CS 600 Homework 9 Solutions

R-17.3 SAT is NP-complete

SAT takes an arbitrary boolean formula S as input and asks whether S is satisfiable. We have to show that SAT is NP-complete. We know that any boolean formula can be converted to conjunctive normal form or CNF. Further, we know that CNF-SAT is NP-complete.

We can show that CNF-SAT $\xrightarrow{\text{poly}}$ SAT by using the restriction form of polynomial-time reduction. CNT-SAT is a special case of SAT where the input boolean formula given to SAT is converted to CNF and passed along. By this, we have SAT is NP-complete.

R-17.7 CLIQUE is in NP

A clique in a graph G is a subset C of vertices such that, for each $v, w \in C$ with $v \neq w$, (v, w) is an edge in G. CLIQUE takes an inputs a graph G and an integer k and asks whether there is a clique in G of size at least k.

In order to show that CLIQUE is in NP, we can construct a non-deterministic algorithm that accepts instances of CLIQUE in polynomial time. We assume that the nodes of the graph are labeled 1 to N. We iteratively call the choose method to non-deterministically select k nodes from the graph which we place in a set C'. Now, we scan through all the vertices in C' and check whether C' forms a clique in G of size at least k. This takes polynomial time. Therefore CLIQUE is in NP.

C-17.10 INDEPENDENT-SET is NP-complete

The INDEPENDENT-SET problem takes a graph G and an integer k and asks whether G contains a set I of vertices of size k such that, for any $v, w \in I$, there is no edge (v, w) in G. For the purpose of this problem, we assume that the vertices of G are labeled from 1 to N.

First we show that this problem is in NP. To accept instances of the INDPENDENT-SET problem, we can call the choose method iteratively to select a collection C of k vertices from the input graph G. We then scan through the vertices in C and check whether they are independent. This takes polynomial time to compute. Therefore INDEPENDENT-SET is NP.

To show that INDEPENDENT-SET is NP-complete, we can show that CLIQUE $\xrightarrow{\text{poly}}$ INDEPENDENT-SET by using the local replacement form for polynomial-time reduction. We define the integer parameter for INDEPENDENT-SET as $n - k$, where k is the integer parameter of CLIQUE, and n is the number of vertices in G. That is, we note that G has a clique of size k if and only if it has an independent set of size $n - k$. Since CLIQUE is NP-hard, so is INDEPENDENT-SET.

A-17.5 CableClock security problem

Many computers in CableClock are infected with malware. The malware comes from a number of websites that users visit. The most likely candidate websites that inject this malware would be in the smallest collection of websites that are visited by all infected computers. We need to show that the decision version of this problem is NP-complete.

We are given a list of all websites visited by each infected computer. The decision problem asks whether there is a collection C of websites, of size at most k, such that each infected computer has visited all the websites in C.

More formally, suppose there are m infected computers and we have a collection of m sets $S_1, S_2, \ldots, S_m$. Each set S_i contains the websites that the i^{th} computer has visited. The decision problem asks whether there is a set C of size at most k such that,

$$\bigcap_{i=1}^{m} S_i = C$$

In order to show that this problem, CABLE-CLOCK, is in NP we can construct a non-deterministic algorithm that accepts instances in polynomial time. We call the choose method k times to select a collection, C', of k websites. Then we keep checking whether $\bigcap_{i=1}^{m} S_i = C'$. Since this would take polynomial time, CABLE-CLOCK is in NP.

In order to show that CABLE-CLOCK is NP-complete, we can show that SET-COVER $\xrightarrow{\text{poly}}$ CABLE-CLOCK.

Recall that SET-COVER takes a collection of m sets $S_1, S_2, \ldots, S_m$ and an integer parameter k as input and asks whether there is a subcollection of k sets $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ such that,

$$\bigcup_{i=1}^{m} S_i = \bigcup_{j=1}^{k} S_{i_j}$$

Let (M, k) be an instance of the SET-COVER problem where $M = \{S_i \mid 1 \leq i \leq m\}$ is the collection of sets. In order to reduce this to an instance of CABLE-CLOCK in polynomial-time, we can use the following law (from DeMorgan's laws for family of sets)

$$\left(\bigcap_{i \in M} S_i\right)^c = \bigcup_{i \in M} (S_i)^c$$

where $(S_i)^c$ represents the complement of S_i, defined as $\{x \mid x \in M_j, x \notin S_i, i \neq j\}$.

We start by first computing the complement of each set in M and take the union of the resulting sets. We then take the complement of the resulting collection to arrive at $\bigcap_{i \in M} S_i$. This reduction takes polynomial time. Therefore, CABLE-CLOCK is NP-complete.

