CPSC 320 2018W2: Assignment 5 (Last assignment!)

Please follow the guidelines given in Assignment 1 for submission to Gradescope, and for group collaboration. Remember to provide short justifications for your answers. Submit by the deadline **Monday April 1, 2019 at 10PM**.

Use LaTeXto prepare your answers. Easiest will be to use the .tex file provided. Please enclose each paragraph of your solution in `\soln{Your solution here...}`.

Your solution will then appear in dark blue, making it a lot easier for TAs to find the parts that you wrote.

1 Subset sums mod n

You've now seen examples of dynamic programming problems where the recurrence describes a number (e.g., the length of a longest common subsequence), and one example where the recurrence value is Boolean (true or false, indicating whether or not a there is a subset of a set items whose sum is a given value). Here you'll work with a problem where the recurrence describes a set.

Let $S = \{x_1, x_2, \ldots, x_n\}$ be a set of nonnegative integers. We say that a value v, with $0 \leq v \leq n-1$, is *feasible* with respect to S if for some non-empty subset R of S,

$$\sum_{x \in R} x = v \pmod{n}.$$

1. For $1 \leq i \leq n$, let $V(i)$ be the set of feasible values that can be obtained from $\{x_1, x_2, \ldots, x_i\}$. Explain why the following recurrence holds for $i \geq 1$, if we define $V(0) = \emptyset$ (the empty set):

$$V(i) = V(i-1) + \cup_{v \in V(i-1)} \{v + x_i \pmod{n}\} \cup \{x_i \pmod{n}\}.$$

   We can see this recurrence in three parts, the first part $V(i-1)$ ensure that $V(i)$ contains the v_s for all subsets of the previous set $\{x_1, x_2 ... x_{i-1}\}$.

   With $\cup_{v \in V(i-1)} \{v + x_i \pmod{n}\}$ we are addding all the v_s for all the combinations of the previous subsets in $V(i-1)$ with our new value x_i, since we only care about the $\pmod{n}$ of the sum of each subset, we dont need to re calculate the sum over each subset to include x_i we just need the corresponding v of each subset then if we add x_i to it and compute $\pmod{n}$ it would yield the same result without having to sum over the subset R, with this recurrence we are progressively adding to the subsets our new values x_i on each iteration.

   Lastly we are missing one subset, we have all the subsets that not include x_i from the first part, and all the subsets that include x_i and one or more x_j in S with $j < i$. But we have not considered the subset $R = \{x_i\}$ and that is what the last part of the recurrence do, $\{x_i \pmod{n}\}$, the sum of that set is just x_i so we have include $x_i \pmod{n}$ into our set $V(i)$.

2. Design an algorithm that, given a set S of n nonnegative integers, determines whether 0 is feasible with respect to S. The algorithm outpus "Yes" or "No". Your algorithm should run in $O(n^2)$ time. Your pseudocode can use set options such as $\cup$ as in the recurrence above.

```
Algorithm Subset-Sum-mod-n(S = {x_1, x_2, ..., x_n})
    // Returns "Yes" if for some subset R of S, sum_{x∈R} x = 0 (mod n)
    Create an array Soln[0..n]
    Set Soln[0] = ∅
    for i = 1 to n:
        Soln[i] = {S[i] mod n}
        for j = 1 to |Soln[i − 1]|:
            v = Soln[i − 1][j]
            Soln[i] = Soln[i] ∪ {v, v + s[i] (mod n)}
```

3. Suppose that indeed there is a subset R of S such that $\sum_{x \in R} x = 0 \pmod{n}$. Write an algorithm that finds such a subset R. You can assume that an array Soln[0..n] has already been pre-computed, where Soln[i] stores the set $V(i)$ defined in part 1, and your algorithm can use the array Soln. Your algorithm should run in time $O(n^2)$.

2 NP True or False

Let X and X' be decision problems, where both problems have "Yes" instances and "No" instances. State whether you think each of the following statements must be true, must be false, or is an open question. Justify your answer.

1. **Statement:** If $X \leq_p X'$ and X is *not* in NP, then X' is not in NP.

   True, if X' is at least as hard as X which is not in NP and also not in P, then it must be that X' has no solution neither in P nor NP. If it was the case that there was a solution for X' that could be checked in polynomial then X would also have a solution that could be checked in polynomial time, but it is not the case since we know that X is not in NP. Hence there is no way to check a solution for X' on polynomial time, therefore X' is not in NP.

