COMS21103: Problem set 6

2015/2016

Remark: Most problem sets for the next few weeks will contain at least one starred problem, which is more challenging. If any of the problems seem unclear, please post a question on the Blackboard discussion board.

1. Arbitrage is a (real) method that can be used to make money from fluxations in currency exchange rates. The idea is to convert one unit of a currency into more than one unit of the same currency. For example imagine that 1 U.S. dollar buys 0.7 UK pounds, 1 UK pound buys 1.5 euros and 1 euro buys 1.1 dollars. By converting 1 dollar into pounds then into euros then into back dollars, you would end up with 1.155 dollars; This is a profit of slightly more than 15%!

   Consider a system with n currencies $c_1, c_2 \ldots c_n$ where one unit of currency c_i buys $r_{i,j}$ units of currency c_j. Give an efficient algorithm to determine whether arbitrage exists (for the current exchange rates). What is the running time of your algorithm?

   Hint: Consider the fact that $\log(ab) = \log(a) + \log(b)$.

   Answer (sketch): Build a graph with one vertex v_i for each currency c_i. For each i, j is an edge (v_i, v_j) with weight $-\log r_{i,j}$. Run the Bellman-Ford algorithm (with any source vertex) on the graph you have constructed. This takes $O(|V||E|) = O(n^3)$ time. Arbitrage is possible iff a negative cycle is found.

   The proof is as follows. We first associate each sequence of currency conversions with a path in the natural way. Specifically, converting from c_i to c_j corresponds to following the edge (v_i, v_j). In particular a cycle corresponds to a sequence of conversions starting from some currency c_i and ending up with the same currency c_i. What we need to prove is that the cycle has negative weight in the graph iff the currency exchanges give a profit.

   Consider a cycle of length k given by vertices $v_{i_1}, v_{i_2}, v_{i_3}, \ldots v_{i_k}$ (see slide 282 of the Bellman-Ford lecture). For notational simplicity we have further assumed that vertices in the cycle are the first k. In general they won't be but it simplifies the notation (and doesn't impact correctness). This cycle has weight,

$$\sum_{j=1}^{k} \text{weight}(v_j, v_{j+1}) = \sum_{j=1}^{k} -\log r_{i,i+1} = -\log\left(\prod_{j=1}^{k} r_{i,i+1}\right).$$

   Here as in the lecture $v_{k+1} = v_1$. The final equality is from the hint.

   Let $x = \prod_{j=1}^{k} r_{i,i+1}$ so that the weight of the cycle is $-\log x$. Now observe that if you perform the currency exchanges given by the cycle (starting with one unit of currency c_1). You end up with x units of currency c_1 (notice that p is in general not an integer). You make a profit iff $x > 1$. The proof then follows from the mathematical fact that $x > 1$ iff $\log x < 0$ (Google "log x" for a graph).

You should verify that assuming that the cycle contains the vertices $v_{i_1}, v_{i_2}, v_{i_3}, \dots v_{i_k}$ doesn't affect the proof. In particular notice that the answer isn't affected by relabelling the currencies.

2. Consider the following modification to Bellman-Ford (which is called Yen's improvement). Assume that the vertices are numbered $v_1, v_2, \dots v_{|V|}$ (any ordering is fine). Separate the edges into two sets E_f and E_b. The set E_f contains every edge (v_i, v_j) such that $v_i < v_j$. The set E_b contains every edge (v_i, v_j) such that $v_i > v_j$. Notice that each edge is in exactly one set (assuming we don't have self loops). In each iteration, instead of relaxing the edges in an arbitrary order, we relax them in the following order. First we relax each edge $(v_i, v_j) \in E_f$ in increasing order of i. For two edges (v_i, v_j) and $(v_i, v_{j'})$, the order remains arbitrary. Second we relax each edge $(v_i, v_j) \in E_b$ in decreasing order of i (again breaking ties arbitrarily). Prove that after this modification, Bellman-Ford's algorithm only needs $\lceil |V|/2 \rceil$ iterations instead of $|V|$ iterations. Notice that even though this doesn't change the time complexity it is almost twice as fast!

