CS 577: Introduction to Algorithms Fall 2022

Homework 9

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers reductions. **Problem 3 must be submitted for grading by 2:29pm on 11/29.** Note that you have two weeks for this assignment because of the second midterm exam and the Thanksgiving break. Please refer to the homework guidelines on Canvas for detailed instructions. Good luck and Happy Thanksgiving!

Warm-up problems

1. Consider the satisfiability problem for Boolean formulas. Show how to reduce search to decision in polynomial time.

2. Suppose you are given two sequences of nonnegative integers $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and two targets s and t. You want to decide whether there exists $I \subseteq [n]$ such that $\Sigma_{i\in I} a_i = s$ and $\Sigma_{i\in I} b_i = t$. Design a polynomial-time reduction from this problem to the subset sum problem.

   The subset sum problem takes as input a sequence of nonnegative integers $a'_1, a'_2, \ldots, a'_{n'}$ and a target t' and determines whether there exists $I' \subseteq [n']$ such that $\Sigma_{i\in I'} a'_i = t'$.

Regular problems

3. [Graded] You are nearing graduation and want to finalize the courses you need to take. You've already gotten all the core courses out of the way, so you only need to take T more credits out of the remaining n courses available. Each course $i \in [n]$ uses c_i credits. Also, taking more credits will increase your tuition, so you want to pick a course schedule that uses exactly T credits. Furthermore, you'd like to have time to go on adventures during your final semesters, so you want to pick a schedule with as few courses as possible. MyUW just released a new feature that given a list of courses and their credits, an integer t, and an integer m, determines if there is some subset of at most m of these courses so that the total sum of their credits is exactly t.

   (a) Design a polynomial-time algorithm that outputs a course schedule with total credit equal to T that uses as few courses as possible. You may assume you can use MyUW's new feature in constant time. In other words, you need to design a polynomial-time reduction from the problem of finding courses to the problem encoded in MyUW's new feature.

   (b) Notice that MyUW's new feature is really solving a variant of the subset sum problem (see problem 2). Namely, given a list of integers, $a_1, a_2, \ldots, a_n$, an integer, T, and an integer, m, determine if there is a subset $I \subseteq [n]$ with $|I| \le m$ such that $\Sigma_{i\in I} a_i = T$. Show that this variation reduces in polynomial time to the classical subset sum problem.

4. You're an avid bicyclist who just moved to Madison. You plan to go on a long bike ride every weekend in an effort to explore the city. You've picked up a Madison cycling guide that has

a list of the best bike routes. Each route includes the names of the landmarks it passes by (and each route only passes by a given landmark once). You are excited to explore Madison, so you refuse to take a weekend off of biking until you have passed by each landmark at least once. Your friend wants to visit you in w weeks, and she has planned a backpacking trip to take you on (which would preclude you from biking that weekend). She wants to know if you will be finished biking past all of the landmarks by the time she arrives. So, she has developed an algorithm to answer this question. Formally, her algorithm takes as input the number k, as well as m bike routes which together pass by n landmarks. Bike route $i \in [m]$ passes by landmarks $L_i \subseteq [n]$. Her algorithm outputs yes if there is a set of k bike routes that together pass by all n landmarks, and it outputs no otherwise.

   (a) Your friend tells you that, yes, it is possible for you to visit all n landmarks using only w of those m bike routes. However, she doesn't tell you which bike routes to use or whether you can use fewer than w routes. Your first ride is tomorrow, so you need to plan your routes now, and you want to plan the fewest routes possible to cover all landmarks. Unfortunately, your friend is camping with limited cell service, so she can't explain how her algorithm works. However, you can text her a number k' and a set of m' bike routes that together pass by n' landmarks, and she will respond whether or not it is possible to visit all n' landmarks using only k' of those m' bike routes.

   Formulate your bike route planning problem as an optimization version of your friend's decision problem. Design a polynomial time reduction from the optimization problem to the decision problem.

   (b) After completing the backpacking trip with your friend, you want to get back into biking every weekend. You enjoyed the bike guidebook and will use those routes again. However, since you've already visited each landmark once, you only want to visit each landmark at most one more time. As you plan your next set of bike routes, you want to choose the maximum number of bike routes such that none of these bike routes visit the same landmark.

   Formally, you are given m bike routes, which together pass by n unique landmarks. Bike route $i \in [m]$ passes by landmarks $L_i \subseteq [n]$. Your goal is to return a set of bike routes $B \subseteq [m]$ such that none of the bike routes pass by the same landmark and $|B|$ is maximized. Show how to reduce this problem to Independent Set in polynomial time.

5. Consider the variant of the Traveling Salesperson optimization problem in which the tour does not need to end in the same city as it starts in. Show that this variant and the original one are equivalent under polynomial-time reductions.

   The standard Traveling Salesperson problem is as follows: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Challenge problem

6. Recall scheduling to minimize the maximum lateness. Suppose we want to minimize the *sum* of the latenesses instead of the *maximum* lateness. Give a polynomial-time reduction from the partition problem to this problem.

Programming problem

7. SPOJ problem The Courier (problem code COURIER).
   Hint: Whereas the trivial algorithm for the Traveling Salesperson Problem takes time $\Theta(n!)$, there is a dynamic programming algorithm that only takes time $O(n^2 \cdot 2^n)$.

