finding the maximum overall rate corresponds to minimizing the sum of $-\log(R(a, b))$.

Construct a complete graph of V vertices, where the V vertices correspond to the V currencies. For any two currencies u and v, have a directed edge from u to v with weight $-\log(R(u, v))$. The goal of maximizing the rate between two currencies u and v corresponds to finding the shortest path from u to v in this graph.

During the pre-processing step, run **Floyd-Warshall's algorithm** to obtain the shortest path between any pair of vertices. The optimal rate between any pair of currencies u and v is given by $e^{-dist(u,v)}$, where $-dist(u, v)$ represents the weight of the shortest path from u to v in the graph.

To reconstruct the sequence of exchanges to perform, maintain a predecessor array. Let's call it $prev$. $prev[u][v]$ will store the vertex directly before v in the shortest path from u to v. For example, if the shortest path were $u \to a_1 \to a_2 \to a_3 \to v$, then $prev[u][v] = a_3$. I have already discussed how to reconstruct the path given such an array, so I won't explain it again here.

To construct this predecessor array, modify the relaxation step of Floyd-Warshall's algorithm slightly. Within Floyd-Warshall, suppose we're considering the intermediary vertex w in the shortest path between u and v.

```
if (dist[u][w] + dist[w][v] < dist[u][v]) {
    // Update shortest distance
    dist[u][v] = dist[u][w] + dist[w][v];
}
```

Let the (current) shortest path from u to w, and from w to v, be as follows:

$$path(u \to w) : u \to a_1 \to a_2 \to \cdots \to a_i \to w$$
$$path(w \to v) : w \to b_1 \to b_2 \to \cdots \to b_j \to v$$

Currently, $prev[u][w] = a_i$ and $prev[w][v] = b_j$. If the relaxation step results in an update in the shortest path from u to v, then the new path would be as follows:

$$path(u \to v) : u \to a_1 \to \cdots \to a_i \to w \to b_1 \to \cdots \to b_j \to v$$

Therefore, the new updated value of $prev[u][v]$ is $b_j = prev[w][v]$.

```
if (dist[u][w] + dist[w][v] < dist[u][v]) {
    // Update shortest distance
    dist[u][v] = dist[u][w] + dist[w][v];
    // Update predecessor array
    prev[u][v] = prev[w][v];
}
```

Time complexity analysis

Pre-processing:

- Building the graph: $\mathcal{O}\left(V^2\right)$
- Floyd-Warshall's algorithm: $\mathcal{O}\left(V^3\right)$
- Overall: $\mathcal{O}\left(V^3\right)$

Answering each query:

- Outputting the optimal rate: $\mathcal{O}\left(1\right)$
- Constructing the optimal sequence of exchanges: $\mathcal{O}\left(V\right)$
  (A shortest path in a graph of V vertices is of length at most $V-1$)
- Overall: $\mathcal{O}\left(V\right)$

(b) **Problem summary:** We say that a currency x is *exploitable* if you can start with 1 unit of currency x, go through a sequence of exchanges, and end up with more than 1 unit of x. Using your solution from 1(a), determine if there is **any** exploitable currency.

Solution: Let x be an exploitable currency, and $x \to a_1 \to \cdots \to a_k \to x$ be the sequence of exchanges to exploit the currency. x being exploitable means the overall rate is larger than 1.

$$Rate_{overall} = R(x, a_1) \times \cdots \times R(a_k, x) > 1$$
$$-\log(Rate_{overall}) < 0$$

In our constructed graph, this corresponds to a negative cycle, so the problem is transformed to "finding a negative cycle in a graph".

Short answer: After running Floyd-Warshall's algorithm, iterate through the vertices. If there is a vertex v such that $dist[v][v] < 0$, then there exists a negative cycle in the graph, and hence, there is exists a currency that is exploitable. Otherwise, there is no exploitable currency.

Time complexity analysis

- Building the graph and running Floyd-Warshall's: $\mathcal{O}\left(V^3\right)$
- Checking $dist[v][v]$ for all vertices: $\mathcal{O}\left(V\right)$
- Overall: $\mathcal{O}\left(V^3\right)$

Long answer: Note that the short answer will not be able to find *every* exploitable currency.

In Floyd-Warshall's algorithm, after the w-th iteration, you're guaranteed that $dist[u][v]$ contains the shortest path from u to v that uses some subset of $\{0, 1, \ldots, w-1\}$ as intermediary vertices. However, note that $dist[u][v]$ may not have found a shortest path that uses the intermediary vertices multiple times. For example, consider the following path from vertices 4 to 5 as follows:

$$4 \to 0 \to 1 \to 2 \to 0 \to 1 \to 2 \to 0 \to 1 \to 2 \to 5$$

Notice that this path uses only vertices 0, 1 and 2 as intermediary nodes, but uses those intermediary nodes three times each. Because they are used more than once, it is not guaranteed that Floyd-Warshall's algorithm has considered this path after the 3rd iteration. You can only guarantee that Floyd-Warshall's has considered every path that uses each of the intermediary nodes **at most once**.

