CS170Hw4

Trustin Nguyen

September 27, 2024

Study Group

Rohan Kudchadker: 3038065006

Sparsity and SCCs

Call a directed graph G *weakly-connected* if, upon making all edges bidirectional (i.e. turning the directed graph into an undirected graph), the graph is connected.

(a) Argue that every weakly-connected directed graph G with n vertices and $n-1$ edges must have n strongly connected components.

Answer. Since G's undirected representation is connected and the graph has n vertices, $n-1$ edges, then it must be a tree. Trees don't have cycles. The claim is that each vertex lies in its own strongly-connected component. Suppose that a SCC had more than one vertex. Then we can pick two vertices, v_1, v_2 and find a directed path $v_1 \to v_2$ and a directed path $v_2 \to v_1$. This means there is a cycle. Contradiction.

(b) Consider a weakly-connected directed graph G with $n \geqslant 3$ vertices and n directed edges such that for every vertex, v, there are exactly two directed edges incident to v. For each k from 1 to n, describe conditions upon which there are k strongly-connected components.

Answer. Since G is weakly connected, with n edges, each vertex of degree 2, removing one edge gives a tree, and therefore putting it back means we connect two leaves. So the whole graph is a single cycle.

So if we remove the edge, by part a, there are then n strongly-connected components. Adding the edge back means that we either generate one large cycle of size n or there remains 0 cycles. This is clear from the fact that readding the edge cannot create a cycle of size $< n$, otherwise, the points outside the cycle must still be connected to the graph, and therefore, the vertices will have degree > 2.

Now, we can classify the sccs. If there is one scc, it is a graph with a directed path, starting at a point, that passes through all points, and back to itself. If there are n sccs, then we can have a directed path that passes through each vertex, but is not able to cycle back to the original start. It is impossible to have k sccs with $1 < k < n$.

(c) From the graph described in part (b), describe a *simple* and efficient algorithm using *only one* DFS or BFS traversal that determines the number of SCCs G has.

Answer. Start DFS at a random point, and see if it ever can revisit that start point. If it does, then we know that the entire graph is a cycle, so there is one SCC. Otherwise, there are n SCCs.

(d) Now, consider any weakly-connected directed graph G′ with $n \geqslant 3$ vertices and n directed edges. Modify your algorithm from part (c) to efficiently count the number of SCCs G′ has (still using at most one DFS or BFS traversal).

Answer. If we take some edge away, then the graph will be a tree. This comes down to what kind of edge we take away, but one does exist, because our weakly connected graph has a spanning tree in the bidirectional version. So the $n-1$ edge version has n strongly connected components by part a. If we add back that edge, we generate at most 1 cycle. So the algorithm should check if there is a cycle or not and if the orientation of the edges allow for a SCC.

We start the algorithm picking a random vertex to start DFS. Then we traverse the graph as a non directed graph, where we don't care if the edges are reversed or not. When we run into a cycle, we try to run dfs on a vertex we already visited, take note of that edges orientation and backtrack. Make sure that the edges in the backtracked path are consistent, and if they are, record the length of that cycle.

Finally, the number of SCCs will be n - the size of the cycle if the orientation of edges was consistent. Else, the number of SCCs will be 1.

2-SAT

In the 2SAT problem, you are given a set of clauses, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value true or false to each of the variables so that all clauses are satisfied – that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

Recall that $\vee$ is the logical-OR operator and $\wedge$ is the logical-AND operator and $\overline{x}$ denotes the negation of the variable x. This instance has a satisfying assignment: set x_1, x_2, x_3, and x_4 to $\mathtt{true}$, $\mathtt{false}$, $\mathtt{false}$, and $\mathtt{true}$, respectively.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes: one of each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I (where α, β are literals), G_I has an edge from $\overline{\alpha}$ to β, and one from the $\overline{\beta}$ to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to each of the implications $\overline{\alpha} \implies \beta$ and $\beta \implies \alpha$. In the sense, G_I records all implications in I.

(a) Show that if G_I has a strongly connected component containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

Answer. This means that there is a path from x to $\overline{x}$, so $x \implies \overline{x}$. So this means $\overline{x} \vee \overline{x} = \overline{x}$ is a clause. But there is also a path from $\overline{x}$ to x, so $x \vee x = x$ is also a clause. Then in the 2SAT, we have $x \wedge \overline{x}$ which is false.

(b) Now show the converse of (a): namely, that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable.

Answer. We can pick a sink SCC and set all the values to true, which is allowed since a variable and its negation are not in the same SCC. We know that all variables v in the sink SCC show up as the second argument of an implication $\implies$. So we are guaranteed that none of the clauses we evaluate will be false. So we are left with a smaller 2SAT problem after setting all sink SCC values to true. This is a subgraph of G_I, so we can repeat until the 2SAT is solved.

(c) Conclude that there is a linear-time algorithm for solving 2SAT. Provide the algorithm description and runtime analysis; proof of correctness is not required.