CS 577: Introduction to Algorithms Fall 2022

Homework 9 Solutions to Warm-up Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

Consider the satisfiability problem for Boolean formulas. Show how to reduce search to decision in polynomial time.

Recall that the decision version of the satisfiability problem for boolean formulas is as follows: Given a boolean formula written as ANDs, ORs, NOTs, and parenthesis of variables $x_1, \ldots, x_n$, output 'yes' if there exists an assignment of `true` or `false` to each variable such that the formula evaluates to `true`, otherwise output 'no'. The corresponding search problem then asks for a satisfying assignment of each variable or an indication that no such satisfying assignment exists.

Reduction We can reduce search to decision using the following strategy. First, we ask if a satisfying assignment exists. If not, we output that no satisfying assignment exists and we are done. Otherwise, assign the value `true` to variable x_1 and create a new formula by replacing x_1 with `true` and simplifying. Then check if a satisfying assignment exists in this new formula. If yes, then x_1 can be assigned the value `true`. If not, then x_1 must have the value `false`, since a satisfying assignment of the original formula existed, and x_1 can only take the values `true` or `false`. Therefore we can replace x_1 with `false` in the original formula and simplify. In either case, we have found an assignment of x_1 and produced a new formula such that a satisfying assignment of the variables $x_2, \ldots, x_n$ in the new formula corresponds to a satisfying assignment in the original formula. Therefore, we can recurse on the new formula to determine a satisfying assignment of the remaining variables.

Note that this solution corresponds to the eager approach from class (we try to set a variable to `true` if we can do so). It produces the lexicographically last satisfying assignment $x_1x_2\ldots x_n$ (meaning that if you wrote the satisfying assignments as strings of length n made of "t"s and "f"s, any other satisfying assignment would come before ours in alphabetical order). Can you think of a reluctant approach for the problem that produces the lexicographically first satisfying assignment?

Correctness Clearly, if there is no satisfying assignment, we output "no" correctly. Otherwise, we know that there is a satisfying assignment. As discussed above, when we ground a variable x_i in a formula, if the new formula is satisfiable, then the original formula is satisfiable when x_i is set to the value we grounded it as. At each step of our recursion, we know that the original formula at that step is satisfiable, and we choose a grounding of x_i that results in the simplified formula being satisfiable (using our decision blackbox). So, by the end of our recursion we have grounded all variables $x_1 \ldots x_n$ in a way that satisfies the original formula.

Runtime Analysis We make polynomially-many ($O(n)$) calls to decision and do a polynomial-time amount of extra work (note that simplifying the formulas can be done in polynomial time). Therefore, the reduction runs in polynomial time.

Problem 2

Suppose you are given two sequences of nonnegative integers $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_n$ and two targets s and t. You want to decide whether there exists $I \subseteq [n]$ such that $\Sigma_{i \in I} a_i = s$ and $\Sigma_{i \in I} b_i = t$. Design a polynomial-time reduction from this problem to the subset sum problem.

Reduction We need to develop an input $a'_1, a'_2, \dots, a'_n$, t' to subset sum such that there exists a subset of a' values that sum to t' iff there exists a subset I of indices of a_i values such that these a_i values sum to s *and* the corresponding b_i values sum to t. To do this, we will create a'_i values that incorporate both a_i and b_i values and a target value t' that incorporates both s and t. We cannot simply sum the values together such that $a'_i = a_i + b_i$ and $t' = s + t$ because the a_i and b_i values can interfere with each other. In order to avoid this interference, we will multiply the a_i values and s by a large number. Specifically, our large number will be $N = \max[t, \sum_{i \in [n]} b_i] + 1$. This way, even if we add together all of the b_i values, they cannot interfere with the a_i or s values. Similarly, even if t is large, it cannot interfere with the a_i or s values. We will develop an input to subset sum of $a'_i = a_i N + b_i$ and $t' = sN + t$.

Intuitively, this process ensures that when we add together the subset sum, one of the subset sum problems occurs in the "larger" section of the number and the other occurs in the "smaller" section of the number. For intuition, you can imagine that if we had set N to a large enough power of 2 and converted the numbers to binary, the a_i and s values would be in the "left" section of the number, followed by a string of zeroes. In this way, even if the b_i values in the "right" section of the number were all added together, there would be no carries from the right section into the left section. This is not specific to the binary representation and could be done in decimal or your preferred number system. Note that we essentially did this in a base-N representation.

Correctness To prove the correctness of our reduction, we need to demonstrate that there exists a subset I of indices of a_i values such that these a_i values sum to s *and* the corresponding b values sum to t iff there exists a subset of a'_i values that sum to t'.

$\Rightarrow$ We first show the easier direction: if there exists a subset I of indices of a_i values such that these a_i values sum to s *and* the corresponding b values sum to t then there exists a subset of a'_i values that sum to t'. We know $\sum_{i \in I} a_i = s$ and $\sum_{i \in I} b_i = t$. This gives us the following equations

$$N \sum_{i \in I} a_i = Ns$$

$$\sum_{i \in I} [N a_i + b_i] = Ns + t$$

By our definitions of a'_i and t', this yields $\sum_{i \in I} a'_i = t'$.