R-18.11 Optimal solution to an instance of SET-COVER

We are given the following collection of sets,

$$S_1 = \{1,2,3,4,5,6\} \quad S_2 = \{5,6,8,9\} \quad S_3 = \{1,4,7,10\}$$
$$S_4 = \{2,5,7,8,11\} \quad S_5 = \{3,6,9,12\} \quad S_6 = \{10,11\}$$

We need to find a collection of k sets, $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$ such that

$$\bigcup_{i=1}^{6} S_i = \bigcup_{j=1}^{k} S_{i_j} = \{1,2,3,4,5,6,7,8,9,10,11,12\}$$

We can construct a Venn diagram (not shown here) for the sets to verify that the optimal solution is $\{S_3, S_4, S_5\}$ where $k = 3$.

A version of the greedy algorithm, SetCoverGreedy, selects sets from the input collection of sets one at a time, each time selecting the set that has the most uncovered elements. If the union of all the collected sets is equal to the universe, then we have found a valid set cover.

```
**Algorithm** SetCoverGreedy
    **Input**: A collection of $m$ sets, $S_1, S_2, \ldots, S_m$
    **Output**: A valid set cover

    $C \leftarrow \emptyset, E \leftarrow \emptyset$
    **while** $E \neq \bigcup_{i=1}^{m} S_i$ **do**
        select a set $S_i$ that has the most number of uncovered elements
        add $S_i$ to $C$
        $E \leftarrow E \cup S_i$
    **return** $C$
```

Performing this algorithm on the sets given above, we get $\{S_1, S_4, S_5, S_6\}$. Depending on the implementation, the algorithm can also pick $\{S_1, S_4, S_5, S_6\}$ since S_3 and S_6 have the same number of uncovered elements.

C-18.1 General optimization version of TSP

Here we consider the general optimization version of the TSP problem, GENERAL-TSP, where the underlying graph need not satisfy the triangle inequality. We need to show that for any fixed $\delta \geq 1$, there is no polynomial-time δ-approximation algorithm unless P $=$ NP.

Suppose, we had an algorithm A that solved GENERAL-TSP in polynomial time with an approximation ratio of $\delta \geq 1$. We want to show that if such an algorithm existed, we could solve HAMILTONIAN-CYCLE in polynomial time (implying that P $=$ NP).

We start by showing that HAMILTONIAN-CYCLE $\xrightarrow{\text{poly}}$ GENERAL-TSP. Recall that HAMILTONIAN-CYCLE takes a graph G and asks whether there is a cycle in G that visits each vertex in G exactly once, returning to its starting vertex.

Suppose we are given an instance G to the HAMILTONIAN-CYCLE problem. We construct a complete graph H for the n vertices of G and weight these edges so that an egde of H also in G has a cost of 1 but an edge of H not in G has a cost of $(1 + \delta n)$. This reduction takes polynomial time. Note that if we include just one of these edges in a cycle, the the cost of that cycle is at least $n + \delta n = (1 + \delta)n$. Thus, a strict $(1 + \delta)$-approximation of the GENERAL-TSP of cost n can only be achieved using the original edges in G that form a Hamiltonian cycle.

This implies that our algorithm, A, can solve HAMILTONIAN-CYCLE in polynomial time. Since HAMILTONIAN-CYCLE is NP-complete, this means that we can solve all problems in NP deterministically in polynomial time, implying that P $=$ NP.

A-18.3 FedUP problem

It is our job to ship a set of n boxes from Rhode Island to California using a give collection of trucks. The trucks will be weighted at various points along the route and FedUP will have to pay a penalty if any of these trucks are overweight. Our task is to minimize the weight of the most heavily loaded truck.

Note that there is no limit on the capacity of the trucks in this version of the problem. Also note that the boxes are loaded in an arbitrary order. We cannot assume we can sort them first, and we don't need to. Images these boxes are coming on a rail from a warehouse and are going into FedUP's trucks.

Assume the boxes are labeled $a_1, a_2, \dots, a_n$ where each a_i weighs w_i. Consider the greedy algorithm of always assigning the next box to the least loaded truck. This algorithm actually forces the loading to become pretty close to average; in fact half as good as the optimal one. (There are other algorithms that provide $3/2$ approximation.)

We need to prove that at the end, the load of the most heavily weighted truck, with weight L, is at most 2 times the optimum weight *OPT*.

Let $g = (\sum w_i)/k$ be the average load of all k loaded trucks. Clearly, $\mathit{OPT} \geq g$ and each $w_i \leq \mathit{OPT}$. Now we can prove by contradiction that this greedy algorithm is a 2-approximation.

Assume that $L > 2 * \mathit{OPT}$ for the most heavily loaded truck. Now, when the last load a_m was added to the truck t_k, the truck t_k had the least weight. This weight, T, is $T = L - w_m$ and clearly $T \leq g$.

From the assumption above, we have $L = T + w_m > 2 * \mathit{OPT}$. Further, we know that $T + w_m \leq g + w_m \leq \mathit{OPT} + w_m$. This implies that $\mathit{OPT} + w_m > 2 * \mathit{OPT}$, resulting in the contradictory statement that $w_m > \mathit{OPT}$. Therefore $L \leq 2 * \mathit{OPT}$.