CS 170 Homework 4

Due **Friday 9/27/2024, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, explicitly write "none".

2 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.

(b) Consider a weakly-connected directed graph G with $n \geq 3$ 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.

Hint: this graph structure is very special. If $n = 5$, is it possible for there to be exactly two strongly connected components?

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

Note: you may *not* use the SCC-finding algorithm from lecture or another black-box SCC algorithm such as Tarjan's algorithm.

(d) Now, consider any weakly-connected directed graph G' with $n \geq 3$ 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).

Solution:

(a) First, in order for G to be weakly connected, there cannot be v_i and v_j such that (v_i, v_j) and (v_j, v_i) are both in $E(G)$ because we don't have enough edges to weakly connect the graph otherwise. Now, the corresponding undirected graph will look like a tree, so there is only one undirected path between every pair of edges in this graph. Thus, there cannot be a directed path from v_i to v_j and a different directed path back (as this will form two separate undirected paths).

(b) Note that, in order for G to be weakly connected and for each vertex to be incident to two other edges, the corresponding undirected graph G must be exactly a single cycle. Without loss of generality, label the vertices $v_1, v_2, \cdots, v_n$ such that there exists an edge (in either direction) between v_1 and v_n, and also v_i and v_{i+1} for all $i \in \{1, 2, \cdots, n-1\}$.

Now, if all edges are facing the same direction (i.e. $E(G) = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$), then this is a single directed cycle and there is only one strongly connected component. Otherwise, we claim there are no two vertices that are strongly connected to each other. Consider an arbitrary $v_a, v_b \in V(G)$, with $a < b$. If there is a path from v_a to v_b, then it must either be $v_a, v_{a+1}, \cdots, v_b$, or $v_a, v_{a-1}, \cdots, v_1, v_n, v_{n-1}, \cdots v_b$. Then, if there is a path from v_b to v_a, it must be along the other sequence of vertices. Thus, this will form a directed cycle.

Thus, if it's a directed cycle, there is a single SCC. Otherwise, there are n single-node SCCs.

(c) We can perform a depth-first search on this cycle starting at an arbitrary vertex v_0 and see if the graph contains a cycle visiting all vertices. If it does, then all vertices are in the same SCC. Otherwise, there are n SCCs, each containing a single vertex.

(d) This time, if we made G' undirected, it would look like a tree with a single cycle. The only possible nontrivial SCC would be the nodes in the cycle, if the edges in the cycle are all oriented in the same direction. Otherwise, there are again n single-vertes SCCs.

Just like before, we can perform a depth-first search, while keeping track of our current path. If we find a directed cycle, all of the vertices in the cycle will be in the same SCC, while the rest of the vertices will be in their own SCCs. In particular, if the length of the directed cycle is c, then there are $n - c + 1$ SCCs (as the c vertices in the cycle form one SCC and the $n - c$ remaining vertices each form their own SCC).

If no directed cycle is found, then there are n SCCs, each containing a single vertex.

3 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 `true, false, false, and 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 for 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 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 $\overline{\beta} \implies \alpha$. In this 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.

(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.

Hint: Pick a sink SCC of G_I. Assign variable values so that all literals in the sink are True. Why are we allowed to do this, and why doesn't it break any implications?

(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 (in fact, you've already done the bulk of the proof!)

Solution:

(a) Suppose there is a SCC containing both x and $\overline{x}$. Notice that the edges of the graph are necessary implications. Thus, if some x and $\overline{x}$ are in the same component, there is a chain of implications which is equivalent to $x \to \overline{x}$ and a different chain which is equivalent to $\overline{x} \to x$, i.e. there is a contradiction in the set of clauses.

(b) Take any sink component, and assign variables so all the literals in this component are True. Because of how we define the graph, there is a corresponding source component which has the negations of all literals in this component. Remove this source/sink component pair, and repeat the process until the graph is empty. Since we set components to true in reverse topological order, there is no implication from a true literal to a false literal. Since no literal and its negation are in the same SCC, we never try to set a variable to be both true and false. So this produces an assignment satisfying all clauses.

(c) Let φ be a formula acting on n literals $x_1, \ldots, x_n$. Construct a graph with $2n$ vertices representing the set of literals and their negations. For each clause $(a \vee b)$ of φ add the edges $\overline{a} \Rightarrow b$ and $\overline{b} \Rightarrow a$. Use the strongly connected components algorithm and for each i, check if there is a SCC containing both x_i and $\overline{x_i}$. If any such component is found, report unsatisfiable. Otherwise, report satisfiable.