$\Leftarrow$ The other direction we need to prove is that if there exists a subset of a'_i values that sum to t' then there exists a subset I of indices of a_i values such that these a_i values sum to s *and* the corresponding b_i values sum to t. By our definition of a'_i and t', we know,

$$N \sum_{i \in I} a_i + \sum_{i \in I} b_i = Ns + t \tag{1}$$

Taking this equation modulo N and realizing that $N > t$ and $N > \sum_{i=1}^{n} b_i \geq \sum_{i \in I} b_i$, we get,

$$\sum_{i \in I} b_i = t \tag{2}$$

Finally, we can subtract t from both sides of (1) and use the equality in (2).

$$\begin{aligned} \sum_{i \in I} N a_i + \sum_{i \in I} b_i - t &= Ns + t - t \\ \sum_{i \in I} N a_i &= Ns \\ \sum_{i \in I} a_i &= s \end{aligned} \tag{3}$$

Together, (1) and (3) prove this direction.

Runtime Analysis To develop our reduction, we compute $O(n)$ new input values $a'_1, a'_2, \dots a'_n$, and t'. Each of these computations uses a constant number of arithmetic operations, which can be computed in polynomial time. Thus, the reduction runs in polynomial time.

CS 577: Introduction to Algorithms Fall 2022

Homework 9 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 3

You need to develop a course schedule that uses exactly T credits and minimizes the number of courses you take. You will choose courses from the remaining n courses available. Each course $i \in [n]$ uses c_i credits. MyUW just released a new feature that given a list of courses and their credits, an integer t, and an integer m, determines if there is some subset of at most m of these courses so that the total sum of their credits is exactly t.

(a) Design a polynomial-time algorithm that outputs a course schedule with total credit equal to T that uses as few courses as possible. You may assume you can use MyUW's new feature in constant time. In other words, you need to design a polynomial-time reduction from the problem of finding courses to the problem encoded in MyUW's new feature.

(b) Notice that MyUW's new feature is really solving a variant of the subset sum problem. Namely, given a list of integers, $a_1, a_2, \dots, a_n$, an integer, T, and an integer, m, determine if there is a subset $I \subseteq [n]$ with $|I| \leq m$ such that $\Sigma_{i \in I} a_i = T$. Show that this variation reduces in polynomial time to the classical subset sum problem.

Part (a)

Reduction/Correctness As stated, the problem asks for an optimal solution, meaning it is the OptSol version of the problem. Instead of solving this problem directly (with a black box for MyUW's feature), we will first find the value of an optimal solution (OptVal). This will help guide the search for the optimal solution. For this problem, solving OptVal means computing m, the minimum number of courses needed so that we can still find a schedule with total credit amount T. We can compute m by finding the smallest number k for which there is still some subset of all the courses of size k whose total credit amount is T yet there is no subset of all the courses of size $k - 1$ whose total credit amount is T. In particular, we can do a binary search over all $0 \leq k \leq n$ and ask MyUW's feature if the above is true. We then set m to be the smallest such k. If this procedure fails to find an m, then no course schedule exists that has total credit amount equal to T, so we just output "No".

Next, we need to compute an actual set of m courses that has total credit amount T. For each $i \in [n]$, we need to decide if we should take course i, which is a c_i credit course. Notice that there is a m course schedule that uses i and has total credit sum T if and only if $[n] \setminus \{i\}$ has a $m - 1$ course schedule that has total credit sum equal to $T - c_i$. To show the direction $\Rightarrow$, if there is some m course schedule, I, that uses i and has total credit sum T, then $I \setminus \{i\}$ is a $m-1$ course schedule that has total credit sum equal to $T - c_i$ and does not use course i. To show the direction $\Leftarrow$, if $[n] \setminus \{i\}$ has a $m - 1$ course schedule, J, that has total credit sum equal to $T - c_i$, then we know course i is not used in this schedule by definition, so $J \cup \{i\}$ is a m course schedule that uses i and has total credit sum equal to $(T - c_i) + c_i = T$. This observation leads to an eager solution for the

problem: For each course, we check if it can be part of an optimal solution, and include it in the solution in case it can.

We now present this idea in more detail. Let $I = [n]$, for each $i \in [n]$, we ask MyUW's feature if there is a set of $m-1$ courses from $I \setminus \{i\}$ that has total credit $T - c_i$. If so, then we take up course i into our schedule, set $I \leftarrow I \setminus \{i\}$, $T \leftarrow T - c_i$, and $m \leftarrow m - 1$ and continue to consider the next course. Otherwise, we know we shouldn't place course i in our schedule, we leave I, T, and m as they are, and continue to consider the next course. We stop when m becomes zero . At the end of this procedure, $S \doteq [n] \setminus I$ will exactly be a set of m courses that have total credit sum equal to T. This is because we continually remove courses from I that are in a course schedule satisfying the desired properties. Lastly, as we know m is the fewest number of courses we could have possibly taken while still being able to achieve exactly T credits, we know that S is a minimum sized set of courses that has total credit equal to T. This procedure is summarized in the pseudocode below where MyUW's feature is represented by a function called MyUWF.