2. **Statement:** If $X \leq_p X'$, X is in P and X' is in NP, then X' must be in P.

   Open question, if I can confirm or deny this, then I would win $1 million US. By proving this we would prove that P = NP.

3. Let $\overline{X}$ be the *complement* of problem X. That is, instance I of X is a No-instance of $\overline{X}$ if and only if I is a Yes-instance of X. For example, the problem $\overline{\text{SAT}}$ is the set of *unsatisfiable* 3-Sat Boolean formulas—formulas for which there is no satisfying truth assignment. Define $\overline{X'}$ similarly in terms of X'. The statement then is:

   Statement: If $X \leq_p X'$ then $\overline{X} \leq_p \overline{X}'$.

   False, even if we know that $X \leq_p X'$ we can't confirm that proving a not instance is reducible in the same way. So even though $\overline{X}$ has the same context as X they can have complexities, sometimes proving things are no-instances is faster than proving solutions are a yes-instances and vice versa.

3 Common superstrings

A *common superstring* of the strings $s_1, s_2, \ldots, s_n$ is a string T such that each s_i is a substring of $T, 1 \leq i \leq n$. (Note that the letters of the substring must appear consecutively in T, unlike our definition of a subsequence.) For example, if the strings are $s_1 = 101, s_2 = 0111$, and $s_3 = 010$ then the strings 010111 and 0111010 are both common superstrings of s_1, s_2, and s_3.

An instance of the *Shortest Common Superstring* problem is a set of strings $\{s_1, s_2, \ldots, s_n\}$ and a number K. The problem is to determine whether there is a common superstring of $s_1, s_2, \ldots, s_n$ that has length at most k. Assume that none of the strings s_i is a substring of another string s_j (otherwise s_i can be removed from the list of strings without changing the length of the shortest common superstring). This problem has been important in the context of developing algorithms for genome sequence assembly.

An instance of the *Traveling Salesperson Problem* (TSP) is a set $\{C_1, C_2, \ldots, C_n\}$ of cities, nonnegative integer costs c_{ij} of traveling from city C_i to city C_j (or from C_j to C_i; the cost is the same either way), and a number K. The problem is to determine whether there is a tour of the cities of cost at most K. A *tour* is a path that starts at some city, say C_1, travels through every other city exactly once, and returns to the starting city C_1. The tour cannot go through a city more than once (except for C_1 which appears both at the start and at the end of the tour).

1. Describe a polynomial time reduction from the Shortest Common Superstring problem to the traveling salesperson problem (TSP).

   We can set every string s_i to be a "city" c_i, $1 \leq i \leq n$. Then we create a n c_{ij} costs, each of this costs will be the minimum number of characters needed to create a super string of the strings $\{s_i, s_j\}$, i.e. for the $s_i = 010$, $s_i = 0111$ the cost c_{ij} would be 2, because we can use the 0 on both on the left side of the s_j and just add the 2 remaining characters 01 on s_i.

   We can create the each city c_i in linear time, just by iterating over the strings. After the creation of our costs, would take us n^2 time because we need to calculate a $n-1$ costs for each string s_i.

2. Is the Shortest Common Superstring problem in NP? Justify your answer.

   Yes, we reduced Shortest Common Superstring in polynomial time to Traveling Salesperson Problem, and we know that TSP can be verified in polynomial time. Which means that we can verify SCS in polynomial time by using TSP to verify it. Therefore SCS is in NP.

3. Does the reduction of part 1, plus your answer to part 2, imply that the Shortest Common Superstring problem is NP-complete? Why or why not?

   No, even we know for our answer in part 2 that SCS is in NP, we don't have enough facts to prove that SCS is in NP-Complete. What we have proved is that TSP is at least as hard as SCS, but to prove that SCS we need to prove that SCS is at least as hard as any NP-Complete problem, we could achieve this by reversing the direction of the reduction reducing TSP to SCS, then we will know that SCS is at least as hard as TSP which is an NP-Complete problem, hence SCS must be an NP-Complete.

4 Maximum Cuts (from Coulter Beeson)

In this problem you'll define and show that the MaxCut problem is NP-complete, using a reduction from a known NP-Complete problem, Partition. We'll first define Partition, which is already familiar to you, and then move on to MaxCut.

