Compsci 330 Design and Analysis of Algorithms
Assignment 6, Spring 2024 Duke University

Example Solutions

Due Date: Thursday, March 7, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on gradescope. You can use the internet, but looking up solutions or using large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

Grading. Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[1]If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Team Partition). You are given n players numbered $1, \ldots, n$ and a list of m constraints $C = c_1, \ldots, c_m$ where each constraint is of the form (i, j) and means that players i and j cannot be placed on the same team. Your goal is to partition the players into two teams T_A and T_B such that every player is assigned to one team and no constraints are violated. Teams do not have to be the same size.

For example, suppose there are $n = 4$ players and you have the constraints $(1, 2)$ and $(2, 4)$. Then you could partition the players into teams $T_A = \{1, 4\}$ and $T_B = \{2, 3\}$. However, if you had the additional constraint $(1, 4)$, it becomes impossible to give a valid partition.

Describe a $O(n+m)$ runtime algorithm that either computes and returns a valid partition or returns that none exists. Prove the correctness of the algorithm and analyze its runtime complexity.

You may use DFS, BFS, Dijkstra's algorithm, or the Bellman-Ford algorithm as described in lecture without restating the algorithm or arguing for its correctness. If you modify the algorithm, make sure to be precise about how and explain why any modifications are correct.

Solution 1. Our algorithm proceeds by constructing a graph and running breadth-first search (BFS), using the distances to create the partition.

(i) Let $G = (V, E)$ be an undirected graph where there is a vertex v_i for every player and an edge (v_i, v_j) for every constraint (i, j).

(ii) Run an exhaustive BFS (BFS from an arbitrary vertex and continue until all nodes are explored). Record the shortest path distances $D[\,]$.

(iii) Let $T_A = \{i \mid D[v_i] \text{ is even}\}$ and let $T_B = \{j \mid D[v_i] \text{ is odd}\}$.

(iv) Check if any constraints are violated. If so return that no partition exists. Otherwise, return T_A and T_B.

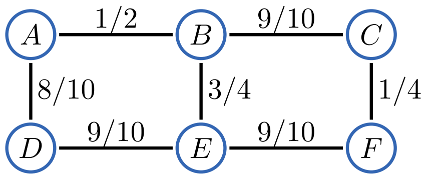
Correctness. Note that we explicitly check constraints, so if the algorithm returns a partition it is certainly correct.

Suppose instead the algorithm returns that no valid partition exists. Then there are two vertices v_i and v_j with the same parity of distance from some BFS start vertex v_1 (both odd or both even) and there is a constraint (i, j). Let v_k be the last node, beginning from v_1 on the shortest path found by the BFS to both v_i and v_j (possibly v_1 itself). Then there is a cycle $v_i \to v_j \to \cdots \to v_k \to \cdots \to v_i$ of odd length, since $v_j \to \cdots \to v_k$ and $v_k \to \cdots \to v_i$ are either both odd or both even length. Any odd length cycle of constraints implies no valid partition exists (note that team assignments need to alternate along the cycle).

Runtime. The runtime of the algorithm is $O(n + m)$, as step (i) takes $O(n + m)$ time to create the graph, (ii) takes $O(n + m)$ time to explore the graph using BFS, (iii) takes $O(n)$ time to partition, and (iv) takes $O(m)$ time to check constraints.

Problem 2 (Probabilistic Routing). You are routing packets of information along a lossy network. You are given a connected, undirected, weighted graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, where the edge weights are real numbers in $(0, 1)$ corresponding to probabilities. The success probability along a path is the product of the probabilities of its edges.

For example, the success probability along the path A, B, C, F in the graph diagrammed below is $(1/2)(9/10)(1/4) \approx 11\%$ whereas the success probability along the path A, D, E, F is $(8/10)(9/10)(9/10) \approx 65\%$.



Describe an algorithm for computing the maximum probability with which a packet can be routed from a source s to a target t. Prove that the algorithm is correct and analyze analyze its running time. Make your algorithm as asymptotically efficient as possible.

You may use DFS, BFS, Dijkstra's algorithm, or the Bellman-Ford algorithm as described in lecture without restating the algorithm or arguing for its correctness. If you modify the algorithm, make sure to be precise about how and explain why any modifications are correct.

Solution 2. One solution is to transform all of the weights in the graph. In particular, let G' be the graph G where we replace each original edge weight p_e with $w_e = -\ln(p_e)$. Note that because $0 < p_e < 1$, we know that $-\ln(p_e)$ is nonnegative. Now, run Dijkstra's algorithm on G' with start vertex s and target vertex t. Suppose Dijskstra's algorithm finds a shortest path $P(s, t)$ in G'. We claim this is a maximum probability path in G.