Algorithm 1 Computing a course schedule of size m having T credits

```
$I \leftarrow [n]$
if MyUWF$(I, T, m)$ = "No" then return "Not Possible"
$i \leftarrow 1$
while $m > 0$ do
    if MyUWF$(I \setminus \{i\}, T - c_i, m - 1)$ = "Yes" then
        $I \leftarrow I \setminus \{i\}, T \leftarrow T - c_i, m \leftarrow m - 1$
    $i \leftarrow i + 1$
return $[n] \setminus I$
```

Runtime Analysis We note that to compute m takes $O(n \log n)$ time because we use a simple binary search that iterates $O(\log n)$ times. At each iteration, we ask two simple queries to MyUW's feature, each of which takes $O(n)$ time to copy the input into MyUW's feature. Similarly, to construct the course schedule we need $O(n)$ time per iteration to update I and query MyUW's feature. We also need to do arithmetic in every iteration, which can be done in time polynomial in the bitlength of the numbers. Since we do $O(n)$ iterations in this loop as well, the total reduction runs in polynomial time.

Part (b)

Reduction/Correctness We are given access to an oracle for the subset sum decision problem and want to determine if there is some $I \subseteq [n]$ so that $\sum_{i \in I} c_i = T$ and $|I| \leq m$. Let's first consider the variation of this problem where we require $|I| = k$ for some given $0 \leq k \leq n$. Notice this variation is a special case of discussion problem 2, where we set $a_i = c_i$, $b_i = 1$ for all i while also setting $s = T$ and $t = k$.

We claim there is an $I \subseteq [n]$ so that $\sum_{i \in I} c_i = T$ and $|I| = k$ if and only if there is a $J \subseteq [n]$ so that $\sum_{j \in J} a_j = s$ and $\sum_{j \in J} b_j = t$. To show the direction $\Rightarrow$, if we have some $I \subseteq [n]$ satisfying $\sum_{i \in I} c_i = T$ and $|I| = k$, then $\sum_{i \in I} a_i = \sum_{i \in I} c_i = T = s$ and $\sum_{i \in I} b_i = |I| = k = t$. To show the direction $\Leftarrow$, suppose $J \subseteq [n]$ satisfies $\sum_{j \in J} a_j = s$ and $\sum_{j \in J} b_j = t$. Then, by definition of the b_i's, we have $|J| = \sum_{j \in J} b_j = t = k$, so $J \subseteq [n]$ satisfies $|J| = k$ and $\sum_{j \in J} c_j = \sum_{j \in J} a_j = s = T$.

Hence, this gives us a mapping reduction from the variation to the problem in discussion problem 2. Then, composing this reduction with the mapping reduction from discussion problem 2 gives us a mapping reduction from the variation to the subset sum decision problem.

Now, returning to our original problem where we just need $|I| \leq m$, we can compute the above reduction for each $0 \leq k \leq m$ and output "Yes" if and only if one of the subset sum oracle calls says "Yes".

Runtime Analysis Each individual reduction for a given k to the subset sum decision problem takes polynomial time because reducing to discussion problem 2 takes $O(n)$ time (creating the s, t, a_i and b_i variables), and reducing from discussion problem 2 to the subset sum decision problem takes polynomial time. We construct these individual reductions $m \leq n$ times. Hence, the entire procedure takes polynomial time and reduces our original problem to the subset sum decision problem.

Problem 4

Your friend has an algorithm that takes as input the number k, as well as m bike routes which together pass by n landmarks. Bike route $i \in [m]$ passes by landmarks $L_i \subseteq [n]$. Her algorithm outputs yes if there is a set of k bike routes that together pass by all n landmarks, and it outputs no otherwise.

(a) Formulate your bike route planning problem (selecting the fewest routes possible to cover all landmarks) as an optimization version of your friend's decision problem. Design a polynomial time reduction from the optimization problem to the decision problem.

(b) You are given m bike routes, which together pass by n unique landmarks. Bike route $i \in [m]$ passes by landmarks $L_i \subset [n]$. Your goal is to return a set of bike routes $B \subset [m]$ such that none of the bike routes pass by the same landmark and $|B|$ is maximized. Show how to reduce this problem to Independent Set in polynomial time.

Part (a)

Reduction The optimization problem (of the OptSol type) is as follows: the input consists of m bike routes which together pass by n landmarks. Moreover, bike route $i \in [m]$ passes by landmarks $L_i \subseteq [n]$. The output is a set of bike routes such that all n landmarks are covered and the set of bike routes is of minimal size.

Similarly to Problem 3 our first goal to solve this problem is to solve the corresponding OptVal version: find the smallest number k for which a set of bike routes of size k can cover all landmarks, but a set of bike routes of size $k-1$ cannot. Because our friend said that it is possible to visit all n landmarks using only w of the m bike routes, we will perform binary search over $0 \leq k \leq w$ and use the decision oracle to determine when we have found such a k. (Note that if we didn't already know that it was possible to visit all n landmarks in w weeks, we could use a binary search over $0 \leq k \leq m$, and if we found no value of k such that k bike routes can cover all n landmarks, we would simply return "not possible.")

