*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology November 2, 2010
Professors Konstantinos Daskalakis and Patrick Jaillet Handout 5

# Problem Set 5

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

**Part A questions** are due **Tuesday, November 16th** at **11:59PM**.

**Part B questions** are due **Friday, November 19th** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be a valid Python file, together with one PDF file containing your solutions to the two theoretical questions in Part B.

Templates for writing up solutions in LaTeX are available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

---

## Part A: Due Tuesday, November 16th

1. (**15 points**) No Odd Cycles

   Given an undirected graph $G = (V, E)$, design an algorithm that decides whether or not $G$ contains a cycle of odd length. Prove its correctness. Your algorithm should run in time $O(|V| + |E|)$.

   **Solution:** One of the objectives of this problem was to make you realize the following fact from graph theory: a graph does not contain any odd length cycles **if and only if** it is bipartite. (A *bipartite* graph is one in which there is a partition of the vertex set into two subsets so that no edge in the graph is incident to two vertices from the same subset.) Notice how this equivalence turns a negative statement, that the graph does not have odd cycles, into a positive statement, that the graph does have a bipartition. Such equivalences are often very useful for designing algorithms.

   **Lemma 1** *A graph does not contain odd cycles if and only if it is bipartite.*

   *Proof.* First, we observe that if the graph contains an odd cycle, it cannot be bipartite. For the sake of contradiction, suppose $V$ does have a bipartition into two sets $V_1$ and $V_2$. Then, the vertices going along a cycle must alternate between being in $V_1$ and being in $V_2$. If the length of the cycle is odd, this is impossible.

   Now, suppose the graph does not contain any odd cycle. We will show that its vertex set can be partitioned into two sets $V_1$ and $V_2$ so that each edge is incident to one vertex in $V_1$ and

one in $V_2$. To do so, start at an arbitrary vertex $s$ and consider the shortest-path tree[1] rooted at $s$. Put $s$ in $V_1$, the vertices in the next level of the tree in $V_2$, those in the next level in $V_1$, and so on alternating between $V_1$ and $V_2$. We claim that there's no edge connecting two vertices both in $V_1$ or both in $V_2$. For the sake of contradiction, suppose there was such an edge $e$ between two vertices $u$ and $v$. Let $w$ be the least common ancestor of $u$ and $v$ in the tree. Then, the cycle formed by concatenating the path in the tree between $u$ and $w$, the path in the tree between $w$ and $v$, and the edge $e$ forms a cycle of odd length, since the length of the two paths between $u$ and $w$ and between $v$ and $w$ are of the same parity, either both odd or both even. This is a contradiction, and so $V_1$ and $V_2$ must be a bipartition of the connected component that $s$ lies in. We can repeat the same argument for every connected component of the graph to produce a bipartition of the whole graph. □

The algorithm to test bipartiteness is immediate from the above proof. Do a BFS to construct the shortest-path tree for each connected component, and as above, assign each vertex to one of two classes $V_1$ and $V_2$ depending on the parity of the distance from the root. Finally, check whether there is an edge between two vertices that are both in $V_1$ or both in $V_2$. The time complexity is essentially that of BFS, $O(|V| + |E|)$.

Many variations of this algorithm are possible for this problem. For example, one could check for violations to bipartiteness as the tree is being built. Or, one could use DFS instead of BFS, since as pointed out above, we just need some spanning tree, not necessarily the shortest-path one.

**Grading:** Most people did very well on this question and got full points. A few people made the mistake of explicitly checking the length of every cycle detected using DFS. This is going to be extremely inefficient, as there can be an exponential number of (even) cycles in a bipartite graph.

2. (**15 points**) Maximum Bandwidth Path

Suppose we have a telephone network $N$ with $n$ nodes, $m$ links (undirected edges) between them, and a distinguished source node $s$. Each link has an associated *bandwidth* which is a non-negative integer that indicates the maximum number of requests that link can handle. In a path of links, the link in the path with the smallest bandwidth constrains the number of requests that can be passed along the path. Hence, we define the bandwidth of a path $P$ consisting of links $\ell_1, \ldots, \ell_k$ to be $\min(\text{bandwidth}(\ell_1), \ldots, \text{bandwidth}(\ell_k))$. The *maximum bandwidth path* to a node $v$ is the path from $s$ to $v$ with the maximum bandwidth.

Design an algorithm that computes the maximum bandwidth path to every node $v$ in the network, and prove its correctness. Your algorithm should have the same asymptotic running time as Dijkstra's.

**Solution:** The purpose of this problem was to make you see that the greedy approach in Dijkstra's algorithm can apply to other optimization problems. Consider the following algorithm.

[1] In fact, any spanning tree suffices for our purposes here.