This is why the short answer may not be able to find *every* exploitable currency. Suppose some currency x needs to go through some negative cycle 5000 times before x can be exploited.

$$x \to \underbrace{(0 \to 1 \to 2) \to (0 \to 1 \to 2) \to \cdots \to (0 \to 1 \to 2)}_{5000 \text{ times}} \to x$$

In such a case, it's not guaranteed that $dist[x][x] < 0$ by the time Floyd-Warshall's terminates.

However, if there are some exploitable currencies, we can guarantee that the algorithm described in the short answer will find *at least one*.

Consider the smallest[1] negative cycle: $v \to a_1 \to \cdots a_k \to v$. Since this cycle is the smallest, that means all of the vertices in the cycle are distinct.

Let $a_{\max}$ be the largest among the intermediary vertices.

$$a_{\max} = \max\{a_1, a_2, \ldots, a_k\}$$

Based on what we have discussed earlier, that means this cycle is guaranteed to have been considered at the end of the $(a_{\max} + 1)$-th iteration. Therefore, we can guarantee that $dist[v][v] < 0$ by the end of that iteration, and hence, by the time Floyd-Warshall's terminates.

Therefore, if there are some exploitable currencies, the algorithm described in the short answer will report it.

2. Comfy Taxi

(a) **Problem summary:** You're given a directed, unweighted graph of V vertices and E edges, represented by an adjacency list, as well as a non-negative integer k.

You will be given multiple queries. Each query consists of two vertices, u and v. Determine if there is a path from u to v that uses *exactly* k edges. You are allowed to use vertices and edges multiple times.

Solution: First, let's simplify the problem to a single fixed source vertex. We can't use BFS or any other shortest-path algorithm as-is, since the shortest path may not necessarily be the path we're aiming for.

We can, however, modify BFS so that it re-visits vertices that have been visited before. However, we have to be careful when doing so; naively removing the 'visited' check may cause the BFS to run for an exponential amount of time in certain graphs.

We'll refer to the source vertex as s. Let S_i represent the set of vertices that are reachable from the source in exactly i steps. Our goal is to generate S_k.

Initially, $S_0 = \{s\}$. Now, given S_i, we can generate S_{i+1}. Consider any vertex $v \in S_i$. Since we can reach v from s in exactly i steps, we can reach any neighbour of v from s in exactly $(i+1)$ steps. Therefore, S_{i+1} consist of the neighbours of the vertices in S_i.

$$S_{i+1} = \bigcup_{v \in S_i} neighbours(v)$$

To represent these sets S_i, we can use a HashSet, since it prevents duplicates.

Therefore, the modified BFS can look like this:

1. Initialise a HashSet $S_0 = \{s\}$.
2. For $i = 0$ to $k - 1$:
   2.1. For each vertex $v \in S_i$, insert $neighbours(v)$ into S_{i+1}

Note that this is the case for a single source. We'll have to repeat this using each vertex as the source.

Time complexity analysis

Pre-processing:

- Generating each set S_{i+1}: $\mathcal{O}(V + E)$
  - In the worst case, the set S_i can contain all the vertices in the graph.
  - Hence, iterating through S_i takes $\mathcal{O}(V)$ time.
  - The total number of insertions performed is given by $\sum_{v \in S_i} d(v)$, where $d(v)$ represents the (outgoing) degree of v.
  - Recall that the sum of the degrees of all the vertices in the graph is equal to the number of edges in the graph.
  - Hence, $\sum_{v \in S_i} d(v) \leq E$, which means all of the insertions will take $\mathcal{O}(E)$ time.
- There are k sets to generate.
- Repeat this process using each of the V vertices as the source vertex.
- Overall: $\mathcal{O}(kV(V + E))$

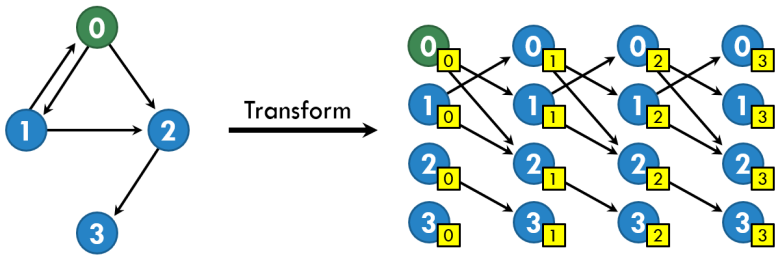
Answering each query:

- To answer a query (u, v), check if v exists in S_k generated with u as the source.
- Overall: $\mathcal{O}(1)$

[1] Smallest in terms of number of vertices. In other words, a negative cycle that has the smallest number of vertices

Alternatively, what if we can imbue the information of 'how many steps have been taken' into the graph itself?