Note that once we know the smallest value k, we are left with the following search problem: return a set of bike routes $B \subseteq [m]$ of size $|B| = k$ such that all landmarks are covered. We will reduce this to decision using an eager approach. We first check if we can include route 1 in order to cover all landmarks using k bike routes. In order to do this, remove route 1 from the set of routes and remove every landmark covered by route 1 from the set of landmarks we are trying to cover. Then ask the decision oracle if you can cover the remaining landmarks using the remaining routes with $|B| = k-1$. If you can, then include route 1 in your set of routes, permanently delete the landmarks that were covered by route 1 from the set of landmarks we are trying to cover, and set $k = k-1$. Otherwise, delete route 1 and put back the landmarks that were removed. Then we can move on to check if we can include route 2 in the same manner. We will repeat this until we have included k routes in our set B. See the pseudocode (2) for a formal description.

Correctness First, we note that using binary search we find the smallest number k for which a set of bike routes of size k can cover all landmarks, but a set of bike routes of size $k-1$ cannot. If covering all landmarks is not possible with $k-1$ bike routes, it certainly won't be possible with less. So, we can see that binary search finds the smallest number of bike routes possible to cover

Algorithm 2 Computing a set of k bike routes that covers all landmarks $[n]$. The bike routes come from the set $i \in [m]$ where bike route i covers landmarks $L_i \subseteq [n]$. BikeDec is our decision oracle that takes as input a set of bike routes, a set of landmarks to cover, and a number of bike routes k.

```
$B \leftarrow \emptyset$;                                        ▷ bike routes used
$M \leftarrow [m]$;                                              ▷ bike routes left
$N \leftarrow [n]$;                                              ▷ landmarks left
if BikeDec$(M, N, k)$ = "No" then return "not possible"
$i \leftarrow 1$;
while $k > 0$ do
    if BikeDec$(M \setminus \{i\}, N \setminus L_i, k-1)$ = "Yes" then
        $B \leftarrow B \cup \{i\}$;
        $M \leftarrow M \setminus \{i\}, N \leftarrow N \setminus L_i, k \leftarrow k-1$
    $i \leftarrow i+1$;
return $B$
```

all landmarks.

To demonstrate the correctness of our solution to the search problem, we prove that there is a k-size set of bike routes chosen from set M that uses route i and covers all landmarks in the set N iff $M \setminus \{i\}$ has a $(k-1)$-size set of bike routes that covers all landmarks $N \setminus L_i$. We begin by showing the direction $\Rightarrow$, if there is some k-size set of bike routes B that uses i and covers all landmarks N, then $B \setminus \{i\}$ is a $(k-1)$-size set of bike routes that covers all landmarks $N \setminus L_i$. To show the direction $\Leftarrow$, if $M \setminus \{i\}$ has a $(k-1)$-size set of bike routes, B', that covers all landmarks $N \setminus L_i$, then we know route i is not used in this schedule by definition, so $B' \cup \{i\}$ is a k-size set of bike routes that covers all landmarks $(N \setminus L_i) \cup L_i = N$. Thus, by the end of the algorithm, we have built a set of bike routes B of size k that together covers all landmarks.

Runtime Analysis Finding the smallest value k takes polynomial time because binary search takes $O(\log n)$ iterations, and each iteration makes two calls to the decision oracle, each of which requires copying the input to the oracle in linear time.

To construct the solution set of bike routes, we perform $O(n)$ iterations, checking if we should include each bike route i. At each iteration, we make one call to the oracle. Building the input for the oracle call requires deleting $O(n)$ landmarks, removing one bike route, and decrementing k. Making these changes and copying the input values into the oracle takes polynomial time. So, constructing the solution set of bike routes takes polynomial time. Thus, we have developed a polynomial time reduction from optimization to decision.

Part (b)

Reduction Consider the similarities between this bike route problem and independents set (IS). Our goal is to choose the maximum number of bike routes while IS wants to choose the maximum number of vertices. Our constraint is that we cannot choose two bike routes that cover the same landmark while IS has the constraint of not choosing two vertices that cover the same edge. This leads to a natural mapping of bike routes to nodes and landmarks to edges. We begin with m nodes, one per bike route, and we place edges between every pair of nodes corresponding to routes

that share landmarks. This way, if we choose node i (i.e. route i that passes landmark j), we cannot choose any node that shares an edge with i (i.e. any route that also passes landmark j).

Our reduction is as follows: create a graph G with one node per bike route. Place an edge between any node i and j iff $L_i \cap L_j \neq \emptyset$. Use IS to determine the set of nodes S forming a maximum independent set on G. Return $B = S$ as the set of bike routes with no overlapping landmarks such that $|B|$ is maximized.

Correctness To prove the correctness, we need to show that S is an independent set on G iff B is a set of bike routes with no overlapping landmarks. If S is an independent set on G, then this means that no two nodes were chosen that share an edge. By our construction of G, this means that no two bike routes in B share a landmark. If B has no overlapping landmarks, then by our construction of G we know that $S = B$ never has two endpoints of the same edge. This means that S is an independent set. Now, since $S = B$ and S is an independent set iff B is a set of bike routes with no overlapping landmarks, we know S is a maximum independent set iff B is a maximal set of bike routes with no overlapping landmarks.

Runtime Analysis Graph G uses m nodes and $O(m^2)$ edges (because in the worst case, every pair of bike routes could have overlapping landmarks). Finding these edges requires looking at all $O(m^2)$ pairs of bike routes i, j and computing the intersection of L_i and L_j, which can be done in polynomial time. So, building graph G takes polynomial time. Returning the solution $B = S$ takes $O(m)$ time to copy the solution set. In all, the reduction from the bike route problem to IS is polynomial-time.

