CS 170 HW 10

Due on 2018-04-09, at 11:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★) Runtime of NP

True or False (with proof): Suppose we can show for some fixed k, an NP-complete problem P has a time $O(n^k)$ algorithm. Then every language in NP has a $O(n^k)$ time algorithm.

Solution: False. The reduction f_L from an arbitary language $L \in \mathsf{NP}$ is guaranteed to run in time $O(n^{c_L})$ and produce a problem $f(x)$ of the NP-complete problem of size $O(n^{c'_L})$ for constants c_L and c'_L. However, these can be arbitrarily larger than k.

3 (★★★) 2-SAT and Variants

While we showed that 3-SAT last week was NP-complete, today we will explore the variant 2-SAT, where each clause contains at most 2 literals (hereby called a 2-clause).

(a) Show that 2-SAT is in P.

Solution: Let φ be a formula acting on n literals $x_1, \ldots, x_n$. Generate a graph with $2n$ vertices representing the set of literals and their negations. For each clause $(a \vee b)$ of φ add the edges $\neg a \Rightarrow b$ and $\neg b \Rightarrow a$. Run your favorite path-finding algorithm to check for any path x_i to $\neg x_i$ or path $\neg x_i$ to x_i for $i = 1, \ldots, n$. If any path is found, report unsatisfiable. Otherwise, report satisfiable.

In the case that the algorithm reports unsatisfiable, notice that the edges of the graph are necessary implications. Thus, some literal must be true and false simultaneously, a contradiction. In the case that the algorithm reports satisfiable, there is a simple satisfying assignment: Pick an unassigned literal s with no path from s to $\neg s$. Assign it to true and all the reachable vertices from it. Assign the negations of these literals as false. Repeat for any unassigned literal.

(b) The problem of Max-2-SAT is defined as follows. Let $C_1, \ldots, C_m$ be a collection of 2-clauses and k a non-negative integer . Output 1 if and only if there exists an assignment of the literals such that at least k of the clauses are satisfied.

Show that Max-2-SAT is NP-complete. Reduce from Max-Cut, which is the problem of determining if given input a graph G and an integer k, determining if there exists a cut of weight at least k.

Solution:

(a)

(b) As suggested, we reduce unweighted `Max-Cut` to `Max-2-SAT`. Let a graph $G = (V, E)$ and an integer k be given. We want to find whether there is a cut in the graph of size k or more. We construct a boolean formula based on the graph. Every assignment of this formula will correspond to a cut in the graph.

For each vertex $u \in V$, we add a variable x_u, representing whether u is in S (true) or in $V \setminus S$ (false). For each edge $(u, v) \in E$, we add *two* clauses: $(x_u \vee x_v)$ and $(\overline{x_u} \vee \overline{x_v})$. Then, if (u, v) crosses the cut, *both* of these clauses will be satisfied (if $u \in S$ and $v \notin S$, then x_u and $\overline{x_v}$ are both true; if $u \notin S$ and $v \in S$, then $\overline{x_u}$ and x_v are both true). If (u, v) does not cross the cut, then exactly one of the clauses will be satisfied (the first if both u and v are in S, the second if neither is). Thus, any cut of size q in the graph corresponds to an assignment of values to variables that satisfies exactly $|E| + q$ clauses (1 for each of the $|E| - q$ pairs representing non-cut edges, 2 for each of the q pairs representing cut edges).

Thus, to solve `Max-Cut`(G, k), we construct this formula Φ_G and return `Max-2-SAT`$(\Phi_G, |E|+k)$.

4 (★★★) More Reductions

Given a set S of non-negative integers $[a_1, a_2, \ldots, a_n]$, consider the following problems:

1 **Partition**: Determine whether there is a subset $P \subseteq S$ such that $\sum_{i \in P} a_i = \sum_{j \in S \setminus P} a_j$

2 **Subset Sum**: Given some integer k, determine whether there is a subset $P \subseteq S$ such that $\sum_{i \in P} a_i = k$

3 **Knapsack**: Given some set of items each with weight w_i and value v_i, as well as fixed numbers W and V there is some subset P such that $\sum_{i \in P} w_i \leq W$ and $\sum_{i \in P} v_i \geq V$

For each of the following briefly describe your reduction and provide a few sentences to justify runtime and correctness.

a Find a linear time reduction from Subset Sum to Partition.

b Find a linear time reduction from Subset Sum to Knapsack.

