COMP3927 assignment 4

Andrew Li

May 22, 2023

1 Q1

1.a

We can establish a Cook reduction as follows: Given an instance of the optimisation problem, run an oracle for the decision problem for all values of $b \in [0, |V|]$ increasing from 0 until the decision problem accepts. We then return the value of b for which it accepts.

1.b

This reduction must terminate, as if the oracle does not halt for $b < |V|$ then it must halt for $b = |V|$, as the set V satisfies the condition trivially. Our reduction returns the lowest value of b for which the oracle for the decision problem accepts, and thus by definition the minimum value of b such that there exists a set B of size b such that for all $v \in V \backslash B$ there exists an edge (u, v) with $u \in B$. This solves our problem and thus the reduction is correct. Our reduction takes no more than $|V| + 1 = O(|V|)$ calls (the number of b values it iterates over is at most this value) and so it is a polynomial time reduction.

1.c

Here is a verifier for the decision problem: We take as input the graph (V, E), integer b and a certificate B. We first check the B describes a subset of V, by for each element of B searching V for a match (implementation-wise, we may represent the vertices as integers from 1 to $|V|$, so we would want to ensure all elements of B fall in this range) and that it is of size b. If it fails this check, reject.

We then for each $v \in V \backslash B$ search E for edges of the form (u, v). We then check for each of these edges (u, v) if $u \in B$. If we do not find an edge satisfying this property, then we reject. If we have iterated through all v without rejecting, we accept.

To show this is correct, we need to show that (V, E) and b is an accepted instance of the decision problem if and only if there exists a certificate B such that our verifier accepts (V, E), b, and B.

To show the forward direction, if (V, E) and b is an accepted instance, then there must be a subset $B \in V$ of size b such that $\forall v \in V \backslash B \ \exists u \in B \ \ (u, v) \in E$. If we provide B as certificate to our verifier along with the other input, as it is a valid subset of V of size b it passes the first check. The second check our verifier performs is equivalent to the above condition and so it accepts B alongside the other input.

To show the reverse direction, if there exists a certificate B which our verifier accepts along with the other input, then first this certificate must describe a valid subset of V of size b and second this subset B satisfies the condition $\forall v \in V \backslash B \ \exists u \in B \ \ (u, v) \in E$ which our verifier checks. Thus as this is the condition defining the decision problem, we have that (V, E) and b must be an accepted instance of the decision problem.

Finally, we show that this verifier is polytime. if we encode our vertices as integers from 1 to $|V|$, then checking whether all elements of B fall in this range, and confirming that B has size b takes $O(|B|)$ time together. In the second check, we iterate over $V \backslash B$, E, and then B to search for a match,

so this check takes $O(|V \backslash B|) \cdot O(|E|) \cdot O(|B|) = O(|V| * |E| * |B|)$ which is polynomial with regard to the input.
As there exists a polytime verifier for the decision problem, it is in NP.

1.d

We show a reduction from set cover to our problem.

Given an instance of set cover $U, \{S_1, S_2, ...S_n\}, k$ we construct an instance of our problem $(V, E), b$ where $b = k$ if $k \leq n$ and $b = n + |U| + 1$ otherwise (this simply is in place to make sure that if the set cover gets a k greater than n the constructed instance of our problem rejects) and the graph (V, E) is constructed as follows: For each element of U construct a node. For each element of the collection construct a node. Construct two edges between all distinct pairs of vertices S_i and S_j going both ways. Construct an edge (S_i, u) for all vertex pairs where $u \in U$ and $u \in S_i$.