Problem 5

Consider the variant of the Traveling Salesperson optimization problem in which the tour does not need to end in the same city as it starts in. Show that this variant and the original one are equivalent under polynomial-time reductions.

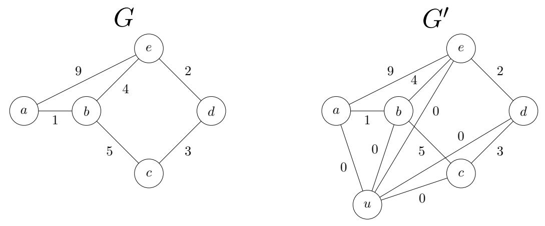
The standard Traveling Salesperson problem is as follows: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Call the version of the Traveling Salesman Problem (TSP) where the tour does not need to end in the same city as it starts Path-TSP. In the same way, call the original formulation where the tour does need to start and end in the same city Circuit-TSP. Denote the optimization version of these problems Path-TSP-optimization and Circuit-TSP-optimization, respectively. Recall that for the optimization version of TSP, we must find the value AND path of the shortest route that visits each city exactly once (meaning it is of the OptSol type). Furthermore, to show an optimization problem A polynomial-time mapping-reduces to an optimization problem B, we must show that for any instance x_A of problem A, we can map it to an instance x_B of B such that we can construct the optimal solution to x_A given the optimal solution to x_B. More formally, we must find a function g that maps an instance x_A of problem A to an "equivalent" instance x_B of problem B, and another function h that takes an instance x_A of A and a solution y_B of x_B and maps to a solution y_A of x_A. The solution y_A of x_A should be optimal, and the mapping must be computable in polynomial time. Finally, optimization problems A and B are equivalent under polynomial-time mapping reductions if A polynomial-time mapping-reduces to B and B polynomial-time mapping-reduces to A. We will start by showing Path-TSP-optimization polynomial-time mapping-reduces to Circuit-TSP-optimization.

Path-TSP-optimization $\leq^P$ Circuit-TSP-optimization

Given an instance of Path-TSP with graph G, construct a copy of graph G, G'. Then, add an additional vertex u and edges of weight 0 from u to all vertices in G'. Thus we have $g(G) = G'$. Now, run Circuit-TSP on the new graph G', which returns a TSP circuit C of G' of minimal cost, or states that no such circuit exists. If no circuit exists, return that no TSP path exists in G; otherwise, construct a path P in G by removing vertex u from C and its corresponding edges. Therefore, $h(G, C) = P$. We claim that P is a minimal-cost path in G or no path exists in G that visits each vertex exactly once. We will show this by showing an equivalence of TSP paths in G and TSP circuits in G'.

Figure 1: Path-TSP-optimization $\leq^P$ Circuit-TSP-optimization Example



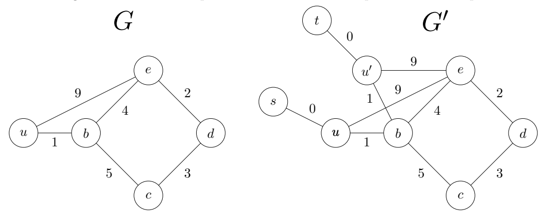
Note that in the method used to construct P, we removed edges of cost 0 from C, so P has the same cost as the circuit C. Further note that since C visits every vertex of G' exactly once, and P visits every vertex C visits other than u, which does not exist in G, P visits every vertex of G exactly once. Finally, since this construction did not depend on C being minimal, this shows for every TSP circuit in G' we can construct a TSP path in G of equivalent cost.

Now, in the other direction, note that for any TSP path X in G, starting at some vertex s and ending at some vertex t, we can construct a TSP circuit Y in G' of equivalent cost by adding the edges (t, u) and (u, s), both of cost 0, to Y. This equivalence shows that if no TSP circuit exists in G', then no TSP path exists in G, otherwise, the minimal cost TSP circuit in G' is the same cost as the minimal cost TSP path in G, completing the reduction. Finally, since constructing the graph G' and removing u from the minimal cost circuit C in G' can be done in linear time, the reduction can be computed in polynomial time.

Circuit-TSP-optimization $\leq^P$ Path-TSP-optimization

Given an instance of Circuit-TSP with graph G, construct a new graph G' by performing the following steps. First, let $G' = G$. Then, arbitrarily choose a vertex u of G' and duplicate it, copying any edges incident to u to the duplicated vertex u'. Now, create auxiliary vertices s and t and connect s to u and t to u', both with edges of cost 0. So we have $g(G) = G'$. Run Path-TSP on the new graph G', which returns a TSP path P of G' of minimal cost or states that no such path exists. If no path exists, return that no TSP circuit exists in G; otherwise, note that since auxiliary vertices s and t have degree 1, if any TSP path exists in G', it must visit s and t first and last. Therefore, we can construct a circuit C in G from P by removing vertices s, u', t and their corresponding edges and adding the edge (z, u), where z is the vertex immediately preceding u' in P. So $h(G, P) = C$. We claim that C is a minimal-cost circuit in G or no circuit exists in G that visits each vertex exactly once. Again, we show this by showing an equivalence of TSP circuits in G and TSP paths in G'.

