CSI 4105 Design and Analysis of Algorithms II — Winter 2012
Computer Science — University of Ottawa

Homework Assignment #2 (100 points, weight 6.67%)
Due: Friday Mar 16, at 11:30 p.m. (in lecture)

---

1. (25 points) Let NearbySet be the problem defined as follows. Given a graph G and a number k, is there a way to select a set $N \subseteq V(G)$ with $|N| = k$ such that every vertex in the graph is either in N or is connected by an edge to a vertex in N. Show that NearbySet is NP-complete.

   For a graph $G = (V, E)$, we say that a vertex v is *dominated* by a set S if either $v \in S$ or there is a vertex $u \in S$ such that $uv \in E$. We must first show that NearbySet is in $\mathcal{NP}$. Given a set N of vertices, we can use the following algorithm to check if N is a solution:

   (a) Ensure that $|N| = k$ and $N \subseteq V(G)$. ($O(k)$ steps.)
   (b) Mark all nodes in $V(G)$ as undominated. ($O(|V|)$ steps.)
   (c) For each $v \in N$, mark v and the neighbourhood of v in $V(G)$ as dominated. ($O(k|V|)$ steps.)
   (d) Iterate over $V(G)$. If any nodes are undominated, return false. Otherwise, return true. ($O(|V|)$ steps.)

   This algorithm can be executed in time $O(k|V|)$, so it is a polynomial time verifier for NearbySet. Thus, NearbySet is in $\mathcal{NP}$.

   To show that NearbySet is NP-complete, we must find a polynomial time reduction from another problem in $\mathcal{NPC}$ to NearbySet. We will reduce from VertexCover and show VertexCover $\leq_p$ NearbySet. Let (G, k), with $G = (V, E)$, be an instance of VertexCover, i.e. the problem of determining whether or not there is a vertex cover of size k in G. Let I be the isolated vertices of G, i.e. the vertices in $V(G)$ such that they are in no edges of $E(G)$, and let $r = |I|$.

   We now create the graph $G' = (V', E')$ as follows: take $V' = V \cup \{w_e : e \in E\}$, and for $e = uv \in E$, we create three edges, uv, uw_e, and w_ev in E'. (The idea is that for each original edge, we add two copies in the new graph. In the second copy, we add one vertex, w_e, in the middle of the edge.) This reduction can clearly be done in polynomial time: we must find the isolated vertices in G, which can be done in time $O(|V||E|)$, and then creating the new vertices and edges can be done in time $O(|V| + |E|)$.

   We now claim that G has a vertex cover of size at most k if and only if G' contains a nearby set of size at most $k + r$. If G has a vertex cover, say S, of size at most k, then we claim that $N = S \cup I$ is a nearby set of G' of size at most $k + r$. We must show that for every vertex $v \notin N$, there is a vertex $u \in N$ such that $vu \in E'$.

Since $V' = V \cup \{w_e : e \in E\}$, we first consider vertices of the form $w_e \in V'$ for $e \in E$. This vertex of G' is in precisely two edges of G', say uw_e and w_ev, with uv an edge of the original graph G by construction. Since S is a vertex cover of the original graph G, then either u or v must be in S since $uv \in E$. Without loss of generality, say $u \in S$. Thus, in the new graph G', the vertex w_e is dominated by $u \in N$.

We now consider all vertices $v \in V'$ that were in the original graph G, so $v \in V$. If v is isolated in G, then $v \in I$, so $v \in N$; thus v is dominated by N. Otherwise, there is some edge $uv \in E$. Since S is a vertex cover of G, either u or v in S. Thus, since $uv \in E'$, the inclusion of u or v in N ensures that v is dominated in G' as required.

If v is an isolated vertex of G', then $v \in I \subseteq N$, so v is dominated by N. If $v \in S$, then obviously v is dominated. If $v \notin S$, then since S is a vertex cover of G, there is some vertex $u \in V$ such that $u \in N$ and $uv \in E$.

We must now show that if N is a nearby set of G' of size at most $k + r$, then there is a vertex cover of size at most k in G. Clearly, all isolated vertices of G', which are precisely the isolated vertices of G, must be in N, so take $N' = N \setminus I$, which has size at most k. We show that from this, we can build a vertex cover S for G in the following way: we may have two types of vertices in N, either vertices of the original graph G, or vertices of the form w_e constructed by bissecting edges of the original graph G. If $u \in S$ is a vertex of the original graph, add it to S. If it is of the form w_e, then $e = vz$ is an edge of the original graph, so add either v or z, chosen arbitrarily, to S. Clearly, the size of S is at most k, so we show that S is a vertex cover of the original graph G. If $e = uv \in E$, then we have edges uv, uw_e, and w_ev in E', so a nearby set in G' must have included one of u, v, or w_e. Thus, one of u or v must be in S, so S is a vertex cover, as required.

