15-451 Algorithms, Fall 2010

Homework # 5 **due: Tuesday-Friday, November 9-12, 2010**

Ground rules:

- This is an oral presentation assignment. You should work in groups of **exactly** three. At some point before **Monday, November 8 at 11:59pm** your group should sign up for a 1-hour time slot on the signup sheet on the course web page. By signing up with fewer than three group members you are giving course staff permission to place unpaired students in your group.
- Each person in the group must be able to present every problem. The TA/Professor will select who presents which problem. The other group members may assist the presenter.
- You are not required to hand anything in at your presentation, but you may if you choose. If you do hand something in, it will be taken into consideration (in a non-negative way) in the grading.

Problems:

(28 points) 1. **Subset Sum is Hard** Consider the following problem, called Subset Sum: Given a set $X = \{x_1, ..., x_n\}$ of integers and an integer M, determine whether there exists a subset $S \subseteq X$ such that $\sum_{x \in S} x = M$. In this problem you will prove that Subset Sum is NP-Complete by reduction *from* one-in-three SAT. One in Three SAT is known to be NP-Complete. The one-in-three SAT problem is a variant of 3-SAT. Similar to 3-SAT the input instance is a collection of clauses, and each clause contains exactly three literals (either a variable or its negation). Unlike 3-SAT, however, we want to determine if there is a assignment of the variables such that each clause has *exactly* one true literal (equivalently exactly two false literals).

Consider the following reduction: Given a one-in-three SAT formula ϕ with clasues $C_1, ..., C_m$ and variables $x_1, ..., x_n$ we build a Subset Sum instance as follows:

- Set
$$M = \sum_{i=1}^{m+n} 2^{2i}$$
- Set
$$x_j^t = 2^{2m+2j} + \sum_{i \in P(x_j^t)} 2^{2i}$$
where $P(x_j^t)$ is the set of all clauses C_i that contain the literal x_j.
- Set
$$x_j^f = 2^{2m+2j} + \sum_{i \in P(x_j^f)} 2^{2i}$$
where $P(x_j^f)$ is the set of all clauses C_i that contain the literal $\neg x_j$
- Set
$$X = \{x_1^t, x_1^f, ..., x_n^t, x_n^f\}.$$

- Our final subset sum instance is

$$(X, M).$$

(a) Suppose that we find a subset $S \subseteq X$ such that

$$\sum_{x \in S} x = M.$$

Use S to construct a satisfying assignment to ϕ.

(b) Suppose that we find a satisfying assignment $x'_1, ..., x'_n$ to ϕ. Using this satisfying assignment show how to construct a subset $S \subseteq X$ such that

$$\sum_{x_j \in S} x_j = M.$$

(c) What (if anything) do we still need to prove before we can conclude that Subset Sum is NP-complete?

(28 points) 2. **Graduation** There is a list of requirements $r_1, r_2, \ldots, r_m$, where each requirement r_i is of the form: "you must take at least k_i courses from set S_i". A student *may* use the same course to fulfill several requirements. For example, if one requirement stated that a student must take at least one course from $\{A, B, C\}$, another required at least one course from $\{C, D, E\}$, and a third required at least one course from $\{A, F, G\}$, then a student would only have to take A and C to graduate. Now, consider an incoming freshman interested in finding the *minimum* number of courses that he (or she) needs to take in order to graduate.

(a) Prove that the problem faced by this freshman is NP-hard, even if each k_i is equal to 1. Specifically, consider the following decision problem: given n items labeled $1, 2, \ldots, n$, given m subsets of these items $S_1, S_2, \ldots, S_m$, and given an integer k, does there exist a set S of at most k items such that $|S \bigcap S_i| \geq 1$ for all S_i. Prove that this problem is NP-complete (also say why it is in NP).

(b) Show how you could use a polynomial-time algorithm for the above decision problem to also solve the search-version of the problem (i.e., actually find a minimum-sized set of courses to take).

(c) We could define a *fractional* version of the graduation problem by imagining that in each course taken, a student can elect to do a fraction of the work between 0.00 and 1.00, and that requirement r_i now states "the sum of your fractions of work in courses taken from set S_i must be at least k_i" (courses not taken count as 0). The student now wants to know the least total work needed to satisfy all requirements and graduate. Show how this problem can be solved using *linear programming*. Be sure to specify what the variables are, what the constraints are, and what you are trying to minimize or maximize.

(28 points) 3. **D's Max Flow Algo**

D's Max Flow algorithm finds the max flow on any flow network, G. In each phase, it constructs a blocking flow on the level graph of the network (described in Alg. 1). It

then computes the residual graph resulting from the blocking flow, and repeats another phase of the algorithm.

Alg 1 describes one phase. We traverse the level graph from source to sink in a depth-first fashion, advancing whenever possible and keeping track of the path from s to the current vertex. If we get all the way to t, we have found an augmenting path, and we augment by that path. If we get to a vertex with no outgoing edges, we delete that vertex (there is no path to t through it) and retreat.

In the following, u denotes the vertex currently being visited and p is a path from s to u.

Algorithm 1 Phase of D's Max Flow Algo

- **Initialize:** Construct a new level graph L_G using s as the root. Set $u := s$ and $p := [s]$. Go to **Advance**. (Note: The level graph L_G of G is the directed breadth-first search graph of G with root s and with sideways and back edges deleted.)
- **Advance:** If there is no edge out of u, go to **Retreat**. Otherwise, let (u, v) be any edge out of u. Set $p := p.append(v)$ and $u := v$. If $v \neq t$ then go to **Advance**. If $v = t$ then go to **Augment**.
- **Retreat:** If $u = s$ then halt. Otherwise, delete u and all adjacent edges from L_G and remove u from the end of p. Set u to be the last vertex on p. Go to **Advance**.
- **Augment:** Let Δ be the bottleneck capacity along path p. Augment by the path flow along p of value Δ, adjusting residual capacities along p. Delete newly saturated edges. Set u to be the last vertex on the path p reachable from s along unsaturated edges of p; that is, the start vertex of the first newly saturated edge on p. Set p to be the portion of p up to and including u. Go to **Advance**.

(a) Explain the complexity analysis for each part of one phase of the algorithm: Initialize, Advance, Retreat, and Augment. (For example, how many times can Advance be executed during one phase of the algorithm? How much total work is done by Advance during one phase?)

(b) Explain the complexity analysis for the entire algorithm, including an upper bound on the number of phases required to find the max flow.

(16 pts) 4. **The Curveball:** After your team present your solution to each of the above problems we reserve the right to make small changes to the problem, and ask how the answers would change. You will be expected to answer these questions on the fly. We are not going to tell you what these variations are in advance, that would defeat the purpose. Your best strategy is to make sure that everyone on your team *understands* each problem and its solution. If you have a good *understanding* of the solution then these variations should not be too difficult to answer.