Figure 2: Circuit-TSP-optimization $\leq^P$ Path-TSP-optimization Example



Since the edges (s, u) and (t, u'), removed from P to construct C, are of cost 0, and the edge (z, u) is the same cost as edge (z, u'), it follows that C has the same cost as P. Furthermore, since P visits every vertex in G', and C visits every vertex that P visits except for u', t, and s, it also follows that C visits every vertex in G. Then, again, since the construction of C did not depend on the minimality of P, this shows that for any TSP path in G' we can construct a TSP circuit in G of equivalent cost.

In the other direction, consider a TSP circuit X in G. Construct a TSP path Y in G' by removing the edge (z, u), where z is a neighbor of u in X, and adding edges (z, u'), (u', t), and (u, s). Again, since (u', t) and (u, s) are of cost 0, and (z, u') is the same cost as (z, u), Y has the same cost as X. Therefore if no TSP path exists in G', then no TSP circuit exists in G, otherwise, the minimal TSP path in G' is the same cost as the minimal TSP circuit in G. Since creating the graph G' and the circuit C can be done in linear time, the reduction can be computed in polynomial time and so we are done.

COMP SCI 577 Homework 09 Problem 3

Reductions

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

29 November 2022

Question (a)

Algorithm

We first reference the new feature of MyUW described in the write-out of this problem as Algorithm 1.

Then we define the main subroutine of question (a) as `Schedule` with an auxiliary subroutine as `MinNumTakenCourses` in Algorithm 2. We define the \ symbol to remove only one course from the set of courses even if there are multiple courses with same number of credits, without changing the indices of elements. Similar definition applies to the $\cup$ symbol. For the `MinNumTakenCourses` subroutine, we check for every $m \in [n]$ increasingly and return the first one found to satisfy the condition of `HasCourses`, and return an error if there is no one found. For the `Schedule` subroutine, we check for every course $c_i \in [n]$ increasingly by the index i to see if there is a subset of exactly $m_c - 1$ courses which fit the condition, to put into the result set s, and we remove c_i from the course list regardless if the check is passed.

Proof

Correctness

Claim: `MinNumTakenCourses` is correct

Proof: Denote the correct number of minimum number of courses to take be m'.

As there are n courses available, $m' \in [n]$, if there is no such m', then all subsets of c with size in $[n]$ cannot satisfy the condition, and we return an error in this case. □

```
1 Function HasCourses($n, c, t, m$):
      Input: $n$ is number of courses; $c$ is a list such that $c_i$ is the number of credits of the
             $i^{\text{th}}$ course with $i \in [n]$; $t$ is the exact sum of credits; $m$ is the most number of
             courses
      Output: True if there is some subset of at most $m$ of these courses so that the
              total sum of their credits is exactly $t$, False otherwise
```
Algorithm 1: MyUW's new feature (No detail)

If there is such m', we attempt all possible $m \in [n]$ which are in the range of m' to be drawn from. If at most m_1 courses satisfy the condition, then at most $m_1 + 1$ courses satisfy the condition by that we do not select the extra course added to the discussion so that at most $m_2 \geq m_1$ courses also satisfy the condition inductively. If we find the smallest m_1, then all $m_2 \in [m_1, n]$ satisfy the condition, that the solutions of this check are monotone. Thus, we return the smallest m which satisfies the condition which must be followed by the logic of m'. ■

Claim: `Schedule` is correct

Proof: If there is no solution of `Schedule`, then we must receive an error at line 8, and the program would not proceed, which is expected. Otherwise, there is a solution with some minimal m received from the proved `MinNumTakenCourses` subroutine, then we start an induction to discuss the for loop.

For the base case $i = 1$, c_1 is in some solution with exactly m courses (i.e., at most m satisfies, but does not satisfy at most $m-1$) and T credits in total if and only if that, in the set $\{c_2, \ldots, c_n\}$ we can find exactly $m-1$ courses and $T - c_1$ credits in total, and we could add c_1 to the solution set s. So if we cannot find such a condition satisfied, then c_1 is not in any solution, and we can safely remove c_1 from the course list defined by n and c without affecting any further result.

For the inductive case, suppose the claim holds for $i \in [k-1]$, now we want to prove that for $i = k$. Regarding T and n as original ones before entering this loop and T_c, n_c, and c_c be in the current repetition of this loop. Suppose at this time we already have $n_s \in [0, m]$ selected courses in s, then the T_c we are finding this loop is $T - \sum_{i=1}^{n_s} s_i$. From the induction hypothesis we also have $\forall i \in [k-1]$, $c_i \notin c_c$ that $c_c = \cup_{i \in [k,n]} \{c_i\}$, $n_c = n - (k-1)$, $m_c = m - n_s$. The problem now becomes that if c_k is in some solution with exactly m_c courses and T_c credits. This condition, similarly, could be equal to that if in the set $\{c_{k+1}, \ldots, c_n\}$ we can find exactly $m_c - 1$ courses and $T_c - c_k$ courses in total. If not, we just remove c_k from the course list. Thus, the invariants of c_c, n_c, m_c, and T_c still hold.