Solution:

1. Suppose we are given some set A with target sum t. Let s be the sum of all elements in A. If $s - 2t \geq 0$, generate a new set $A' = A \cup \{s - 2t\}$. If A' can be partitioned, then there is a subset of A that sums to t.

   We know that the two sets in our partition must each sum to $s - t$ since the sum of all elements will be $2s - 2t$. One of these sets, must contain the element $s - 2t$. Thus the remaining elements in this set sum to t.

If $s - 2t \leq 0$, generate a new set $A' = A \cup \{2t - s\}$. if A' can be partitioned, then there is a subset of A that sums to t.

We know that the two sets in our partition must each sum to t since the sum of all elemens will be $2t$. The set that does not contain $\{2t - s\}$ will be our solution to subset sum.

This reduction also clear operates in $O(n)$, as we simply new to generate a new set with a single additional element (whose value is determiend by summing all the elements of the set A).

2. Suppose we are given some set A with target sum t. For each element k of the set, create an item with weight k and value k. Let $V = t$ and $W = t$. We know Knapsack will determine if there is a combination of items with sum of weights $\leq t$ and values $\geq t$. Because the weights and values are the same, we know (Sum of chosen weights) $=$ (Sum of chosen values) $= t$. And since each weight/value pair is exactly the value of one of the original elements of A, we know that there will be a solution to our Knapsack problem iff there is one for our subset sum problem. This solutions is linear time, as we need constant work for each element of A to generate our new input.

5 (★★★★) An approximation algorithm for Subset-Sum

The **SUBSETSUM** problem is defined as: Given $A = \{a_1, \ldots, a_m\}$ positive integers, and a positive integer t (the "target"), find a subset $A' \subseteq A$ s.t. $\sum_{a \in A'} a = t$.

(a) Give a Dynamic Programming algorithm for Subset Sum. Argue that if the inputs (the a_i and t) are given in unary, this algorithm runs in polynomial time.

Solution: There is a reduction to Knapsack by defining items $1, \ldots, m$ with weights and values $w_i = v_i = a_i$ and $W = V = t$. Clearly, if $\sum w_i \leq W$ then $\sum a_i \leq t$ and $\sum v_i \leq V$ then $\sum a_i \geq t$ proving $\sum a_i = t$. Therefore, correctness and runtime follows from Knapsack.

If the input was in unary then the runtime would be linear in the input.

(b) **SUBSETSUM** (with the integers given in ordinary binary notation) is known to be **NP**-complete; e.g., there is a reduction from 3-SAT. So, instead of solving it exactly, we are going to come up with an approximation algorithm. For a fixed $\epsilon > 0$, we want to *decide* if a subset $A' \subseteq A$ exists s.t.

$$\sum_{a \in A'} a \in [t(1-\epsilon), t(1+\epsilon)]$$

Specifically, we are going to build a streaming algorithm. In a streaming algorithm, the elements $a_1, \ldots, a_m$ are received one at a time and we take some immediate action when they are received. (We'll be more formal about the model later in the course.)

Here is a sketch for the algorithm. **Fill in the details and prove its correctness.**

The algorithm maintains T, a set of non-intersecting intervals. Initially, T is a set consisting of a single interval: $\{[t(1-\epsilon), t(1+\epsilon)]\}$.
While the stream hasn't ended:

(a) Let $a_j \leftarrow$ the next element in the stream.

(b) If there is an interval $[\alpha, \beta] \in T$ such that $a_j \in [\alpha, \beta]$ then output YES.

(c) Update T using a_j. [**You need to explain how T is updated.**]

Otherwise, output NO.