First, we show that if $U, \{S_1, S_2, ...S_n\}, k$ is an accepted instance of the set cover decision problem then our construction $(V, E), b$ is an accepted instance of our problem. If our set cover instance is accepted, there exists a subset B, $|B| = k$, of $\{S_1, S_2, ...S_n\}$ such that $\forall u \in U \; \exists S_i \in B \; u \in S_i$. Then for all $v \in V \backslash B$, either $v \in \{S_1, S_2, ...S_n\}$ or $v \in U$. If $v \in \{S_1, S_2, ...S_n\}$ then as there are edges between all S_i and S_j we have edges from B to all such vertices v. If $v \in U$ then as $\forall v \in U \; \exists S_i \in B \; v \in S_i$ by construction there is an edge from the existing $S_i \in B$ satisfying this property to v. This for all $v \in V \backslash B$, there exists $u \in B$ so that $(u, v) \in E$. As B is of size $b = k$ (as $k \leq n$ in accepted instances), this shows the existence of a set of vertices as described in our decision problem and thus $(V, E), b$ is an accepted instance.

Second, we show that if $(V, E), b$ is an accepted instance of our problem generated by our reduction then $U, \{S_1, S_2, ...S_n\}, k$ is an accepted instance of the set cover decision problem. If our problem's instance is accepted, then there exists $B \in V$, $|B| = b$, such that for all $v \in V \backslash B$, there exists $u \in B$ so that $(u, v) \in E$. We now need to show that if there exists a $B \in V$, $|B| = b$, with this property, then there also exists a $B \in V$, $|B| = b$ with this property such that for all $u \in U$, $u \notin B$. Our reduction enforces the constraint that either $b = k \leq n$ or $b = n + |U| + 1 > |V|$ but in the latter case, the instance cannot be accepted as we would need a subset of V greater in cardinality than it. This contradiction then implies $b \leq n$.

Now, for $B \in V$, $|B| = b$ satisfying our property, we remove all elements $u \in U, u \in B$ and for each element we remove we add an arbitrary S_i not already in B to B. We always will be able to add such elements as a result of the fact that $b \leq n$ and thus at each step except the last the number of elements $S_i \in B$ will be less than n. Removing elements $u \in U$ does not affect the number of vertices covered by B as u by construction has no outgoing edges and thus does not cover any vertices. At each step B already covers all vertices in $V \backslash B$, so when we add S_i to B, all vertices in $V \setminus (B \cup \{S_i\})$ are still covered by B (vaguely speaking, the coverage won't decrease when we expand B). Having now established that we have a set $B \in V$, $|B| = b$ consisting only of elements of type S_i satisfying our conditions, we have that as all vertices are covered, for all $u \in U$, there exists $S_i \in B$ such that $(S_i, u) \in E$, which by construction implies $u \in S_i$. This is the definition of a set cover, and thus B is a set cover of size $k = b$, and thus $U, \{S_1, S_2, ...S_n\}, k$ is an accepted instance of the set cover decision problem.

Having shown that one is an accepted instance if and only if the other is, we conclude that our reduction is valid. Our reduction also runs in polytime; computing b takes constant time, and we construct $n + |U|$ vertices in $O(n + |U|)$ time (with an adjacency matrix we can do this in constant time). We then construct edges for all pairs (S_i, S_j) of which there are $O(n^2)$ so this takes $O(n^2)$ time as each edge can be constructed in constant time. We finally construct the edges of form (S_i, u) where $u \in S_i$. We iterate through all sets S_i and make an edge for each $u \in S_i$. This thus takes $O(|U|n)$ time as each such set has no more than $|U|$ elements. Adding these times together, it is clear our reduction is polytime.

As we can reduce an NP-complete problem to our problem, our problem is NP-hard. It is also in

NP, so it is NP-complete.

1.e

We can reduce the decision problem to the original optimisation problem - there exists $B \in V$ of size b satisfying the required condition if and only $b >= b_{min}$ and $b <= |V|$, where b_{min} is the minimum b returned by the optimisation problem. To see this, note how given a set B of size b_{min} satisfying the required condition, we can arbitrarily add vertices to B up to a total size of $|V|$ and as we saw earlier, this expanded set still covers all remaining vertices. This gives us the reverse direction, and to see the forward direction - if $b_{min} > b$ then this would contradict minimality and if $b > |V|$ this would imply a subset of V greater in cardinality than V. This if and only if relation supplies a clearly polytime reduction - call the optimisation problem and then check the inequalities hold - if yes, then accept, otherwise refuse. As we can reduce the decision problem, which is NP-complete to the original problem, the original problem is NP-hard. It is not NP-complete, as it is not in NP as it is not a decision problem and NP is a set of decision problems.