Thus, NEARBYSET is in $\mathcal{NPC}$.

2. (25 marks) Consider the treasure splitting problem: there are n objects $1, 2, \dots, n$ each of value v_i, $1 \le i \le n$. Two pirates need to split the treasures evenly. The TREASURESPLITTING problem asks: given $v_1, v_2, \dots, v_n$ is it possible to partition $\{1, 2, \dots, n\}$ into two sets S_1, S_2 (partitioning means $S_1 \cup S_2 = \{1, 2, \dots, n\}$ and $S_1 \cap S_2 = \emptyset$) such that

$$\sum_{i \in S_1} v_i = \sum_{j \in S_2} v_j \ ?$$

Prove that TREASURESPLITTING is NP-complete.

We first show that TREASURESPLITTING is in $\mathcal{NP}$. Let $N = \{1, \dots n\}$, and say we have a set S that is a certificate for the problem. (The treasure is split into S and $N \setminus S$.) We can use the following algorithm to verify whether or not this is a solution:

(a) Ensure that $S \subseteq N$. This can be done in time $O(n)$.

(b) Calculate the sums $\sum_{i \in S} v_i$ and $\sum_{j \in N \setminus S} v_j$. This can be done in time $O(n)$.

(c) If the sums are equal, return true; otherwise, return false.

Thus, TreasureSplitting is in $\mathcal{NP}$. We now give a polynomial time reduction from SubsetSum, which is in $\mathcal{NPC}$, to show that TreasureSplitting is NP-complete. Consider an arbitrary subset sum problem with numbers $w_1, \ldots, w_n$ and target sum W. We now construct an equivalent instance of treasure splitting. Let $T = \sum_{i=1}^{n} w_i$ be the total sum. Create two new numbers:

$$w_{n+1} = W + 1, \qquad w_{n+2} = T + 1 - W.$$

Let $N = \{1, \ldots, n+2\}$. We now have that $\sum_{i=1}^{n+2} w_i = 2T+2$. Clearly, adding two elements can be done in polynomial time. We claim that the treasure splitting problem with $w_1, \ldots, w_{n+2}$ is equivalent to the original problem.

Assume that the answer to the subset sum problem is *yes* with solution S such that $\sum_{i \in S} w_i = W$. Consider the set $S' = S \cup \{w_{n+2}\}$. We have that:

$$\sum_{i \in S'} w_i = w_{n+2} + \sum_{i \in S} w_i = (T + 1 - W) + W = T + 1.$$

This is precisely $\frac{1}{2} \sum_{i=1}^{n+2} w_i = \frac{1}{2}(2T + 2)$, so then necessarily, $N \setminus S'$ has sum:

$$\sum_{i \in N \setminus S'} w_i = \sum_{i=1}^{n+2} w_i - \sum_{j \in S'} w_j = (2T + 2) - (T + 1) = T + 1.$$

Thus, S' and $N \setminus S'$ is a partition of N' and is a solution to the treasure splitting problem.

Now, assume we have some *yes* solution, say S, to the treasure splitting problem constructed from an instance of the subsetsum problem. We show that we can construct an instance of the subset sum problem from S. We have that:

$$\sum_{i \in S} w_i = \frac{1}{2} \sum_{i=1}^{n+2} w_i = T + 1.$$

Similarly, the value of the remaining treasure, $N' \setminus S$, must also sum to $T + 1$. Now, we have that if $w_{n+1} \in S$, then $w_{n+2} \in N \setminus S$: otherwise, the sum of S would be greater than or equal to $(T + 1 - W) + (W + 1) = T + 2$, which is too large. Assume without loss of generality that S is the set of the partition that contains w_{n+2}. We claim that $S' = S \setminus \{w_{n+2}\}$ is a solution to the original subset sum problem. We have that:

$$\sum_{i \in S'} w_i = \left(\sum_{i \in S} w_i \right) - w_{n+2} = (T + 1) - (T + 1 - W) = W.$$

Thus, S' is a solution to the subsetsum problem, as required.

Thus, we have a solution to the subsetsum instance if and only if we have a solution to the equivalent treasure splitting instance, so this is a polynomial time reduction from SubsetSum to TreasureSplitting. Hence, TreasureSplitting is in $\mathcal{NPC}$.