Partition is another name for the decision problem we saw in class, where thieves try to evenly share their loot. An instance of this problem is a set $I = \{v_1, v_2 ... v_n\}$ of values for n items. I is a Yes-instance if the items can be fairly split, that is, if we can partition the set $[1..n]$ into two nonempty subsets S and $\overline{S}$ such that $\sum_{i \in S} v_i = \sum_{j \in \overline{S}} v_j$.

A **cut** in an undirected graph $G = ([1..n], E)$ divides the set of nodes $[1..n]$ into two non-empty disjoint subsets. That is, a cut is a pair $(S, \overline{S})$ where $\overline{S} = V - S$ and $0 < |S| < n$. If weight $w(i, j)$ is associated with each edge (i, j) of the graph, then the weight of the cut $(S, \overline{S})$ is the sum of the weights of edges that cross the cut. That is,

$$w(S, \overline{S}) = \sum_{i \in S, j \in \overline{S}} w(i, j).$$

We would like to show that there is there is unlikely to be an efficient algorithm that takes as input a graph G and weight function w, and finds a cut of maximum weight. However NP-Completeness is only defined for decision problems.

1. Write down a decision version of the problem, which we'll call MaxCut. What is an instance I of MaxCut? Which are the Yes-instances of MaxCut?

   The decision would be, is there a cut with a weight of at least k. Then an instance I would be the graph $G = ([1..n], E)$, a set of weights W for each edge in G, and a k that would represent the boundary to determine if I is a yes-instance or a no-instance.

2. Show that MaxCut $\in$ NP.

   Given a certificate C which is a set of edges in the MaxCut for a yes-instance I, we can verify that is truthly a yes-instance by:

   (a) Verifying that every edge (i, j) in C is in MaxCut, and there are no duplicates on C

   (b) Sum each $w(i, j)$ for every edge (i, j) in C. If this sum is at least k then I is truthly a yes-instance

   This can be done in polynomial time, because the edges in C are upper bounded by n^2, because there can't be duplicates on C. Hence MaxCut is in NP.

3. Here you'll reason about a useful building block for a reduction from Partition to MaxCut. Let $I = \{v_1, v_2, \ldots, v_n\}$ be an instance of Partition, with $\sum_{i=1}^{n} v_i = 2V$. From I we'll build a graph $G = ([1..n], E)$ where E has an undirected edge between every pair of nodes (i, j), with weight $w(i, j) = v_i v_j$. Show that if $(S, \overline{S})$ is a partition of $[1..n]$ such that $\sum_{i \in S} v_i = V$, then in G, the weight of the cut (S, S') is V^2.

   Consider an I with $n = 4$ with elements $\{v_1, v_2, v_3, v_4\}$, arbitrary assuming that a subset S has the elements $\{v_1, v_2\}$, therefore $\overline{S} = \{v_3, v_4\}$. Since we are setting the weights $w(i, j)$ to be a multiplication between v_i and v_j on G, we will have that the sum of the weights in the cut is, $v_1v_3 + v_1v_4 + v_2v_3 + v_2v_4$, we can think about this as a distribution of two factors, the sets $S, \overline{S}$. So we end up with $(v_1 + v_2) * (v_3 + v_4)$, which still is equivalent to the weight of the cut.

   But we can go further with this, we know that $v_1 + v_2 = \sum_{i \in S}^{n} v_i$ and $v_3 + v_4 = \sum_{i \in \overline{S}}^{n} v_i$, so we can reduce the weight of the cut to $(\sum_{i \in S}^{n} v_i) * (\sum_{i \in \overline{S}}^{n} v_i)$, furthermore we are given that $\sum_{i \in S} v_i = V$ which yields, weight of cut $= V * (\sum_{i \in \overline{S}}^{n} v_i)$. Lastly we know that the sum of all elements in I add up to $2V$, since $\overline{S}$ is the complement of S, $2V = V + \sum_{i \in \overline{S}}^{n} v_i$, therefore $\sum_{i \in \overline{S}}^{n} v_i = V$ which give us the weight of the cut $= V * V = V^2$.

4. Show that if $(S, \overline{S})$ is a partition of $[1..n]$ such that $\sum_{i \in S} v_i \neq V$, then in the graph G of part 3, the weight of the cut $(S, \overline{S})$ is less than V^2. *Hint:* use the fact that if $x + y = 2V$ then xy attains its unique maximum when $x = y = V$.