2

2.a

The input consists of a number of planets n, partition of planets into quadrants Q_i, distances δ_l, base-building costs c_l, and budget value D. The budget value occupies $log(D)$ bits, and each building cost c_l is $log(D)$ at most as $c_l \leq D$ for all l. Each distance occupies at most $log(\Delta)$ bits, the number of planets occupies $log(n)$ bits, and the partition of planets into quadrants occupies no more than $log(k^n) = nlog(k)$ bits as each planet can be sorted into at most k possible quadrants and thus there are no more than k^n possible states for the partition. The input size is then upper bounded by $n(log(D) + log(\Delta)) + nlog(k) + log(D) + log(n) = nlog(Dk\Delta) + log(D) + log(n) = O(nlog(Dk\Delta))$.

2.b

Given n planets, a partition $Q = \{Q_1, Q_2, ...Q_k\}$ of these planets in quadrants, a distance d_l and cost c_l for each planet l, a total budget D, and a maximum distance S does there exist a choice of one planet $l_i \in Q_i$ from each quadrants such that $\sum_i c_{l_i} \leq D$ and $\sum_i \delta_{l_i} \leq S$.

2.c

We provide a verifier for the decision problem. The verifier takes in the input for the decision problem alongside a certificate describing a choice of $l_i \in Q_i$ for each quadrant. Our verifier first checks that the input is valid - there are k items given and the i-th item describes a planet in Q_i. It then calculates $\sum_i c_{l_i}$ and $\sum_i \delta_{l_i}$ and checks that they are less than D and S respectively. If so, it accepts, and otherwise it rejects.

We wish to show that there exists a certificate such that the verifier accepts if and only if the problem instance is accepted. For the forward direction, if the verifier accepts some certificate then the certificate must describe a valid choice of l_i such that $\sum_i c_{l_i} \leq D$ and $\sum_i \delta_{l_i} \leq S$. By the definition of the decision problem, we must also have an accepted instance of it. For the reverse direction, if the input for the decision problem is accepted, then there exists a choice of one planet $l_i \in Q_i$ from each quadrants such that $\sum_i c_{l_i} \leq D$ and $\sum_i \delta_{l_i} \leq S$. Then providing this choice as our certificate, the verifier must accept.

Checking the validity of input can be done in $O(n)$ time - checking the i-th choice falls in the i-th quadrant takes $O(|Q_i|)$ time, and adding this together for all i we get $O(\sum_i |Q_i|) = O(n)$. Checking the input size is correct can be done in constant time (technically logarithmic as our input scales linearly with $n > k$, but this doesn't change things). Adding the benefits and weights and comparing them to B and C takes at most linear time in the number of bits used to encode these values. Thus as each step is polynomial time in the input size, our verifier is polytime, and so as there exists a polytime verifier for the decision problem, it is in NP.

2.d

We show a reduction from the knapsack problem to this problem.

The decision knapsack problem can be framed thus: Given N objects with a benefit b_m and weight w_m for each object m, and a capacity C and a minimum total benefit B, does there exist a choice of $x_m = \{0, 1\}$ for each object such that $\sum_m x_m b_m \geq B$ and $\sum_m x_m w_m \leq C$.

Our construction is as follows: we construct N quadrants matching the objects, and $2N$ planets. Each quadrant m has 2 planets, one with a distance $(\sum_i b_i) - b_m$ and cost w_m and one with a distance $\sum_i b_i$ and cost 0. We choose $D = C$ and $S = N \sum_m b_m - B$. The sum offset in the distances is to ensure distances remain non-negative.