Correctness. We assume the correctness of Dijsktra's algorithm to compute a shortest path $P(s, t)$ in G'. Then for any other path from s to t $P'(s, t)$

$$
\begin{aligned}
\sum_{e \in P(s,t)} w_e &\leq \sum_{e \in P'(s,t)} w_e \\
\implies \sum_{e \in P(s,t)} -\ln(p_e) &\leq \sum_{e \in P'(s,t)} -\ln(p_e) \\
\implies -\ln\left(\prod_{e \in P(s,t)} p_e\right) &\leq -\ln\left(\prod_{e \in P'(s,t)} p_e\right) \\
\implies \ln\left(\prod_{e \in P(s,t)} p_e\right) &\geq \ln\left(\prod_{e \in P'(s,t)} p_e\right) \\
\prod_{e \in P(s,t)} p_e &\geq \prod_{e \in P'(s,t)} p_e
\end{aligned}
$$

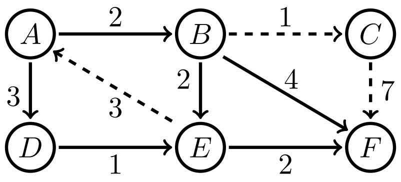
where the last inequality follows because log is a monotone increasing function for nonnegative numbers. So $P(s, t)$ must have maximum probability among all paths from s to t.

Alternatve. It is also possible directly modify Dijkstra's algorithm to maintain maximum probability paths during a search on the original weights. The pseudocode for such a modification is given below.

```
1: procedure ROUTE($G, s, t$)
2:     Initialize $Prob(s) = 1$ and $Prob(u) = 0$ for all $u \neq s$
3:     Initialize priority queue $Q$ all vertices in $V$, using $Prob$ values for keys
4:     while $Q$ not empty do
5:         $u$ =ExtractMax($Q$)
6:         for $(u, v) \in E$ do
7:             if $Prob(v) < Prob(u) \cdot p_{uv}$ then
8:                 $Prob(v) = Prob(u) \cdot p_{uv}$
9:                 $Prev(v) = u$
10:                Update the key of $v$ to $Prob(v)$
11:    return $Prob(t)$
```

Runtime. Either way, the runtime complexity of the algorithm is just $O((n+m)\log(n))$ dominated by the runtime complexity of a single run of Dijkstra's algorithm.

Problem 3 (Useless Edges). Let $G = (V, E)$ be a directed graph, not necessarily acyclic, with $|V| = n$, $|E| = m$ and with positive edge weights $w(u, v)$ for every $u \to v \in E$. For given vertices s and t, note that there may be multiple shortest paths from s to t. For example, there are three distinct shortest paths from A to F in the following figure (edges of these paths drawn as solid lines).



Describe an algorithm that, given vertices s and t, returns the number of edges in the graph that are *not* included in *any* shortest paths from s to t. In the example graph above with $s = A$ and $t = F$ there are three such edges (drawn in dashed lines). Prove that the algorithm is correct and analyze analyze its running time. Make your algorithm as asymptotically efficient as possible.

You may use DFS, BFS, Dijkstra's algorithm, or the Bellman-Ford algorithm as described in lecture without restating the algorithm or arguing for its correctness. If you modify the algorithm, make sure to be precise about how and explain why any modifications are correct. Analyze the

Solution 3. We run single-source Dijsktra's algorithm twice: Once from s in the original graph to calculate the shortest path distances $d[s, u]$ from s to every other node u, and once from t in the reverse graph (with the same weights) to calculate the shortest path distances $d[u, t]$ from every other node u to t. We then count an edge $u \to v$ as useless if $d[v, t] = \infty$ (in which case it is impossible to traverse the edge and still reach t) or if $d[s, u] + w(u, v) + d[v, t] > d[s, t]$ (in which case traversing the edge necessarily results in a longer path than the shortest). We write as a procedure below.

```
1: procedure USELESS($G, s, t$)
2:     Run Dijkstra's algorithm on $G$ starting from $s$ to compute distances $d[s, u]$
3:     Compute the reverse graph $G^R = (V, E^R = \{v \to u \mid u \to v \in E\})$
4:     Run Dijkstra's algorithm on $G^R$ starting from $t$ to compute distances $d[u, t]$
5:     $c = 0$
6:     for $u \to v \in$ do
7:         if $d[v, t] = \infty$ or $d[s, u] + w(u, v) + d[v, t] > d[s, t]$ then
8:             $c = c + 1$
9:     return $c$
```

Correctness. We assume the correctness of Dijkstra's algorithm for computing the shortest path distance $d[s, u]$ from s to u in G for every $u \in V$ and for computing the shortest path distance from t to u in $G^R$for every $u \in V$. Note that the shortest path distance from t to u in G^R is equal to the shortest path distance $d[u, t]$ from u to t in G.

We want to argue that $u \to v$ is "useless" (not included in any shortest path from s to t) if and only if $d[v, t] = \infty$ or $d[s, u] + w(u, v) + d[v, t] > d[s, t]$.

- Suppose $d[v, t] = \infty$. Then there is no path from v to t. Then there is no path that traverses $u \to v$ and then reaches t. The edge $u \to v$ is therefore not on any path from s to t.