(Note: A common mistake is to report unsatisfiable if there is a path from x_i to $\overline{x_i}$ in this graph, even if there is no path from $\overline{x_i}$ to x_i. Even if there is a series of implications which combined give $x_i \to \overline{x_i}$, unless we also know $\overline{x_i} \to x_i$ we could set x_i to False and still possibly satisfy the clauses. For example, consider the 2-SAT formula $(\overline{a} \vee b) \wedge (\overline{a} \vee \overline{b})$. These clauses are equivalent to $a \to b, b \to \overline{a}$, which implies $a \to \overline{a}$, but this 2-SAT formula is still easily satisfiable.)

4 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 $n = |V|$, $m = |E|$.

Solution: Main idea:
We want to build a new graph G' such that we can apply Dijkstra's algorithm on G' to solve the problem.

We'll start by creating a graph G' with vertices (v_1, v_2) where $v_1, v_2 \in V$ each represent a city. Each vertex represents a state where the first car is in city v_1 and the second car is in city v_2. Note that if v_B is the city of Berkeley and v_C is the city of Chicago, we have starting vertex (v_B, v_B) and ending vertex (v_C, v_C).

Note: we can optimize slightly by not creating vertices (v_1, v_1) for $v_1 \neq v_B, v_C$ since the two cars cannot be in the same city at the same time, but this will not affect the correctness or asymptotic runtime of the algorithm.

Additionally, for cities u_1, u_2, v_1, v_2 where $u_1 \neq u_2$ and $v_1 \neq v_2$, if G contains edges (u_1, v_1) and (u_2, v_2), then G' will contain an edge $((u_1, u_2), (v_1, v_2))$ with weight $c(u_1, v_1) + c(u_2, v_2) + 2r$, representing the combined cost of group 1 driving from u_1 to v_1 and group 2 driving from u_2 to v_2, plus the rental fees for both groups.

Since a group can stay in a city without driving, we also need to consider the case where one group stays in a city while the other group drives to a different city, so we add edges $((u_1, u_2), (v_1, u_2))$ with weight $c(u_1, v_1) + 2r$ and $((u_1, u_2), (u_1, v_2))$ with weight $c(u_2, v_2) + 2r$.

Finally, if v_C is the city of Chicago, we add edges $((v_C, u_2), (v_C, v_2))$ with weight $c(u_2, v_2) + r$ and $((u_1, v_C), (v_1, v_C))$ with weight $c(u_1, v_1) + r$ to denote that once a group arrives in Chicago, they will wait in Chicago, return the car, and do not have to pay any more rental or gas fees.

Our algorithm is now just to create G' as described above, and find the shortest path p_{alg} from (v_B, v_B) to (v_C, v_C) in G' using Dijkstra's algorithm.

Correctness:
Assume there is a valid route with a lower cost than the one returned by the algorithm. Since this route is valid, e.g. two cars cannot be in the same city at the same time, it is possible to represent such a path as a sequence of edges in G' and the corresponding path p' must go from (v_B, v_B) to (v_C, v_C). However, by assumption, the cost of p' is less than the cost of p_{alg}. But then p' would be a shorter path in G' than the one produced by Dijkstra's, which is a contradiction.

Runtime:
G' has $O(n^2)$ vertices and $O(n^2m)$ edges. Since $\log(n^2) = O(\log n)$, The runtime of Dijkstra's is thus $O\left(\left(n^2 + n^2m\right)\log n\right) = O\left(n^2m\log n\right)$ with a binary heap or $O(n^2m + n^2\log n)$ with a Fibonacci heap.

5 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 the 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.

Solution: For convenience, let $n = |V|$ and $m = |E|$.

Main Idea: First, find all of the SCCs of the graph in linear time using the SCC-finding algorithm from class. Then, process each SCC in topological order, first calling UPDATE on each edge from other SCCs to the current SCC, then running Bellman-Ford on each SCC to find the shortest path from s to each vertex in the SCC. At this point, we have the shortest path from s to every vertex in the SCC.