We show first that if an instance of knapsack decision is accepted then our constructed instance is accepted. If the knapsack instance is accepted then there exists a choice of x_m so that $\sum_m x_m b_m \geq B$ and $\sum_m x_m w_m \leq C$. Then for our problem, for each quadrant m if $x_m = 0$ then select the planet with cost $0 = x_m w_m$, otherwise if $x_m = 1$ selecting the planet with cost $w_m = x_m w_m$. We also observe that the distance ends up being equal to $\sum_i b_i - x_m b_m$ under this choice. Then $\sum_m c_{l_m} = \sum_m x_m w_m \leq C = D$ and $\sum_m \delta_{l_m} = \sum_m (\sum_i b_i - x_m b_m) = M \sum_i b_i - \sum_m x_m b_m \leq N \sum_i b_i - B = S$. With a choice of x_m existing such that $\sum_m c_{l_m} \leq D$ and $\sum_m \delta_{l_m} \leq S$ our construction is an accepted instance of our problem.

We next show that if our constructed instance is accepted, then the knapsack decision instance is accepted. If our constructed instance is accepted then there exist a choice l_m of planets such that $\sum_m c_{l_m} \leq D$ and $\sum_m \delta_{l_m} \leq S$. If we select the planet with cost 0 for some quadrant m, then select $x_m = 0$ for the knapsack problem, and otherwise select $x_m = 1$. We see that under this choice $c_{l_m} = w_m x_m$ and $\delta_{l_m} = (\sum_i b_i) - x_m b_m$. Then, $\sum_m x_m b_m = \sum_m (\sum_i b_i - \delta_{l_m}) = N \sum_i b_i - \sum_m \delta_{l_m} \geq N \sum_i b_i - S = N \sum_i b_i - (N \sum_m b_m - B) = B$. $\sum_m x_m w_m = \sum_m c_{l_m} \leq D = C$. With a choice of x_m existing so that $\sum_m x_m b_m \geq B$ and $\sum_m x_m w_m \leq C$, the knapsack instance is an accepted instance.

Thus we have a reduction from knapsack decision to this problem. This reduction is polytime - after calculating $\sum_m b_m$ in $O(N)$ time (addition does take linear time in the bit size of integers, but this doesn't change the fact this takes polynomial time), we can construct each planet's distance and cost following the formulae in the construction in constant time for each planet, and with $2N$ planets this entire process takes $O(N)$ time. Constructing the partition similarly will take $O(N)$ time, through the process of iterating through the $2N$ planets. Calculating D and S takes constant time (once we've calculated $\sum_m b_m$) (while addition/multiplication/comparison does not actually take constant time, it doesn't affect the result that the reduction is poly-time as additional and multiplication and comparison add a quadratic factor at worst). Summing these together, it is clear the reduction is polytime.

Thus, as knapsack decision is an NP-complete problem, our reduction shows our decision problem is NP-hard. Our problem is also in NP, so it is NP-complete.

2.e

We can easily cook reduce the decision problem to the search problem - we run a search problem oracle and get a choice of planets in each quadrant. We then compute $\sum_i \delta_{l_i}$ for this choice and get some value S_{min}. Then for our decision problem, if $S \geq S_{min}$ then accept, and otherwise reject as there exists a choice of planets with total distance less than S if and only if the minimum total distance S_{min} over all planet choices is less than S. This reduction is clearly polytime - we call the oracle once, compute a sum of k values and then compare two values (although the sum and comparison may be linear in bit size). Thus as the decision problem is NP-complete, the search problem must be NP-hard.