Answer. The runtime of the algorithm is the time to find the SCCs which is $\mathcal{O}(|2V + 2E|)$ and the runtime to set the truth values $\mathcal{O}(|2V|)$. So the runtime is

Road Trip

The CS 170 staff are preparing to drive from Berkeley to Chicago to attend a computer science conference!

Assume that the roads from Berkeley to Chicago can be expressed as a directed weighted graph $G = (V, E, c)$, where each $v \in V$ represents a city, each $(u, v) \in E$ represents a road from city u to city v, and $c(u, v)$ represents the cost of gas required to drive from city u to city v. Each road takes exactly one day to traverse, and after driving across road (u, v), the staff will stop in city v overnight to rest.

Unfortunately, there are too many course staff, so they will have to rent two separate cars for the trip, and each city along the way can only accommodate one group of staff per night. Both cars start in Berkeley and depart on the same day.

Additionally, each group has to pay r dollars per day in car rental fees. They are allowed to spend a day in any city without driving, but they still have to pay the rental fee for that day. Once a group arrives in Chicago, the group will return the car and does not have to pay any more rental fees.

Design an efficient algorithm to find the route that minimizes the total cost of the trip, including gas and rental fees. (You may assume that no matter the route, the staff will always arrive in Chicago before the conference starts.)

Provide a 3-part solution with runtime in terms of $n = |V|$, $m = |E|$.

Answer. Here is the pseudocode using Djikstra's

```
function minTrip(start, end, r, G(V, E, c))
    queue1 ← PriorityQueue()
    queue2 ← PriorityQueue()
    path1, path2 ← [-1]*len(V), [-1]*len(V)
    currCost1, currCost2 ← [math.inf]*len(V), [math.inf]*len(V)
    for n in start.neighbors
        queue1.insert(n, c(start, n)), currCost1[n] = c(start, n)
        queue2.insert(n, c(start, n)), currCost2[n] = c(start, n)
    queue1.insert(start, r)
    queue2.insert(start, r)

    while True
        city1, cost1 ← queue1.pop()
        city2, cost2 ← queue2.pop()
        if city1 == end or city2 == end:
            break

        while city1 == city2
            queue2.insert(city2, cost2 + r)  # Wait one day
            city2, cost2 ← queue2.pop()

        for n in city1.neighbors
            newCost = cost1 + c(city1, n)
            if not visited[n] and newCost < currCost1[n]
                queue1.insert(n, newCost)
                path1[n], currCost1[n] = city1, newCost

        for n in city2.neighbors
            newCost = cost2 + c(city2, n)
            if not visited[n] and newCost < currCost2[n]
```

```
                        queue2.insert(n, newCost)
                        path2[n], currCost2[n] = city2, newCost

    path, queue, currCost = path1, queue1, currCost1
    if path2[end] == -1:
        path, queue, currCost = path2, queue2, currCost2

    while True
        city, cost ← queue.pop()
        if city == end
            break
        for n in city.neighbors
            newCost = cost + c(city, n)
            if not visited[n] and newCost < currCost[n]
                queue.insert(n, newCost)
                path[n], currCost[n] = city, newCost

    return path1, path2
```

Proof. The algorithm runs dijkstra's for both cars at the same time, allowing one of the cars to wait for one day if needed. Selecting which car to wait for one day does not matter, as that just increases the total cost by r no matter which we choose.

We know that one car always takes priority over the other and never waits for the other. So Djikstra's algorithm gives the optimal path for one car. So if there existed another optimal travel plan for the two cars, both of the cars in the optimal travel plan would spend more money than the car that finished first in this algorithm. Let us name the two cars in the optimal travel plan C_1, C_2 and the one that finished first in our algorithm F_1. Then we know that C_1, C_2 must have both waited at least one day in some city. Let the number of times that C_1 waits total w_1 and w_2 for C_2. But recall that it does not matter which car does the waiting, so we can just make car C_2 do all the waiting and car C_1 none at all. □

Runtime Analysis: The runtime is $O((|V| + |E|) \cdot \log |V|)$ because we run Dijkstra's two times, once for each car.

Shortest Path with Clusters

Sometimes, we can exploit the structure of a graph to come up with faster shortest-path algorithms, as we saw in class with Dijkstra's algorithm and Shortest Path in DAGs algorithms.

In this problem, consider a graph $G = (V, E, w)$ with possibly negative edge weights and the guarantee that every strongly connected component of G has at most k vertices for some fixed $k \in \mathbb{N}$.

Design an algorithm to find the shortest path from a source vertex s to a target vertex t in G that runs in time asymptotically faster than $O(|V| \cdot |E|)$ when k is asymptotically smaller than $|V|$, and does not run slower than $O(|V| \cdot |E|)$ if $k = \Theta(|V|)$.

Give a 3-part solution.