- Suppose $d[s,u] + w(u,v) + d[v,t] > d[s,t]$. Consider any path P from s to t that traverses the edge $u \to v$. Let $|P|$ be the sum of the weights of a path consisting of zero of more edges. The path can be decomposed into $P_{s \dashrightarrow u}$ (the path edges from s to u), $u \to v$, and $P_{v \dashrightarrow t}$ (the path edges from v to t). Since $d[s,u]$ is the shortest path distance from s to u, $|P_{s \dashrightarrow u}| \geq d[s,u]$. Similarly, $|P_{v \dashrightarrow t}| \geq d[v,t]$. So we conclude that

  $$\begin{aligned} |P| &= |P_{s \dashrightarrow u}| + w(u,v) + |P_{v \dashrightarrow t}| \\ &\geq d[s,u] + w(u,v) + d[v,t] \\ &> d[s,t]. \end{aligned}$$

  To see the other direction, observe that if $u \to v$ is "useless" then it cannot have $d[s,u] + w(u,v) + d[v,t] \leq d[s,t]$, or else the path composed of the shortest path from s to u, then the edge $u \to v$, then the shortest path from v to t would be a shortest path.

Runtime. There are two runs of Dijkstra's algorithm, each of which takes $O((n+m)\log(n))$ time. Computing the reverse graph can be done in linear $O(n+m)$ time in a single pass over the adjacency list representation. The for loop on line 6 has m iterations and each iteration is constant time. The two runs of Dijkstra's algorithm dominate, so the overall runtime complexity is $O((n+m)\log(n)$.

Problem 4 (Applied). Your are working at an investment firm and are given the task to build an auto-detector for arbitrage ("risk-free" profit) opportunity from the currency exchange market. Given a list of exchange rates, your detector should tell you if you can make profit and a currency exchange "path" you can follow to make profit (if such opportunity exists).

Suppose the current exchange rate is:

$$USD \xrightarrow{130} JPY \qquad JPY \xrightarrow{0.006} GBP \qquad GBP \xrightarrow{1.3} USD$$

Then starting with 1 USD, you can convert it along USD-JPY-GBP-USD and get $130 \times 0.006 \times 1.3 = 1.014$ USD, which is more that what you have initially. We call such path an *arbitrage path*, since we can earn guaranteed profit from the trades.

The exchange rates are represented by (i, j, w_{ij}) tuples, where i, j are integers representing types of currencies and w_{ij} is the exchange rate (1 unit of currency i exchanges to w_{ij} units of currency j). Let V be the set of currencies and E the set of exchange rate tuples. You want to find a path that starts with currency v_i, goes through v_{i+1}, v_{i+2},... v_{i+k} and ends at v_i so that the products of their corresponding rates are greater than 1 ($w_{i,i+1}w_{i+1,i+2}\dots w_{i+k,i} > 1$) for some $k \le |V|$.

You will need to design and implement an algorithm that efficiently determines an arbitrage path, if it exists. Your solution will need to have an empirical runtime that is within constant factors of an reference solution with $O(|V||E|)$ worst case runtime. If there are multiple arbitrage paths, you algorithm only needs to return one and the choice of starting currency doesn't matter. Also note that the existence of (i, j, w_{ij}) doesn't imply the existence of $(j, i, 1/w_{ij})$.

Hint: How could you modify the exchange rates to use an existing algorithm?

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline. **NOTE:** Unlike the theory problems, the applied problem grade **is the raw score shown on Gradescope**. See the course assignments webpage for more details.

- **Python.** You should submit a file called `arbitrage.py` to the Gradescope item "Assignment 3 - Applied (Python)." The file should define (at least) a top level function `arbitrage` that takes in a list of tuples `(i:int, j:int, w:float)` and returns:
  - If an arbitrage path exists: `[ ]` - a list of integers that are ordered to form an arbitrage path
  - Otherwise: `None`

  For example, your algorithm should return `[1,2,3,1]`, if we take USD as currency 1, JPY as currency 2 and GBP as currency 3,

- **Java.** You should submit a file called `Arbitrage.java` to the Gradescope item "Assignment 3 - Applied (Java)." The file should define (at least) a top level function `arbitrage` that takes in a 2D array of currencies and an array of weights where the conversion rate from `currencies[i][0]` to `currencies[i][1]` is `weight[i]`. The method header should be as follows:
  - `public List<Integer> arbitrage(int[][] currencies, double[] weights);`

  As shown the header, the output should be a list of integers, which is the arbitrage path, if there is one. If there is no path, return `null`. For example, your algorithm should return `[1,2,3,1]` if we take USD as currency 1, JPY as currency 2, and GBP as currency 3.

Solution. We do not provide code solutions for applied problems, but we do sketch the idea here. The goal is to detect a particular kind of cycle in the exchange graph: One in which the *product* of weights along the cycle is greater than 1. Unfortunately, computing the *maximum* cycle, under some notion of maximum, is typically computationally intractable.

Instead, we will attempt to reduce the problem to that of *negative cycle detection* which we can solve using the Bellman-Ford algorithm. In particular, if we take the negative logarithm of the weights (see, for example, problem 2), then an arbitrage cycle corresponds to a negative total (summing) weight cycle. We can seach for the existence of such a cycle by running the Bellman-Ford algorithm for an additional iteration than usual to check if it further reduces any distances. If so, we then have to backtrack to get the cycle itself.