Therefore, as the invariants hold and the m is a guaranteed number of courses to take, we

```
1  Function MinNumTakenCourses($n, c, t$):
       Input: $n$ is the total number of courses; $c$ is a list such that $c_i$ is the number of
              credits of the $i^{\text{th}}$ course with $i \in [n]$; $t$ is the exact sum of credits
       Output: $m$ is the smallest number of courses with total credit equal to $T$
2      foreach $m \in [n]$ do
3          if HasCourses($n, c, t, m$) is True then
4              return $m$
5      return Error: Not possible
6  Function Schedule($n, c, T$):
       Input: $n$ is number of courses; $c$ is a list such that $c_i$ is the number of credits of the
              $i^{\text{th}}$ course with $i \in [n]$; $T$ is the exact sum of credits
       Output: $s$ is a course schedule with total credit equal to $T$ that uses as few courses
               as possible
7      $s \leftarrow \emptyset$;
8      $m \leftarrow$ MinNumTakenCourses($n, c, t$);
9      foreach $i \in [n]$ do
10         $n \leftarrow n - 1, c \leftarrow c \setminus \{c_i\}, T_c \leftarrow T - c_i, m_c \leftarrow m - 1$;
11         if HasCourses($n, c, T_c, m_c$) is True and HasCourses($n, c, T_c, m_c - 1$) is False
            then
12             $s \leftarrow s \cup \{c_i\}$;
13             $T \leftarrow T_c, m \leftarrow m_c$;
14     return $s$
```
Algorithm 2: Question (a): Schedule

must reach a full solution when $T_c = 0$ and $m_c = 0$ without any further update possible, and at any such point, s contains a set of m courses with exactly T credits, and we return the s. ■

Termination

The subroutine must terminate as there is no recursive call. ■

Complexity

The complexity of `MinNumTakenCourses` is $O(n)$ as there is only a loop through $[n]$ and the call to `HasCourses` is linear. The complexity of `Schedule` (i.e., the main subroutine) is $O(n)$, as we already have `MinNumTakenCourses` to be $O(n)$, and then we have only one extra loop through $[n]$ and the call to `HasCourses` is linear. By the definition at Page 11 and Page 12 on Handout on 15 November 2022, `Schedule` runs in polynomial time (i.e., `Schedule` $\leq^P$ `HasCourses`). ■

```
1 Function TwoSubsetSum($a, b, s, t$):
      Input: two sequences of nonnegative integers $a$ and $b$ and two targets $s$ and $t$
      Output: True if there exists $I \subset [n]$ such that $\sum_{i \in I} a_i = s$ and $\sum_{i \in I} b_i = t$, False
              otherwise
```
Algorithm 3: Problem 2: Two Subset Sum Problem

```
1 Function HasCourses($n, c, t, m$):
      Input: $n$ is the total number of courses; $c$ is a list such that $c_i$ is the number of
             credits of the $i^{\text{th}}$ course with $i \in [n]$; $t$ is the exact sum of credits; $m$ is the
             most number of courses
      Output: True if there is some subset of at most $m$ of these courses so that the
              total sum of their credits is exactly $t$, False otherwise
2     $a \leftarrow \underbrace{\{1, 1, \ldots, 1\}}_{n \text{ 1's}}$, $b \leftarrow c$;
3     foreach $s \in [0, m]$ do
4         if TwoSubsetSum($a$, $b$, $s$, $t$) is True then
5             return True
6     return False
```
Algorithm 4: Question (b): MyUW's new feature (Reduction)

Question (b)

Algorithm

We reference the subroutine in Problem 2 of the same Homework 09 defined as `TwoSubsetSum` as Algorithm 3.

For the main subroutine `HasCourses` as Algorithm 4 to implement, we reduce the problem to `TwoSubsetSum`, with one subset b be the c of courses/credits and a be the n 1's, if there is one $s \in [m]$ to satisfy $\sum \{a_{i_1}, a_{i_2}, \ldots, a_{i_s}\} = s$ with i be a function shuffling indices without duplication and the corresponding sum of b equals to T, we return `True`, otherwise we return `False`.

Proof

Correctness

From the question specification, we want to determine if there is a subset $I \subset [n]$ with $|I| \leq m$ such that $\sum_{i \in I} a_i = T$.

For our reduction, we can use `TwoSubsetSum` to determine if there is a subset $I \subset [n]$ such that $\sum_{i\in I} a_i = s$ and $\sum_{i\in I} b_i = T$, i.e., we are selecting exactly s courses with a sum of exactly T credits, that by the definition of a, $\sum_{i\in I} a_i = |I|$. As $0 \le |I| \le m$, so we just check in this interval $s \in [0, m]$ for `TwoSubsetSum`, to change one factor to determine to be $\sum_{i\in I} a_i \le m$. Therefore, this is a valid reduction. ■

Termination

The subroutine must terminate as there is no recursive call. ■

Complexity

From the proof of Problem 2 in HW09 Solution to Warm-up Problems, we already have that `TwoSubsetSum` $\le^P$ `SubsetSum`. Regarding `TwoSubsetSum` running in $O(n^c)$, there is only one outer loop through $[m]$ so that `HasCourses` runs in $O(n^c)\cdot O(m) = O(n^c m) = O(n^{c+1})$ as $m \le n$ by the problem constraint. Therefore, `HasCourses` runs in polynomial time (i.e., `HasCourses` $\le^P$ `TwoSubsetSum` $\le^P$ `SubsetSum`). ■