   Lets consider a similar scenario like the above one, $S = \{v_1, v_2\}$ and $\overline{S} = \{v_3, v_4\}$. Now we are given that $\sum_{i \in S} v_i \neq V$, so lets pick an arbitrary number c, $c > 0$ to be the difference between $\sum_{i \in S} v_i$ and V, so we have $\sum_{i \in S} v_i = V - c$ and $\sum_{i \in \overline{S}} v_i = V + c$ because the difference must be added to the complement for consistency.

   Given these expressions we can follow the procedure of the previous question to arrive to the conclusion that the weight of the cut $= (V - c) * (V + c)$, given that the I has $\sum_{i=1}^{n} v_i = 2V$ and the fact that if $x + y = 2V$, the unique maximum is when $x = y = V$. Then the weight of the cut $(V - c) * (V + c) < V^2$. Because $(V - c) \neq (V + c) \neq V$ for $c > 0$, which has to be greater than 0 to meet the initial assumtion $\sum_{i \in S} v_i \neq V$.

5. Describe an efficient reduction that maps instances I of Partition to instances I' of MaxCut. You can describe your reduction in words or implement it in pseudocode.

   To reduce Partition to MaxCut, we can create a graph G containing a node a_i for each element in the Partition set $\{v_1, v_2, ...v_n\}$, $1 \leq i \leq n$. Then create an edge between all a_i and a_j nodes and set $w(i,j) = v_i v_j$, $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$. Lastly set $k = (\frac{1}{2}\sum_{i=1}^{n} v_i)^2$.

6. Explain why your reduction runs in polynomial time.

   This reduction takes linear time to create the nodes, quadratic time for creating the edges and weights between every x_i and x_j, and again linear time to calculate k. The total running time is n^2 which is polynomial time.

7. Show that if I is a Yes-instance of Partition then I' is a Yes-Instance of MaxCut. (Part 3 should be useful here, as well as your definition of part 1.)

   If I is a yes-instance then the sum of each v_i in I will be divisible by 2, because we need to separate I in two equal parts to be a yes-instance, this implies that $\sum_i^n v_i = 2V$ hence $\sum_{i \in S} v_i = \sum_{i \in \overline{S}} v_i = V$.

   From part 3 we know that if the sum of the subset S equal V then the weight of the maximum cut would be V^2, so now we just need to compare k and V^2, so we have the following comparison $V^2 = (\frac{1}{2}\sum_{i=1}^{n} v_i)^2$, which if we replace the sum of every v_i in I with $2V$ yields $V^2 = (\frac{1}{2}(2V))^2 \rightarrow V^2 = (V)^2$, which is true. Therefore I' is also a yes-instance.

8. Show that if I' is a Yes-instance of MaxCut then I is a Yes-Instance of Partition. (Part 4 should be useful here.)

   If I' is a yes-instance then the weight of the max cut $= (\frac{1}{2}\sum_{i=1}^{n} v_i)^2$, which can also be represented as MaxCut $= V^2$. We know from part 4 that there is no subset that for any subset S that add up to some number different than V, the weight of MaxCut would be strictly less than V^2. But in this case we know that it is $= V^2$, therefore I must also be a yes-instance.

9. In class we saw a dynamic programming algorithm for the Partition problem, with running time $\Theta(nV)$, where $2V = \sum_{i=1}^{n} v_i$. Why does this not imply that P = NP? (Think what is the size of a Partition instance; is the running time of the dynamic programming algorithm polynomial or exponential in the input size?)

   This do not imply that P = NP, the running time of that algorithm grows not only with the size of the input, but with the size of the data in the input. So we can have an instance I with $n = 5$ and another $\hat{I}$ with $n = 5$, but they would have different running times based on the value of V_I and $V_{\hat{I}}$.

10. **Bonus** Prove the hint given in part 4.

    To prove that xy reach their unique maximum if $x + y = 2V$ when $x = y = V$. Consider the following expression to prove, $(x - c)(y + c) < xy$ for some $c \neq 0$, $c < V$, if we distribute that expression we get:

$$(x - c)(y + c) < xy$$

$$xy + xc - yc - c^2 < xy$$

$$xc - yc - c^2 < 0$$

$$x - y - c < 0$$

$$x - c < y$$

By considering $x + y = 2V$

$$x - c < 2V - x$$

$$2x - c < 2V$$

We know for $x + y = 2V$, that x is at most $2V$, so by replacing it

$$2V - c < 2V$$

Which yields true, due to our constraints on c, $c \neq 0$, $c < V$. Therefore, our initial clause $(x - c)(y + c) < xy$ is also true. Hence xy reach the maximum when $x = y = V$, when $c = 0$.