Problem Set 5 Solutions

This problem set is due before lecture on Tuesday, November 9, 2010.

Exercise 5-1. Do Exercise 29.2-3 in CLRS.

Exercise 5-2. Do Exercise 34.2-1 in CLRS.

Exercise 5-3. Do Exercise 34.5-7 in CLRS.

Exercise 5-4. Do Exercise 35.3-2 in CLRS.

Problem 5-1. A Scenic Hike

You are planning a hike in a wooded area where there are various hiking trails that connect clearings. This can be modeled as a graph \((V, E)\); each clearing is a vertex in the graph and each trail an edge. Veteran hiker that you are, you don’t want to backtrack too much, so you decide to pick a direction for each trail (that is, you may assume the graph is directed). You start at a clearing \(s\) and want to hike to a clearing \(t\).

Additionally, there are \(k\) various scenic views nearby—a particular mountain, or river, and so forth—but the forest is dense enough that you can only (possibly) see them from the clearings, not the trails. For each piece of scenery \(i\), there is a set of clearings \(S_i \subseteq V\) from which the scene is visible.

Define the Scenic View Problem as the following: Given a set of trails and clearings, and a set of scenes (along with the clearings from which they are visible), determine whether or not there exists a hiking path where you can see every scene from some point on the path. Mathematically, given a directed graph \((V, E)\), subsets \(S_1, \ldots, S_k \subseteq V\), and nodes \(s, t \in V\), determine whether or not there exists a path from \(s\) to \(t\) containing at least one node from each of the sets \(S_i\).

Prove that the Scenic View Problem is NP-complete. Hint: Reduce from SAT.

Solution: Given an instance of SAT with \(n\) variables and \(m\) CNF clauses, we reduce it to an instance of the Scenic View Problem with \(2n + 2\) vertices (clearings) and \(m\) subsets (scenes).

For each variable \(x_i\), add a vertex for that variable and its negation. Additionally, add vertices \(s\) and \(t\). Create directed edges from \(s\) to \(x_1\) and \(!x_1\), from \(x_i\) and \(!x_i\) to \(x_{i+1}\) and \(!x_{i+1}\), and from \(x_n\) and \(!x_n\) to \(t\). (See Fig. 1.) Clearly, any path directly corresponds to an assignment of truth values to the variables, and all assignments are possible paths.
Map each CNF clause to a scene that is visible only from the terms in that clause. Thus, in order to satisfy the clause, at least one of the variables has to have the correct assignment, meaning the path will pass through that clearing.

Thus, if we can determine whether a path exists that enables a view of all scenes, we know that corresponds to a variable assignment that satisfies all clauses.

This mapping only takes polynomial time (actually, linear time in \(n\) and \(m\)), so the Scenic View Problem is NP-hard. Similarly, given a witness (that is, a path that claims to satisfy the property), we can easily check whether it contains vertices from each of the sets \(S_i\) in polynomial time, by simply following the path and checking off each set \(S_i\) we hit, then seeing if any \(S_i\) remain. Thus, the Scenic View Problem is in NP, and is therefore NP-complete.

**Problem 5-2. Diet and Nutrition**

A cafeteria offers \(n\) foods, labeled \(f_1, \ldots, f_n\), costing $1 each. Posted in the cafeteria is a sign listing \(m\) important vitamins, with the foods that supply each. For example, it might state that foods \(f_1, f_2,\) and \(f_3\) supply vitamin \(v_1\), whereas \(f_3, f_4,\) and \(f_5\) supply vitamin \(v_2\).

(a) You’d like to eat healthily, but have a limited budget. You decide you can only afford to spend $\(k\). Thus, you would like to determine whether there is a set of at most \(k\) foods that, between them, supply all \(m\) vitamins. Prove that this problem is NP-complete.

**Solution:** This is just Set Cover: You have a number of elements \(v_i\), and some foods that each contain a set of vitamins. Finding if there is a set of at most \(k\) foods that contain all the vitamins is the same as asking if there are \(k\) sets that cover all \(m\) elements.

(b) The problem from part (a) is a decision problem; it only gives a yes-or-no answer. We can also consider the related search problem: Given a set of vitamins, what is the minimum number of foods necessary to supply all \(m\) vitamins? Assume we have access to a “black box” that solves the search version of the nutrition problem in polynomial time. How could we solve the decision version of the problem in polynomial time?
Solution: Run the black box on our input (not including the bound \( k \)) to search for the minimum answer. Check if the value it outputs is \( \leq k \). Running time is the time of the oracle plus \( \Theta(1) \).

(c) Assume we have access to a “black box” that solves the decision version of the nutrition problem in polynomial time. How could we solve the search version in polynomial time?

