CS 573: Graduate Algorithms, Fall 2011
HW 3 (due in class on Tuesday, October 18th)

This homework contains five problems. **Read the instructions for submitting homework on the course webpage**. In particular, *make sure* that you write the solutions for the problems on separate sheets of paper. Write your name and netid on each sheet.

Collaboration Policy: For this home work students can work in groups of up to three students each. Only one copy of the homework is to be submitted for each group. Make sure to list all the names/netids clearly on each page.

Note on Proofs: Details are important in proofs but so is conciseness. Striking a good balance between them is a skill that is very useful to develop, especially at the graduate level.

1. (20 pts) Given a directed graph $G = (V, E)$ and two nodes s, t, an *s-t walk* is a sequence of nodes $s = v_0, v_1, \ldots, v_k = t$ where (v_i, v_{i+1}) is an edge of G for $0 \leq i < k$. Note that a node may be visited multiple times in a walk — this is how it differs from a path. Given G, s, t and an integer $k \leq n$, design a linear time algorithm to check if there is an *s-t* walk in G that visits at least k *distinct* nodes including s and t. *Hint:* You need to use a linear time algorithm to find all strong connected components of a directed graph. Moreover you need to understand the DAG representation of the strong connected components of a graph. You can assume that you have an algorithm for giving you such a representation. Read Chandra's CS 473 lecture notes if you are unfamiliar with this.

2. (20 pts) You are given a directed graph $G = (V, E)$ where each edge e has a length/cost c_e (which may be negative) and you want to find shortest path distances from a given node s to all the nodes in V. The Bellman-Ford algorithm takes $O(nm)$ time where $n = |V|$ and $m = |E|$ while Dijkstra's algorithm can be implemented in $O(m + n \log n)$ time when the edge lengths are non-negative. Suppse G has only k edges with negative lengths where k is small. Show how you can take advantage of this to obtain an algorithm that runs in time $O(k(m + n \log n))$; in other words the time to run Dijkstra's algorithm $O(k)$ times. Your algorithm should output the following: *either* that there is a negative length cycle in G or correct shortest path distances from s to each node v. *Hint:* First solve the problem when $k = 1$ and then generalize.

3. (20 pts) This problem will lead you to an algorithm for the all-pairs-shortest path problem (APSP) in undirected unweighted graphs via matrix multiplication. The algorithm is quite clever to come up with but its analysis can be understood via a sequence of simple claims. Let $G = (V, E)$ be an undirected graph; its *square*, denoted by G^2, is the graph on the same vertex set V and there is an edge uv in G^2 if uv is an edge in G or if there is 2-hop path $u - w - v$ in G.

   - Given a graph G on n nodes show how to compute G^2 from G in time $O(M(n) + n^2)$ where $M(n)$ is the time to multiply two $n \times n$ matrices.
   - If G is connected show that repeatedly squaring a graph $\lceil \log n \rceil$ times results in a complete graph.

For notational simplicity let H be the square of G. We wish to understand how to obtain information for distances in G from distances in H. Prove the following.

- Let $u, v \in V$. Suppose $d_G(u,v)$ is an even number then $d_G(u,v) = 2d_H(u,v)$ and if $d_G(u,v)$ is an odd number then $d_G(u,v) = 2d_H(u,v) - 1$.

The above property shows that if we can recursively compute the distances in H then we would be able to obtain the distances in G approximately to within an additive error of 1. In particular if we knew the parity of the distances in G then we would be able to recover the distances in G from those in H. How do we obtain the parity? We will see that the following facts are helpful. You should prove them.

- Let u, v be distinct nodes in G. Then for *every* neighbor w of v in G we have $d_G(u,v) - 1 \leq d_G(u,w) \leq d_G(u,v) + 1$. Moreover there is at least one neighbor w of v such that $d_G(u,w) = d_G(u,v) - 1$.
- Let u, v be distinct nodes in G. If $d_G(u,v)$ is even then for *every* neighbor w of v in G we have $d_H(u,w) \geq d_H(u,v)$. And if $d_G(u,v)$ is odd then for *every* neighbor w of v in G, $d_H(u,w) \leq d_H(u,v)$ *and* there is some neighbor w of v in G for which $d_H(u,w) < d_H(u,v)$.

As a consequence of the previous two facts prove the following.

- Let u, v be distinct nodes in G. Then $d_G(u,v)$ is even if and only if

$$\sum_{w \text{ neighbor of } v} d_H(u,w) \geq d_H(u,v) \cdot \deg_G(v),$$

  where $\deg_G(v)$ is the degree of v in G.