Construct a new graph where each vertex is represented using two values: (v, i), which represents the event of "reaching vertex v (from the source) in exactly i steps". For every edge (u, v) in the original graph, we'll have an edge from $(u, 0)$ to $(v, 1)$, stating that "If it's possible to reach vertex u in 0 steps, then it's possible to reach vertex v in 1 step." Similarly, have an edge from $(u, 1)$ to $(v, 2)$, from $(u, 2)$ to $(v, 3)$ and so on.



Example of the construction of the new graph given the graph on the left

Now, to determine if some vertex v is reachable from the source in exactly k steps, we can simply check, in our new graph, whether the vertex (v, k) is reachable from $(s, 0)$. This can be done simply using a BFS or DFS.

Time complexity analysis

- For every vertex v in the old graph, we duplicate it a total of $k + 1$ times (From $(v, 0)$ to (v, k)). Therefore, the number of vertices in this new graph is $V' = (k + 1)V$ and the number of edges is $E' = kE$.
- Construction of new graph: $\mathcal{O}(V' + E') = \mathcal{O}(k(V + E))$
- BFS/DFS from a single source on this new graph takes $\mathcal{O}(V' + E') = \mathcal{O}(k(V + E))$
- Repeat the BFS/DFS using $(v, 0)$ as the source vertex for each vertex v in the original graph.
- Overall: $\mathcal{O}(kV(V + E))$

Can we do better? Possibly take advantage of the fact that we're checking all the paths for every source?

Well, for dense graphs and large values of k, yes we can. You might remember a result from CS1231S, where taking the adjacency matrix A and squaring it (i.e. $A \cdot A$ where $\cdot$ represents matrix multiplication) will result in a matrix where the (i, j) entry represents the number of paths from vertices i to j of length exactly 2.

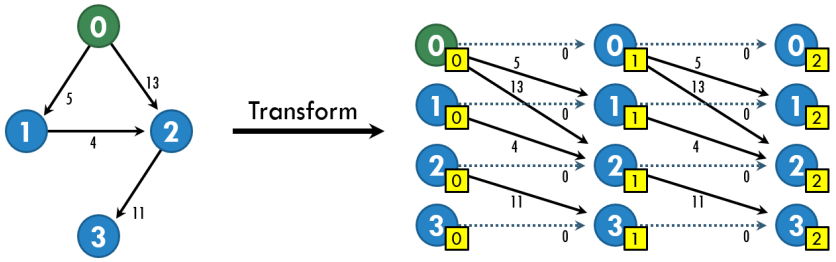
With a slightly modified 'matrix multiplication' operation (specifically, replacing the operators $\times$ and $+$ with $\wedge$ and $\vee$ respectively), we can calculate A^k in $\mathcal{O}\left(V^3 \log k\right)$ (or possibly faster using other matrix multiplication methods). However, this isn't the *main* solution, so we won't discuss it further here.

(b) **Problem summary:** You're given a directed, weighted graph of V vertices and E edges, with non-negative weights, represented as an adjacency list, as well as a non-negative integer k.

You will be given multiple queries. Each query consists of two vertices, u and v. Find the length of the shortest path from u to v that uses no more than k edges.

Solution: We'll use the graph-duplicating technique as mentioned in the alternative solution for 2(a).

To relax the constrain from *exactly k edges* to *at most k edges*, for each vertex v in the original graph, we can add an additional edge of weight 0 from $(v, 0)$ to $(v, 1)$ to represent "sitting at vertex v without moving". We'll add similar edges from $(v, 1)$ to $(v, 2)$, from $(v, 2)$ to $(v, 3)$ and so on.



Example of the construction of the new graph given the graph on the left

Now, the problem of finding the shortest path vertices u to v that uses at most k edges in the original graph is equal to finding the shortest path from $(u, 0)$ to (v, k) in the new graph.

We can run Dijkstra's algorithm to find that shortest path, and that will work, but we can do better. Since edges will always go from an 'earlier timestamp' to a 'later timestamp', there are no cycles in our new graph. Therefore, it is a Directed Acyclic Graph.

Finding the shortest path on a DAG can be done by processing the vertices in topological order. For this graph, a valid topological ordering can simply be 'all the vertices, sorted by increasing timestamp'.

Time complexity analysis

- The new graph has $V' = (k+1)V$ vertices and $E' = kE + kV$ edges.
- Construction of new graph: $\mathcal{O}(V' + E') = \mathcal{O}(k(V+E))$
- Running a shortest path algorithm on this DAG takes $\mathcal{O}(V' + E') = \mathcal{O}(k(V+E))$.
- Repeat using $(v, 0)$ as the source vertex for each vertex v in the original graph.
- Overall: $\mathcal{O}(kV(V+E))$