Solution: Perform binary search for the value of \( k \), using the black box to check if each value of \( k \) is feasible or not. Running time is the running time of the oracle times \( \Theta(\log n) \).

(d) Now assume that you are allowed to buy fractional portions of food at linearly-corresponding prices; thus, for example, you could buy one-third of a portion of \( f_1 \) at one-third of the cost. You decide you want to buy at least one portion’s worth of food supplying each vitamin (even if this consists of fractional portions of several foods), at minimal cost.

Formulate the search version of this new problem as a linear program. Remember to give the objective function and all linear constraints, and explain where the parts of your linear program come from. Assuming \( P \neq NP \), is this new problem NP-complete? Why or why not?

Solution: For each \( i \leq n \), let \( x_i \) represent the fraction of \( f_i \) we decide to take. We want to minimize the total amount of money spent, which is \( \sum x_i \). For each vitamin \( v_j \), we have a constraint that states that the sum of the \( x_i \) values for the foods \( f_i \) associated with that vitamin must be \( \geq 1 \); this corresponds to fulfilling the nutritional requirements. Furthermore, we have constraints stating that each \( x_i \geq 0 \), since we can’t take negative parts of a portion.

Assuming \( P \neq NP \), this is not \( NP \)-complete, since it is expressable as a linear program and can therefore be solved in polynomial time.

Problem 5-3. Linear Programming

(a) You are a contractor working with large earth-moving equipment in order to make a roadbed ready for paving. There are \( n \) measuring points along the roadbed, one every 100 feet. You measure the altitude in feet above sea level at each measure point, obtaining measurements \( x_1, x_2, \ldots, x_n \). Your goal is to ‘smooth’ the roadbed so it can be paved. After you’ve smoothed out the roadbed, the new altitudes will be \( y_1, y_2, \ldots, y_n \). The new roadbed will be ‘smooth enough’ if \( |y_{i+1} - y_i| \leq 1 \) for \( i \) in \( 1, \ldots, n - 1 \).
The total work it takes to smooth the bed is \( \sum_i |y_i - x_i| \), the sum of the amounts by which each measuring point is raised or lowered. Show how to use Linear Programming to compute, given the \( x \) values, \( y \) values such that the road is smooth enough and work is minimized. *Hint:* It might help to introduce new variables.

**Solution:** Introduce variables corresponding to amount raised and lowered (\( c_i \) and \( d_i \), say), and reduce to LP.

Specifically, you have the following sets of equations for all \( i \):

- \( y_i = x_i + c_i - d_i \) (defining \( c_i \) and \( d_i \) as the increase and decrease respectively)
- \( c_i \geq 0 \) (for decreases, use \( d_i \))
- \( d_i \geq 0 \) (for increases, use \( c_i \))
- \( y_{i+1} - y_i \leq 1 \) (making the roadbed smooth enough)
- \( y_{i+1} - y_i \geq -1 \) (making the roadbed smooth enough)

We want to minimize \( \sum_i (c_i + d_i) \).

(Note that the \( x_i \) values are given constants. It is also acceptable, though not required, to substitute for the \( y_i \) values, leaving only \( c_i \) and \( d_i \).)

**(b)** Explain how to use Linear Programming to efficiently solve a ‘leaky max-flow’ problem - this is a problem that is just like a max-flow problem except that each edge \( e \) is now ‘leaky’. The flow \( p(e) \) into the edge must be less than or equal to the capacity \( c(e) \), but the flow out the other end of that edge is only \( \lambda_e p(e) \), where \( \lambda_e \) is a known constant for each edge and \( 0 \leq \lambda_e \leq 1 \). We wish to maximize the flow entering the sink vertex.

**Solution:** Have one variable for the flow going into each edge. Maximize flow going into \( t \), subject to constraints that the flow into any node other than \( s \) and \( t \) (which, recall, has been reduced by a factor of \( \lambda_e \)) equals the flow out. Additional sets of constraints comes from the capacities on each edge, and the fact that flows are non-negative.

Maximize \( \sum_{e=(x_i,t)} \lambda_e p(e) \) (flow into \( t \))

subject to \( \forall v \neq s, t, \sum_{e=(x_i,v)} \lambda_e p(e) = \sum_{e=(v,x_i)} p(e) \) (flow in = flow out if not \( s, t \))

and \( \forall e, p(e) \leq c(e) \) (capacity constraints)

and \( \forall e, p(e) \geq 0 \) (flows are nonnegative)

**Problem 5-4. Open-Addressed Hash Table**

Suppose we have a hash table of size \( m \), and we wish to store \( n \) items with keys \( k_1, k_2, \ldots, k_n \). The hash table does *not* use chaining to resolve collisions—rather it uses “open addressing”. (See Section 11.4 of CLRS.) In this case there is a two-argument hash function \( h(k, i) \) such that when
inserting an item with key $k$ we look in slots $h(k, 0), h(k, 1), h(k, 2), \ldots$ until we find a slot that is empty and can be used to store the item.