Note: We can optimize slightly by noting that nodes in any SCC that comes before the SCC containing s in the topological order cannot be reached from s. Therefore, such nodes cannot be part of the shortest s-t path and so any SCC that comes before the component containing s can be ignored. Likewise, any SCC that comes after the SCC containing t in the topological order cannot reach t and can also be ignored. Note that these optimizations do not affect the worst-case asymptotic runtime of the algorithm, since s could be in the first SCC and t could be in the last SCC.

Proof of Correctness: Let C_s be the SCC containing s. For every $v \in C_s$, the algorithm finds the shortest path from s to v since any such path must lie entirely within C_s. Otherwise, if the shortest s-v path went through some $u \notin C_s$, then u would be reachable from s and v would be reachable from u, contradicting the fact that C_s is a strongly connected component.

For a subsequent SCC C, suppose the shortest paths for every vertex in SCCs that come before C in the topological order are correct (this will be our "inductive hypothesis").

Then, after processing all edges pointing into C, for every $v \in C$, $d(v)$ correctly stores the length of the shortest path from s to v that does not pass through any other vertex in C, as any such path can only pass through SCCs that come before C in the topological order. This must be true because there are no edges from C to any prior SCCs, and any $v \in C$ is only reachable from vertices in C or from prior SCCs.

Finally, after running Bellman-Ford on C, $d(v)$ correctly stores the length of the shortest path from s to v, as any such path can either go from $s \rightsquigarrow v$ passing only through vertices in prior SCCs, or go from $s \rightsquigarrow u \rightsquigarrow v$ where $u \in C$. In the latter case, $s \rightsquigarrow u$ is optimal by the inductive hypothesis, and $u \rightsquigarrow v$ is optimal by correctness of the Bellman-Ford algorithm.

This content is protected and may not be shared, uploaded, or distributed.

Runtime Analysis: The SCC-finding algorithm runs in $O(n+m)$ time. Running Bellman-Ford on each SCC takes at most $n_C \cdot m_C$ calls to UPDATE, where n_C and m_C are the number of vertices and edges in strongly connected component C. Updating the shortest paths to downstream SCCs takes $\text{OutDeg}(C)$ calls to UPDATE where $\text{OutDeg}(C)$ is the number of edges from C to other SCCs. Then the number of calls to UPDATE is at most:

$$\begin{aligned}\sum_{C \in \text{SCCs}} (n_C \cdot m_C + \text{OutDeg}(C)) &\leq \sum_{C \in \text{SCCs}} (n_C \cdot m_C) + \sum_{C \in \text{SCCs}} \text{OutDeg}(C) \\ &\leq \sum_{C \in \text{SCCs}} (n_C \cdot m_C) + m \leq \sum_{C \in \text{SCCs}} (k \cdot m_C) + m \\ &\leq k \cdot \sum_{C \in \text{SCCs}} m_C + m \leq k \cdot m + m\end{aligned}$$

where the second inequality follows from the fact that the number of edges between SCCs is at most m, the third inequality follows from the fact that each SCC has at most k vertices, and the fourth inequality follows from the fact that the number of edges inside all SCCs is at most m.

Since each call to UPDATE takes $O(1)$ time, combined with the runtime of the SCC-finding algorithm, the total runtime of the algorithm is $O(n + k \cdot m)$.

This content is protected and may not be shared, uploaded, or distributed.

6 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_j at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geq 0$ 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 $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. 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.

Hint 1: represent the currencies and rates by a graph whose edge weights are real numbers.

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

(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.*

Solution:

(a) **Main Idea:**
We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from a to b corresponds to finding the path with the largest product of exchange rates. To turn this into a shortest path problem, we weigh the edges with the negative log of each exchange rate, i.e. set $w_{ij} = -\log r_{ij}$. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

Proof of Correctness:
To find the most advantageous ways to converts c_a into c_b, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to

minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. The correctness of the entire algorithm follows from the proof of correctness of Bellman-Ford.

Runtime:
Same as runtime of Bellman-Ford, $O(|V|^3)$ since the graph is complete.

(b) **Main Idea:**
Just iterate the updating procedure once more after $|V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

Proof of Correctness:
Same as the proof for the modification of Bellman-Ford to find negative edges.

Runtime:
Same as Bellman-Ford, $O(|V|^3)$.

Note:
Both questions can be also solved with a variation of Bellman-Ford's algorithm that works for multiplication and maximizing instead of addition and minimizing.

This content is protected and may not be shared, uploaded, or distributed.