MAX-DIJKSTRA$(G, s)$
  $b[s] = \infty$
  **for** each $v \in V - \{s\}$
    $b[v] = 0$
  $S = \emptyset$
  $Q = V$ // Initialize $Q$, a *max*-priority queue
  **while** $Q \neq \emptyset$
    $u =$ EXTRACT-MAX$(Q)$
    $S = S \cup \{u\}$
    **for** each $v \in Adj[u]$
      **if** $b[v] < \min(b[u], w(u, v))$
        $b[v] = \min(b[u], w(u, v))$

The key differences from Dijkstra's algorithm are:

- The field $b[v]$ is initialized to infinity when $v = s$ and 0 otherwise. This is in contrast to Dijkstra's algorithm where it's the opposite.
- $Q$ is a max-priority queue, instead of a min-priority queue.
- When relaxing edge $(u, v)$, the field $b[v]$ is updated to the maximum of the previous $b[v]$ and $\min(b[u], w(u, v))$.

The running time of MAX-DIJKSTRA is the same as that of Dijkstra, because the above modifications do not affect the running time asymptotically.

Let's prove its correctness. Denote the bandwidth of the maximum bandwidth path from $s$ to $v$ by $\beta(v)$. The proof is quite similar to that of Dijkstra.

First, observe that for all $v \neq s$ and at all times, we have $b[v] \leq \beta(v)$. This is true by induction. $b[v]$ is initialized to satisfy this condition. For the inductive step, if $b[v]$ is changed while relaxing edge $(u, v)$, then $b[v] = \min(b[u], w(u, v)) \leq \min(\beta(u), w(u, v)) \leq \beta(v)$.

Now, we will show that for every vertex $u$, when $u$ is extracted from $Q$, $b[u] = \beta(u)$. This is immediate for $u = s$ which is the first vertex extracted from the queue. Thus, for the sake of contradiction, suppose that $u \neq s$ is the first vertex for which $b[u] < \beta(u)$ at the time of extraction. Consider the maximum bandwidth path $P$ from $s$ to $u$; there must exist at least one path from $s$ to $u$ because otherwise, $b[u]$ would stay 0 all the time which is indeed correct. Let $y$ be the first vertex along $P$ which is not in $S$ (at the time $u$ is extracted), and let $x$ be $y$'s predecessor along $P$. Since $x \in S$, we have by assumption that $b[x] = \beta(x)$, but then $b[y] = \beta(y)$ since the edge $(x, y)$ which lies on $P$ must have been relaxed. Now, observe that $b[y] = \beta(y) \geq \beta(u) \geq b[u]$, where the first inequality is from the definition of bandwidth and the second from the paragraph above. Because $b[u] \geq b[y]$ for $u$ to have been extracted from $Q$, it follows that $\beta(u) = b[u]$.

**Grading**: Most people got the general idea of the algorithm, though many had details incorrect. 4 points were taken off for error in initialization, 6 points if the relaxation was incorrect.

Some people did not provide the proof, even though one was requested. For this, the number of points that were taken off depended on how much of the proof was sketched.

3. (**20 points**) Shortest Roundtrip Path

   Suppose $G$ is a strongly connected directed graph, meaning that for every two vertices $u$ and $v$ in $G$, there is a directed path from $u$ to $v$ and a directed path from $v$ to $u$. Each edge has a weight, not necessarily non-negative. Let $s$ be a distinguished node in $G$. For every vertex $v$ in the graph, we want to calculate a path from $s$ to $v$ and a path from $v$ to $s$ such that the sum of the weights of edges on the two paths is minimized. Give an efficient algorithm to perform this task, and show correctness. If your algorithm encounters a negative-weight cycle, it should output `false`. Otherwise, it should output the pair of paths from $s$ to $v$ and from $v$ to $s$ that minimizes the roundtrip cost.

   **Solution:**

   The length of the shortest roundtrip path between $s$ and $v$ is the sum of the lengths of the shortest path from $s$ to $v$ and the shortest path from $v$ to $s$. This is seen by an easy cut-and-paste argument since the two paths do not constrain each other.

   In the presence of negative-weight edges, we can find the shortest path from $s$ to $v$ for all $v$ by running the Bellman-Ford algorithm with source $s$. To find the shortest path from $v$ to $s$ for all $v$, we can take the conjugate of the original graph (i.e., the original graph with all edge orientations reversed) and run the Bellman-Ford algorithm on that with source $s$. If there is a negative weight cycle, then it will already be detected in the first run of Bellman-Ford.

   The runtime is $O(|V||E|)$, that of Bellman-Ford.

   **Grading**: Instead of running Bellman-Ford on the conjugate graph with source $s$, several people ran it on the original graph with source $v$ for each vertex $v$ in the graph, leading to an $O(|V|^2|E|)$ time algorithm. 10 points were taken off since the main purpose of this problem was to make you come up with a more efficient algorithm.