3. (25 points) Consider a special case of QSAT (Quantified 3-SAT) in which the formula $\phi(x_1, \ldots, x_n)$ has no negated variables. We define the decision problem NNQSAT to be the problem of deciding the truth value of:

$$\exists x_1 \forall x_2 \ldots \exists x_{n-2} \forall x_{n-1} \exists x_n \; \phi(x_1, x_2, \ldots, x_n),$$

where n is odd and $\phi(x_1, x_2, \ldots, x_n)$ is a 3-CNF formula with no negated variables. Give a polynomial time algorithm to solve NNQSAT; analyse the running time of the algorithm.

To make it easier to think about the problem, we can instead consider this problem to be an competitive instance of CSAT: we have two players, P1 and P2, where P1 and P2 alternate turns setting the values of variables in sequential order. Thus, P1 sets the values of variables x_1, x_3, x_5, etc, and P2 sets the values of the variables x_2, x_4, x_6 in order, with the goal of P1 trying to construct an assignment that makes the formula true, and P2 trying to construct an assignment that makes the formula false. The question then becomes: can P1 force a win, i.e. can P1 find a strategy so that he always wins? This is equivalent to a solution to NNQSAT: P1 can force a win if and only if regardless of the choice of assignments for P2, he can find a winning strategy; thus, for all possible assignments made by P2, he must be able to find a winning assignment. Thus, the "exists" corresponds to the specific choices made by P1, and the "for-all" corresponds to all possible choices made by P2.

We claim that P1 can force a win if and only if every clause in $\phi(x_1, \ldots, x_n)$ has a variable of odd index. If this is the case, then P1 can simply pick 1 for every variable of odd index, thus making each clause true. If this is not the case, though, then for the clause of all even indexed variables, P2 can simply pick 0 for all these variables, thus making the entire expression false.

Thus, determining the truth value of the expression equates to determining if there is a clause that contains only even-indexed variables. To do this, we must examine $\phi(x_1, \ldots, x_n)$: we iterate over the clauses, and for each clause, we determine if the clause has only even-indexed variables. If this is the case for any clause, we return false, indicating that the expression is false; otherwise, we return true, indicating that the expression is true.

The number of possible clauses is bounded by $\binom{n}{3} \in O(n^3)$ (for each clause, pick three indices for the variables in the clause). Thus, our algorithm iterates over each clause, which can be done in time at most $O(n^3)$, and then determines if the variables are even- or odd-indexed, which can be done in time $O(3)$ for each clause. Thus, the whole algorithm can be executed in time $O(n^3)$.

4. (25 points) Define the choice set and describe a backtracking algorithm for the problem: given G and k, find all k-vertex colourings of G.

Let $G = (V = \{v_1, \ldots, v_n\}, E)$ be an arbitrary graph over which we want to find all k-colourings, so we want to assign values $\{1, \ldots k\}$ to each vertex so that no two adjacent vertices have the same colour. Let $X = [x_1, \ldots, x_n]$ be our solution list. This equates to having the condition that if $v_iv_j \in E$, then $x_i \neq x_j$. During the backtracking, when choosing a colour for vertex v_l, $1 \leq l \leq n$, which equates to choosing a value for x_l, we have that P_l, the possibility set for x_l, is all of the colours, so $P_l = \{1, \ldots, k\}$. However, we want to restrict our choice set for x_l to only the valid colour choices for v_l, namely the colours that v_l can assume that do not clash with its neighbours who have already been assigned colours. Thus, this amounts to:

$$C_l = \{1, \ldots, k\} \setminus \{x_i : 0 \leq i < l, v_iv_l \in E\}.$$

Thus, our full backtracking algorithm is as in Algorithm 1. We initially invoke the algorithm by executing FindAllColourings$(G, X, k, 1)$ for $X = [\underbrace{0, \ldots, 0}_{n}]$.

Algorithm 1 FindAllColourings$(G = (V = \{v_1, \ldots, v_n\}, E)$, $X = [x_1, \ldots, x_n]$, k, $l)$

```
if $l = n + 1$ then
    $X$ is a valid $k$-colouring, so output it.
    return
end if
$C_l = \{1, \ldots, k\} \setminus \{x_i : 0 \leq i < l, v_iv_l \in E\}$
for $c \in C_l$ do
    $x_l = c$
    FindAllColourings($G$, $X$, $k$, $l + 1$)
end for
```