2.f

We will use dynamic programming where our sub-problem takes as input the following: the number of planets n, partition of planets into quadrants $\{Q_i\}_i$, the costs and distances for each planet $\{c_l\}_l$

and $\{\delta_l\}_l$, and the number of dollars remaining to spend D and as output gives a choice of planets l_i alongside a distance sum $S = \sum_i \delta_{l_i}$. We can relate a sub-problem to smaller sub-problems with quadrant removed by a recursion. To get the sub-problem, from each possible of choice of planet l_1 in the first quadrant with cost within our remaining budget, we add its distance δ_{l_i} to the distance sum S given by a smaller sub-problem with the first quadrant removed and budget reduced from the starting sub-problem by c_{l_i} to get a new distance sum S'. We then select the l_1 that gives the minimum S' values over all the choices of planet in the first quadrant and append to the l_i choices for the corresponding sub-problem with first quadrant removed. We reach a base case when either we cannot afford anything but still have planets left to choose, in which case we return any arbitrary choice of planets along with a sum of ∞, or there are no planets or quadrants remaining and we return an empty list for l_i choices and 0 as the distance sum. We use memoisation to record sub-problems in a table. To solve the problem, we call an original instance of the sub-problem, setting n to the total number of planets, $\{Q_i\}_i$ to the partition of all planets into quadrants, the costs and distances for all planets as $\{c_l\}_l$ and $\{\delta_l\}_l$, and D as the total budget. We then return the planet choice l_i given by this sub-problem.

2.g

First, we show our recursion is correct. The minimum distance for each affordable choice of planet in the first quadrant of a sub-problem is clearly given by the distance for the chosen planet added to the minimum distance for the remaining quadrants with the cost appropriately reduced. Our algorithm chooses the planet choice that gives the minimum over the minimum distance for all choices of planet in the first quadrant that can be afforded, thus giving a minimum distance for the sub-problem. So our recursion is correct.

We will always reach a base case, as each recursion reduces the number of quadrants by 1, so if we do not reach a base case from no more affordable planets, we will reach a base case as no quadrants remain. As for justifying the base cases themselves - if we cannot afford any more planets then we return ∞ as a placeholder for there being no possible choice of planets. This means that our recursion will never select a planet that leads to there being no possible choice if there is another viable option with a possible choice as ∞ is greater than all integers. If we have no more planets to choose, we return an empty set with no planet choices made and a minimum distance of 0 as the with no planets chosen, the sum of distances is always 0.

Finally, our sub-problem for the original problem's input is equivalent to the initial problem, justifying our algorithm calling the sub-problem on the original problem's input parameters.

2.h

There are kD sub-problems, as each sub-problem is defined by how many quadrants there are remaining (k) and how much money is remaining (D). Each recursive step to calculate a subproblem takes $O(n)$ time, because we iterate over the choices of planet in the quadrant, and there are at most n choices. Constructing the input to each sub-problem can be done in constant time if we store the length of lists alongside them - then we simply need to remove terms from the lists efficiently which can be done in constant time using an array implementation. The problem can be solved directly with a sub-problem call so we only add on a constant time factor for translating the original problem to a subproblem. Thus the complexity is $kD * O(n) = O(knD) = O(n^2 D)$.

3

3.a

Iterating through each vertex, we calculate the gradient of all vertex pairs including it. If the line passing through the pair is vertical, we can set the gradient to any unique value, we arbitrarily choose ∞. We sort by gradient and then iterate through each of the vertex pairs including this vertex, and if two adjacent pairs in the sorted list have the same gradient, we accept. If after iterating through all vertices, we have not accepted, we reject.

3.b

Three points are collinear if and only if two distinct pairs of these three points have the same gradient. Any two distinct points must share an vertex. Thus we can reformulate the equivalent condition for some triplet of points being collinear as there exists a vertex v and two vertex pairs (u, v) and (v, w) such that (u, v) and (v, w) have the same gradient. After sorting by gradient, pairs with the same gradient form a contiguous section in the sorted list and thus if two or more pairs share a gradient then two of them will be adjacent. Thus our algorithm, which checks for adjacent pairs with the same gradient in the list of pairs containing a vertex v equivalently checks for pairs (u, v) and (v, w) with the same gradient, which is equivalent to there being three collinear points including v. Iterating over all v, our algorithm then accepts if and only if there is a collinear triplet.

3.c

We iterate over n vertices, and on each iteration sort and then search through the $n-1$ vertex pairs including the vertex being iterated on. The sorting step takes $O(nlogn)$ time at best and the search takes $O(n)$ time. Thus the running time is $n(O(n) + O(nlogn)) = n(O(nlogn)) = O(n^2logn)$.