With the above in place, show the following.

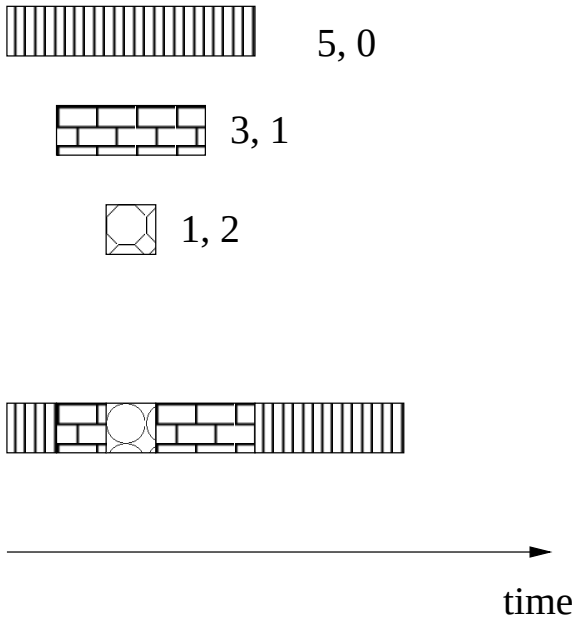
- Given all pairs shortest path distances in H in a matrix A and the adjancency matrix of G in B reduce the problem of finding the parity of $d_G(u,v)$ for each pair u, v to matrix multiplication.

Put together all the above observations to describe a recursive algorithm for APSP in an undirected unweighted graph G on n nodes in $O(M(n) \log n)$ time. Here you will be assuming that $M(n)$ is $\Omega(n^2)$.

4. (20 pts) You are given a collection of n jobs that need to be scheduled on a single machine. Each job j has two attributes: an integer processing time $p_j > 0$ and an integer release time $r_j \geq 0$ when it becomes available. A job that is released at r_j cannot be scheduled before r_j (naturally). The goal is to schedule the jobs to minimize the total completion time of the jobs; in other words minimize $\sum_{j=1}^{n} C_j$ where C_j is the completion time of job j. At any time, only one job can be processed on the machine, however the schedule is allowed to be *pre-emptive*; that is, a job that is currently being processed can be set aside to process a new job that becomes available and so on. A job j is completed when it receives a total of p_j units of processing on the machine. Consider the following simple algorithm. At any time t, schedule the job that has the least amount of time still left to process. See example below for

an illustration of a schedule created by the algorithm. The numbers indicate the processing time and release time of the job.

- Is this an optimal algorithm? If so, prove its correctness.
- Give a polynomial time algorithm that outputs a schedule for the jobs. The schedule should be specify for each job the time intervals (the start and end points) during which the job is processed on the machine. Note that an algorithm that runs in $O(\sum_i p_i)$ time is *not* a polynomial time algorithm.



5. (20 pts) Let $T = (V, E)$ be a tree and let $S \subseteq V$ be a set of special vertices called terminals. For any given subset of edges $X \subseteq E$ let $h(X)$ be the number of connected components in the forest $T \setminus X$ that contain at least one terminal from S. Define a tuple $(E, \mathcal{I})$ where $\mathcal{I} \subseteq 2^E$ as follows.

$$\mathcal{I} = \{X \subseteq E \mid h(X) = |X| + 1\}$$

- Prove that $\mathcal{M}_{T,S} = (E, \mathcal{I})$ is a matroid. A *loop* in a matroid is an element of a ground set which is a circuit. For a given tree $T = (V, E)$ and $S \subseteq V$ when is an edge $e \in E$ a loop in the matroid $\mathcal{M}_{T,S}$?
- For any matroid $\mathcal{M} = (N, \mathcal{I})$ and integer k define $\mathcal{M}_k = (N, \mathcal{I}')$ where $\mathcal{I}' = \{I \in \mathcal{I} \mid |I| \leq k\}$. Show that $\mathcal{M}_k$ is also a matroid.
- From the above two properties derive a greedy algorithm for the following problem. Given a tree $T = (V, E)$ with non-negative edges costs $c : E \to \mathbb{R}_+$, a set of terminals $S \subseteq V$ and an integer $k \leq |S|$ find the smallest cost set of edges in T whose removal results in k components each of which contains at least one terminal. (This is called the Steiner k-cut problem and is NP-Hard in general graphs but can be solved in trees. The algorithm on trees leads to a 2-approximation for general graphs via Gomory-Hu tree representation of a graph.)