Introduction to Algorithms December 15, 2002
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 35

Problem Set 9 Solutions

Problem 9-1. NP **Problems**

(a) A witness w for an instance $x = \langle G, u, v, k\rangle$ is a simple path of length at least k from u to v, i.e. a list of vertices $w_0, w_1, \ldots, w_\ell$. The verification algorithm checks that all w_i are distinct, $\ell \geq k$, $w_0 = u$, $w_\ell = v$, and that $(w_{i-1}, w_i) \in E$ for all $i = 1, \ldots, \ell$. It accepts if and only if all these conditions are met.

If $x \in D_1$, then such a witness exists because it is given by the path, which exists by assumption. Conversely, if the verification algorithm accepts, then the witness gives a simple path of length at least k, so $x \in D_1$. The size of the witness is clearly polynomial in the size of x, and the verification algorithm runs in polynomial time, so $D_1 \in NP$.

(b) A witness w for an instance G is an assignment χ of three colors $\{1, 2, 3\}$ to the vertices. The verification algorithm checks that only the colors $\{1, 2, 3\}$ are used, and that $\chi(u) \neq \chi(v)$ for every $(u, v) \in E$.

If $G \in D_2$, then such a coloring exists by assumption. Conversely, if the verification algorithm accepts, then χ is a coloring such that no two neighboring vertices have the same color, therefore $G \in D_2$. The witness and verification time are polynomial in the size of G, so $D_2 \in NP$.

Problem 9-2. Largest Common Subgraphs

The task is to prove that LARGESTCOMMONSUBGRAPH is NP-complete (this was left out of the problem statement!). First, LARGESTCOMMONSUBGRAPH $\in NP$: a witness is a correspondence between vertices in G_1, G_2 and a list of at least k edges that they have in common. The verification algorithm checks that the correspondence is a bijection and that the edges belong to both graphs.

Next, we show how to reduce CLIQUE to LARGESTCOMMONSUBGRAPH: on an instance $x = \langle G, c\rangle$ of CLIQUE ("is there a clique of size at least c in G?"), our reduction f outputs the triple $f(x) = \langle G, K_c, \binom{c}{2}\rangle$. Here K_c is the complete graph (a clique) on c nodes, and $\binom{c}{2} = c(c-1)/2$ (the number of edges in K_c). Clearly, computing this reduction only requires polynomial-time.

We now show that $x \in$ CLIQUE $\iff$ $f(x) \in$ LARGESTCOMMONSUBGRAPH. First suppose $x = \langle G, c\rangle \in$ CLIQUE: then G has a clique of size c, so K_c is a subgraph of itself and G. This subgraph has $\binom{c}{2}$ edges, so $f(x) \in$ LARGESTCOMMONSUBGRAPH. Conversely, suppose $f(x) \in$ LARGESTCOMMONSUBGRAPH: then G and K_c have a common subgraph of at least $\binom{c}{2}$ edges. But K_c only has that many edges, so K_c in its entirety must be a subgraph of G. Therefore $x = \langle G, c\rangle \in$ CLIQUE. This completes the reduction. Because CLIQUE is NP-complete, so is LARGESTCOMMONSUBGRAPH.

Problem 9-3. Approximate TSP

Our solutions is similar to the 2-approximation in CLRS, but it uses one extra trick to reduce the approximation factor. You should familiarize yourself with that algorithm before reading on.

Our algorithm is as follows: first, find an MST of the graph. This gives us some edge set that forms a tree. Next, find a minimum perfect matching on those nodes which have *odd degree* in the tree (for example, any leaf in the MST has odd degree in the tree). Add the edges of that matching to the MST, so that now every vertex has even degree with respect to the chosen edge set. This is an Eulerian graph, which has an easy-to-find Eulerian tour (a tour which traverses every edge exactly once). We turn this Eulerian tour into a valid traveling saleman tour by simply "shortcutting" over nodes that have already been visited, as in CLRS. By inspection, this algorithm is efficient.