   Answer (sketch): Consider a shortest path from s to some arbitrary t. This shortest path can be uniquely decomposed into 'sections' as follows. A forward section is a contiguous subsequence of the edges on the path such that every edge is in E_f and both the edges before and after the section (if they exist) are in E_b. A backward section is the same but with the roles of E_b and E_f reversed. As an example the path $(v_2, v_4), (v_4, v_5), (v_5, v_8), (v_8, v_3), (v_3, v_1), (v_1, v_7)$ has three sections. The first (forward) section is $(v_2, v_4), (v_4, v_5), (v_5, v_8)$, the second (backwards) section is $(v_8, v_3), (v_3, v_1)$ and the third (forward) section is (v_1, v_7). We insist that the first section is a forward section to simplify the proof. This may mean that the first section is empty. However, by definition, every other section contains at least one edge.

   Recall that the Bellman-Ford algorithm is guaranteed to have found the shortest path from s to t after each edge in the shortest path has been relaxed in order. Now consider the modified relaxation order. We first relax each edge in $(v_i, v_j) \in E_f$ in increasing order of i. After these relaxations, we have relaxed every edge in the first (forward) section in the order it appears on the path. We next relax each edge in $(v_i, v_j) \in E_b$ in decreasing order of i. After these relaxations, we have relaxed every edge in the second (backward) section in the order it appears on the path. This is the end of the first iteration. By the same argument, after the second iteration, we have relaxed each edge in sections three and four and so on. Recall that the maximum number of edges on a shortest path is $|V| - 1$. As each section (except possibly the first) contains at least one edge, the number of sections (including the empty section) is at most $|V|$. We make two sections worth of progress in each iteration except for the last iteration where there may be only one section left. Therefore the number of required iterations is at most $\lceil |V|/2 \rceil$.

3. Every day you drive across Bristol from your home (h) to your work (w). You can think of Bristol of a directed graph consisting of nodes (junctions) and edges (roads). The weight of edge (u, v) is length of the road between junction u and junction v. You know that you can find the shortest path to work by running Dijkstra's algorithm.

   To allow for road-blocks you decide to allow for one road being closed on each journey. Being organised you decide to compute the shortest path for each possible road closure in advance. This means that on a given day, you can simply look up the shortest path that avoids the closed road. Give an algorithm for this problem that runs in $O(|V||E|\log|V|)$ time. Formally, your algorithm should produce $|E|$ outputs; For each edge $e \in E$, you should output the length of the shortest path between h and w that doesn't use edge e. You don't need to work out how to output the paths themselves (though in practice you would want to).

Answer (sketch): First consider the following algorithm which runs in $O(|E|^2\log|V|)$ time. Consider each possible road-block (i.e. each $e \in E$ individually) and run Dijkstra's algorithm (with h as the source) on the graph with that edge deleted. We perform $|E|$ iterations and each iteration takes $O(|E|\log|V|)$ time (assuming the graph is connected). The iteration with edge e omitted outputs the corresponding shortest path. The key idea is to modify this algorithm so that we only have to perform at most $|V|-1$ iterations, one for each edge on the shortest path from h to w.

As the first step in our improved algorithm we run Dijkstra's algorithm once with h as the source in $O(|E|\log|V|)$ time. Although you did not see this in lectures it is straightforward to modify Dijkstra's algorithm to output the shortest path between h and w (i.e. the actual path, not its length). The basic idea is that when a vertex v is settled we write down the final edge (u, v) on the shortest path from h to v. The set of all of these edges form a tree called the shortest path tree (for more details see CLRS 24.3). The unique path from h to w in the shortest path tree is the shortest path from h to w in the graph.

Now let E' be the set of edges on the shortest path between h and w. If the road-block is an edge which is not on this path, the shortest path length does not change. Therefore we already have the answer for any road-block edge in $E \setminus E'$ (i.e. an edge not on the path). To find the shortest path for a road-block in $e \in E'$ we run Dijkstra's algorithm with e deleted (and h as the source). The final observation is that as E' is a shortest path it contains at most $|V|-1$ edges (we proved this in the Bellman-Ford lecture). Therefore the time complexity is $O(|V||E|\log|V|)$ as required.

As an extension, you could think about how you could generalise this solution to the problem where you want to plan for each possible road-block and all possible destinations. That is w can now be any vertex in V. You should output one shortest path length for each pair (w, e) where $w \in V$ and $e \in E$. Your algorithm should still run in $O(|V||E|\log|V|)$ time.