The arrangement of items in the hash table thus depends on the order in which the items are inserted, and there are multiple valid arrangements of items in the hash table.

Give an efficient algorithm that, given the $n$ items and the two-argument hash function $h$, computes the minimum integer $t$ such that there is an arrangement of items in the hash table such that each of the $n$ items can be retrieved by looking at at most $t$ slots. Note that a search in an open-addressed hash table terminates when it hits an empty slot; thus, we want you to (for a given $h$), find a $t$ such that for all $i$:

- The item with key $k_i$ is at location $h(k_i, j)$ for some $j < t$, and
- Location $h(k_i, j')$ is nonempty for all $j' < j$.

You do NOT need to show that this arrangement of keys came from a valid sequence of insertions.

**Hint:** Guess a value for $t$; check it by phrasing the question as a (possibly weighted) bipartite matching problem.

**Solution:** For a given value of $t$, we can set up a bipartite matching problem as follows: Set up $n$ nodes on the left corresponding to $k_1 \ldots k_n$, and nodes on the right corresponding to the first $t$ hash values of each of the keys (that is, $h(k_1, 0), h(k_1, 1), \ldots h(k_1, t - 1), h(k_2, 0), \ldots h(k_n, t - 1)$). Note that there may be fewer than $tn$ nodes on the right since some of the values may collide. Put edges between $k_i$ and $h(k_i, j)$ for $i = 1 \ldots n$, $j = 0 \ldots t - 1$.

If there is a matching of cardinality $n$, then every key can fit into one of its first $t$ values in the hashtable.

We can solve the bipartite matching problem in $O(|V||E|)$ time using the Ford-Fulkerson method. Here $|V|$ and $|E|$ are both $O(tn)$, so we can solve an instance for a given $t$ in time $O(t^2n^2)$.

We can binary-search for the minimum value of $t$ that works by working up from 1 (doubling each time) until we find the first value of $t$ that works, then performing a standard binary search between the final $t/2$ and $t$. This takes $O(\log t_{\text{min}})$ iterations. The total running time is $O(t^2n^2 \log t_{\text{min}})$ since we end up performing $O(\log t_{\text{min}})$ iterations between $t/2$ and $t$ to find $t_{\text{min}}$.

This doesn’t guarantee that the search algorithm will be capable of finding each element, since there could be empty entries in the hashtable on a search path. However, we can resolve this issue in any of several methods:

- We could fill the empty slots on search paths with the special symbol DELETED, as described in CLRS sec. 11.4.
- We could copy each key into the empty slots in its search path. This only decreases the number of slots looked at when searching for each element.
• We could move each key “forward” into the first empty slot on its search path. This only decreases the number of slots looked at for the first element, but it might introduce a new empty slot on some other key’s search path. However, we can then iteratively move that key into the newly-vacated slot. After $O(nt)$ iterations of this process, we will have removed all of the empty slots.

• We can use edge weights in the original graph in order to ensure that we do not get any empty slots. If we set the weight of the edges corresponding to the first slot (for each element) as the highest weight, the edges corresponding to the second slot (for each element) as the next-highest weight, and so forth. A maximum-weight matching then will not have any empty slots on search paths, since if there were, the corresponding edge would have a higher weight than the one selected, yet would be unused. Since there exist algorithms for weighted bipartite matching that operate in $O(|V||E| + |V|^2 \log |V|)$ time, and we again run this $O(\log t_{\min})$ times, the total running time is $O(n^2t_{\min}^2 \log(nt_{\min}) \log t_{\min})$.

There is one detail, however: we need to ensure that a matching of maximum weight matches every element to a slot. (That is, there is a danger that a matching exists where some element goes unmatched, but the other elements get assigned to earlier slots, so the total weight is higher.) We can set the weights appropriately to prevent this; namely, we need to ensure that a matching in which $n$ elements are matched to the $t$-th location is more desirable than a matching in which $n - 1$ elements are matched to the first location. If the weights for an individual element range from $w$ (for the $t$-th slot) to $w + (t - 1)$ (for the first slot), then we can ensure this by setting $w$ such that

$$nw > (n - 1)(w + (t - 1))$$

$$w > (n - 1)(t - 1)$$