We now analyze the approximation factor of the algorithm: let OPT be the length of an optimum traveling salesman tour. Then the weight of the MST is $\leq OPT$, as argued in CLRS. Let MAT be the cost of a minimum-weight matching on the odd-degree vertices in the MST. Also, let ODD be the length of an optimum traveling salesman tour on the odd-degree vertices of the MST. We can shortcut a tour on all the vertices to get a tour of only the odd-degree vertices, so $ODD \leq OPT$. Now note that in a tour of the odd-degree vertices, we can partition the edges into "odd" and "even," so that the odd edges form a perfect matching on the odd-degree vertices, and so do the even edges. Therefore $2 \cdot MAT \leq ODD \leq OPT$, so $MAT \leq OPT/2$. Now note that the Eulerian tour on the selected edges uses each edge exactly once, so its cost is at most $3OPT/2$. Shortcutting the tour to make it a valid traveling salesman tour can only make it shorter, and this completes the proof.

Problem 9-4. Decision to Search

(a) The intent of the question was to find a clique of size $\geq k$ (not $\leq k$, which is trivial). Suppose there is a polynomial-time algorithm C for deciding whether $\langle G, k\rangle \in$ CLIQUE. We can use this algorithm to find in G a clique of size $\geq k$, if one exists.

The algorithm FIND-CLIQUE(G, k) works in the following way: first, see if C accepts $\langle G, k\rangle$. If not, return "NONE." If so, and G has only k vertices, return G. If C accepts and G has more than k vertices, do the following: pick an arbitrary vertex v of G, and remove it and all its incident edges. Call the resulting graph G'. Then run C on $\langle G', k\rangle$: if C rejects, restore v and its incident edges and remove a different vertex, calling it G' again. Repeat until C accepts some $\langle G', k\rangle$: this must happen, because some vertex v of G is not part of a k-clique in G, so removing it means that G' still has a k-clique. Once C accepts, recursively call FIND-CLIQUE(G', k) and return its output.

First, we argue correctness: clearly the base case of FIND-CLIQUE is correct. Also, we have already argued that C will accept some $\langle G', k\rangle$. Furthermore, a clique of size k in G' is also a clique in G, so by induction on the number of nodes, correctness follows.

Next, this algorithm is polynomial-time: we make at most $|V|$ calls to C in the body of FIND-CLIQUE. Each recursive call to FIND-CLIQUE decreases the number of nodes

by 1, so the number of calls is at most $|V|$. Therefore C is called at most $|V|^2$ times. Removing and restoring vertices are polynomial-time modifications, so the whole algorithm is polynomial-time, as desired.

(b) Because CLIQUE is NP-complete, if CLIQUE $\in P$ then TSP $\in P$ (because TSP $\in NP$). That means there is a polynomial-time procedure T which, on input $\langle G, \ell \rangle$, tells whether there is a traveling salesman tour in G of cost $\leq \ell$. First, we find the optimum tour cost c by binary searching between 0 and the sum of all the edge weights, using T to test each candidate optimum.

Next, we search for an optimum tour in the following way: first, if the graph is a cycle, return its edges as the optimum tour. Otherwise, remove an edge (u, v) (alternatively, increase its cost to ∞) and use T to test if the optimum cost c is maintained. If not, restore the edge and remove different ones, until the optimum is maintained as c (this must happen for some edge, because an optimum tour cannot use all edges). In this way we can keep removing edges that we know are not in a particular optimum tour, until only those edges in the tour remain. Such a procedure is clearly polynomial-time.

Problem 9-5. Randomized Reductions

The algorithm for A is as follows: on input x, choose a random y of appropriate length and compute $f(x, y)$. Then run the algorithm for B on $f(x, y)$. Repeat this 100 times, each time with a new, independent random value y. If ever the algorithm for B rejects, then reject x. If it accepts every time, then accept.

First, it is clear that this procedure is polynomial-time: its running time is 100 times the time for computing f, times the time for deciding B. All of these factors are polynomial, so the product is too.

Now we analyze the error of this algorithm: if $x \in A$, then $f(x, y) \in B$ for every y, so the algorithm will *always* accept (zero error). If $x \notin A$, then the algorithm accepts (i.e., makes an error) only if $f(x, y) \in B$ for all 100 random choices of y. For each y this probablility is at most $1/2$, and each y is independent. Therefore the error probability is at most 2^{-100}, and the algorithm is correct with probability at least $1 - 2^{-100}$.