3.d

We construct a reduction from ‡ to the collinearity problem. We begin by sorting $A[i]$ and for all groups of equal $A[i]$ values if 1) there are three or more duplicates of the same value equal to 0 or 2) there are only two, and after binary searching for $-2A[i]$ in the array where $A[i]$ is the duplicated value we find it, then construct any arbitrary collection of points with a collinear triplet. Otherwise, for each array element $A[i]$, we construct a point $(A[i], A[i]^3)$.

3.e

We want to show that if an instance of ‡ is accepting then our constructed collinearity problem instance is accepting. Let there be some i, j, k so that $A[i] + A[j] + A[k] = 0$. If any two of these are equal, then either (up to relabelling) $A[i] = A[j]$ and $A[k] = -2A[i]$ or $A[i] = A[j] = A[k] = 0$ then the construction will have a collinear triplet and thus be accepting. Otherwise, all three values are distinct and the gradient is well defined for all point pairs. Then the difference in gradients between the point pair $\frac{A[i]^3-A[j]^3}{A[i]-A[j]} - \frac{A[i]^3-A[k]^3}{A[i]-A[k]} = A[i]^2 + A[i]A[j] + A[j]^2 - A[i]^2 - A[i]A[k] - A[k]^2 = A[i](A[j]-A[k])+(A[j]-A[k])(A[j]+A[k]) = (A[i]+A[j]+A[k])(A[j]-A[k]) = 0$ as $A[i]+A[j]+A[k] = 0$. As the difference in gradients is zero, the three points $(A[i], A[i]^3)$, $(A[j], A[j]^3)$, and $(A[k], A[k]^3)$ in the constructed instance are collinear.

We now want to show that if an instance of our constructed problem is accepting then the instance of ‡ is accepting. Then either, we have some case where $A[i] = A[j]$ and $A[k] = -2A[i]$ or $A[i] = A[j] = A[k] = 0$ (so that our constructed problem has some arbitrary collinear triplet) in which case $A[i] + A[j] + A[k] = 0$ as desired, or there is some i, j, k so that $(A[i], A[i]^3)$, $(A[j], A[j]^3)$, and $(A[k], A[k]^3)$ are constructed points that are collinear. To simplify things, if two points overlap in a triplet, this does not count as collinear. This ensures that $A[i], A[j], and A[k]$ are distinct. Then $0 = \frac{A[i]^3-A[j]^3}{A[i]-A[j]} - \frac{A[i]^3-A[k]^3}{A[i]-A[k]} = A[i]^2 + A[i]A[j] + A[j]^2 - A[i]^2 - A[i]A[k] - A[k]^2 = A[i](A[j] - A[k]) + (A[j]-A[k])(A[j]+A[k]) = (A[i]+A[j]+A[k])(A[j]-A[k])$. As $A[j] \neq A[k]$, we get $A[i]+A[j]+A[k] = 0$ as desired. Thus the instance of ‡ is accepting. This proves our reduction is correct.

(Our reduction sorts in $O(nlogn)$ and then iterates through the sorted list. For duplicates, it may perform a binary search through the list to find $-2A[i]$ in $O(logn)$ time, so the process of iterating through the sorted list and dealing with duplicates takes $O(n^2)$ time. In the case of duplicate being involved in a 0 sum, we can arbitrarily construct an instance with three collinear points in constant time. Otherwise, we need to iterate through all array elements to construct $(A[i], A[i]^3)$ which adds another $O(n)$ time factor. This reduction thus takes a total of $O(nlogn) + O(nlogn) + O(n) = O(nlogn)$. Thus if there is a sub-$n^{1.999}$ algorithm for our collinear points problem, then combined with this $O(nlogn)$ reduction, we can get a sub-$n^{1.999}$ algorithm for ‡. This is a contradiction, and thus there is no sub-$n^{1.999}$ algorithm for our collinear points problem.)