Solution: Consider the following algorithm:

Maintain a list of non-intersecting intervals T initialized as $\{[t(1-\epsilon), t(1+\epsilon)]\}$. Maintain T as a (linked) list of intervals sorted by left endpoint. While the stream hasn't ended

- Let $a_j \leftarrow$ be the next stream element.
- Check via binary search if there is a $[\alpha, \beta] \in T$ s.t. $a_j \in [\alpha, \beta]$. If so, output YES.
- Calculate $[\alpha - a_j, \beta - a_j]$ for each interval $[\alpha, \beta] \in T$, and then in a linear sweep over T merge these two sorted lists.
- In another linear sweep over T, for the first interval $[\alpha, \beta] \in T$, find the last interval $[\alpha', \beta']$ such that $\alpha' < \beta$. Discard all the intervals and add the new interval $[\alpha, \beta']$.

If the steam ends, output NO.

Proof of Correctness: We claim that if a subset $A' \subseteq \{a_1, \ldots, a_j\}$ exists s.t. $\sum_{a \in A'} a \in [t(1-\epsilon), t(1+\epsilon)]$, then it is found by the time a_j is streamed. Proof by indution. If $j = 1$, then a subset only exists is $a_j \in [t(1-\epsilon), t(1+\epsilon)]$, which is trivially checked for. For induction, let a' be the last element of A'. Therefore,

$$a' \in \left[t(1-\epsilon) - \sum_{a \in A'/\{a\}} a, t(1+\epsilon) - \sum_{a \in A'/\{a\}} a \right]$$

Notice that the construction precisely shifts instervals by a_j on stream element a_j and then collapses intersecting intervals to save space. As a' is the last element of A', then this interval must be contained in some interval of T, and thus the output would be YES.

Conversely, if the algorithm ever outputs YES on steam element a_j, then we can look to see how a_j was included in some interval in T. Clearly, the interval it is in was derived from a process of collapsing intersecting intervals and shift intervals by elements of the stream. Those shifts give proof that a subset sum exists.

(c) Now we are going to prove that this algorithm runs in **poly**$(n, 1/\epsilon)$ time and space where n is the length of the input.

Give good space and time bounds on this algorithm (in particular be explicit about the polynomial and the $O()$ above). Hint: first find an upper bound on $|T|$.

Solution: Notice that the intervals of T are non-intersecting and have minimum length $2t\epsilon$. Therefore, the total number of intervals is at most $t/(2t\epsilon) = O(1/\epsilon)$ as the intervals lie in $[0, t]$. We can store each interval uniquely by its endpoints which require $O(\log t)$ space per interval yielding the whole storage space of T as $O(\log t/\epsilon)$. We call this space size S from now on.

Conducting binary search for an interval containing a_j takes $O(\log S)$ space. The linear sweep over T for adding intervals will take $O(S)$ as it is the length of the sweep. Collapsing the intervals is a linear procedure so will take $O(S)$ time. Total time per stream element: $O(S)$.

As this occurs at most $O(m)$ times, the runtime is $O(mS) = O(m \log t/\epsilon)$. The space complexity is only storing T which takes $O(\log t/\epsilon)$ space.

(d) We would like to also extract the set A' when the answer is YES. Using the previous algorithm as a subroutine extract a set A' that achieves the approximation bound if one exists. Give a total runtime bound.

Solution: We first use the subroutine to check if a solution exists. If one exists, we can check if there is a solution that includes a_1 by using the subroutine on input $\{a_2, \dots, a_n\}, t_1 = t - a_1$ and ϵ_1 which satisifies $\epsilon_1(t - a_1) = t\epsilon$. If this outputs yes, then we know some solution including a_1 exists. If not, we know that all solutions exclude a_1. We can apply this inductively to find if $a_2, a_3, \dots$ are in a satisfying subset.

This takes a total of $O(m)$ subroutine calls. Notice that the recursive calls have smaller values for m, t and $1/\epsilon$ than the original. Therefore, the total runtime is $O(m^2 \log t/\epsilon)$.