Answer.

```
def shortestPath(G(V, E, w), s, t)
    SCCs ← kosaraju(G)
    SCCs = topologicalSort(SCCs)

    costs ← [inf] * |V|
    parents ← [-1] * |V|
    currSCC = SCCs.find(contains(s))
    while True
        parents ← BellmanFord(currSCC)
        if t in currSCC.V
            break
        nextSCC = getNextTopological(currSCC) # the next scc in the
        ↪ topological ordering.
        for v in nextSCC.vertices
            edges = getEdgesInto(v)
            setCosts(edges, costs) # for every edge coming into our
            ↪ scc, set the cost for each vertex in our scc
        currSCC = nextSCC

    return backtrack(parents, start=s, end=t)
```

Proof. The algorithm first finds the sccs of the graph and computes the topological ordering we will traverse the sccs in. For the current scc, we use BellmanFord algorithm to find the best path from the start vertex s to that node. Then we go to the next scc, set the initial values based on the costs of the edges going into our next scc and rerun BellmanFord. We repeat until we find the SCC with the end vertex t in it. At every step, since we run BellmanFord, we know that within that scc, the shortest path from every vertex to the root is computed correctly. Now suppose we are in a new scc that does not contain s. Then we first relax the cost to those vertices with its cost to the start vertex s. Running BellmanFord on our scc will then give us the best path out of all paths of length $1, \ldots, 2k$. By induction, we know that after we reach vertex t, we will have found the shortest path. We can stop when we find a scc with t because our SCCs were topologically sorted. □

Runtime Analysis: Running kosaraju will take $O(|V| + |E|)$ time. Then for label each SCC by its edge set and vertex set $E_1, E_2, \ldots, E_n, V_1, V_2, \ldots, V_n$. We have:

$$\sum |E_i| < |E| \qquad \sum |V_k| = |V|$$

Now we run bellmanford for every SCC in $O(|V_i||E_i|)$. Each SCC has at most k vertices, so this reduces to

$$k \sum \mathcal{O}(|E_i|) \leqslant k \cdot \mathcal{O}(|E|)$$

where $k \leqslant \mathcal{O}(|V|)$. So total runtime is $\mathcal{O}(|E| + |V|) + k\mathcal{O}|E|) = k\mathcal{O}(|E|)$

Arbitrage

Shortest-path algorithms can also be applied to currency trading. Suppose we have n currencies $C = \{c_1, c_2, \ldots, c_n\}$: e.g., dollars, Euros, bitcoins, dogecoins, etc. For any pair of currencies c_i, c_j, there is an exchange rate $r_{i,j}$: you can buy $r_{i,j}$ units of currency c_{jj} at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geqslant 0$ for all i, j.

The Foreign Exchange Market Organization (FEMO) has hired Oski, a CS170 alumnus, to make sure that it is not possible to generate a profit through a cycle of exchanges; that is, for any currency $i \in C$, it is not possible to start with one unit of currency i, perform a series of exchanges, and end with more than one unit of currency i. (That is called *arbitrage*)

More precisely, arbitrage is possible when there is a sequence of currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. This means that by starting with one unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for profit.

We say that a set of exchange rates is arbitrage-free when there is no such sequence, i.e. it is not possible to profit by a series of exchanges.

(a) Give an efficient algorithm for the following problem: given a set of exchange rates $(r_{i,j})_{i,j \in n}$ which is *arbitrage-free*, and two specific currencies a, b, find the most profitable sequence of currency exchanges for converting currency a into currency b. That is, if you have a fixed amount of currency a, output a sequence of exchanges that gets you the maximum amount of currency b.

For part (a), give a 3-part solution.

Answer. Here is the pseudocode:

```
def getBestExchange(G, start, end)
    parents ← [-1] * |V|
    costs ← [0] * |V|
    costs[start] = 1
    for i in range(|V| - 1)
        for (u, v) in E
            currCost = cost[u] * v
            if currCost > costs[v]
                cost[v] = currCost
                parents[v] = u
    return backtrack(parents, start, end)
```

Proof. This is using the BellmanFord algorithm. At each step, we find the best path from the start node to another of length 1, then 2, and so on. Because we relax each visited vertex each time. So after the $|V| - 1$-th step we will have found the best path considering all paths of length $1, 2, \ldots, |V| - 1$ from the start to the end node. □

Runtime Analysis: We iterate over every edge $|V| - 1$ times, doing constant work at each iteration. So the runtime is $O(|V| \cdot |E|)$.

(b) Oski is fed up of manually checking exchange rates, and has asked you for help to write a computer program to do his job for him. Give an efficient algorithm for detecting the possibility of arbitrage, and state the runtime of your algorithm. *Proof of correctness is not required.*

Answer. We would run the BellmanFord from the previous question an extra step or for $|V|$ iterations. At this step, we check to see if there is any relaxation. If there is, we know that there is arbitrage because there should not be a cycle of length $|V|$ that results in an increase of currency. The runtime is $\mathcal{O}(|V| \cdot |E|)$ because it is BellmanFord with one extra step.