6 (★★★★★) Steiner Tree

The Steiner tree problem is the following:

Input: An undirected graph $G = (V, E)$ with non-negative edge weights $\text{wt} : E \to \mathbb{N}$, a set $S \subseteq V$ of special nodes.

Output: A Steiner tree for S whose total weight is minimal

A Steiner tree for S is a tree composed of edges from G that spans (connects) all of the special nodes S. In other words, a Steiner tree is a subset $E' \subseteq E$ of edges, such that for every $s, t \in S$, there is a path from s to t using only edges from E'. The total weight of the Steiner tree is the sum of the weights included in the tree, i.e., $\sum_{e \in E'} \text{wt}(e)$.

The Steiner tree problem has many applications in different areas, including creating genealogy trees to represent the evolutionary tree of life, designing efficient networks, to even planning water pipes or heating ducts in buildings. Unfortunately, it is NP-hard.

Here is an approximation algorithm for this problem:

1. Compute the shortest distance between all pairs of special nodes. Use these distances to create a modified and complete graph $G' = (S, E_S)$, which uses the special nodes as vertices, and the weight of the edge between two vertices is the shortest distance between them.

2. Find the minimal spanning tree of G'. Call it M.

3. Reinterpret M to give us a Steiner tree for G: for edge in M, say an edge (r, s), select the edges in G that correspond to the shortest path from r to s. Put together all the selected edges to get a Steiner tree.

Prove that this algorithm achieves an approximation ratio of 2.

(Note that the efficiency of an approximation algorithm does not contradict NP-completeness because it gives an approximate (rather than an exact) solution to the problem.)

Solution: Let W be the weight of the optimal Steiner tree. Let H be the tree outputted by the algorithm. It is clear that $weight(H) \leq weight(M)$, as the quantity $weight(M)$ can count edges in the tree H multiple times. Using the optimal tree, we can construct a tour through all of the special vertices as follows: pick any vertex in the optimal tree, and explore using the DFS ordering. Every time we push a vertex v onto the stack, we add the edge (u, v) to the path, where u is the current vertex. Every time we pop a vertex v off the stack, we add (v, u) to the path, where u is the vertex that is on top of the stack after the pop. This constructs a tour that visits all the vertices in the optimal tree, and uses each edge exactly twice. Since it visits all the vertices in the optimal tree, in particular it visits all the special vertices. The tour has weight $2W$ since it uses each edge exactly twice.

Claim: Let F be any tour that goes through all of the special vertices. Then $weight(M) \leq weight(F)$.

Proof: From any tour F, we can construct a tree $T' \in G' = (S, E_S)$ that visits all the special vertices. First, we can construct a tree T so that the edges of T are in E_S, but the weights are different. We do this by simply following the tour and combining segments of the form $(s_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, s_n)$ into one edge (s_1, s_n) of weight $weight(s_1, v_2) + \cdots + weight(v_{n-1}, s_n)$, where $s_1, s_n \in S$, and all other $v_i \notin S$. If an edge is repeated, we ignore it, and we ignore any edge that would create a cycle. This guarantees that T a tree. Because we count the weight of each edge from F at most once, we see that $weight(T) \leq weight(F)$. Now consider T', a spanning tree in $G' = (S, E_S)$, where the edges of T' are the same as T, just with possibly smaller weights. The weights could be smaller since the edge weights in G' are equal to the lengths of the shortest paths, while the edge weights for T are only the lengths of paths (in particular, not necessarily shortest paths). This shows that $weight(T') \leq weight(T)$. Since M is an MST in G', $weight(M) \leq weight(T')$. Therefore, $weight(M) \leq weight(T') \leq weight(T) \leq weight(F)$, for any tour F. This proves the claim.

In particular, from the tour we constructed earlier we see that $weight(M) \leq 2W$. Combining with the inequality from earlier, we get that $weight(H) \leq 2W$. But W is the optimal weight, so $\frac{weight(H)}{W} \leq \frac{2W}{W} = 2$. Thus, the algorithm achieves an approximation ratio of 2.