CS4102 - ALGORITHMS - SPRING 2022

DUE TUESDAY, MAY 3, 2022 AT 11:30 PM VIA GRADESCOPE

DADV(SSB3VK)

Collaboration Policy: You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: mak3zaa

Sources: Cormen, et al, Introduction to Algorithms. *(add others here)*

PROBLEM 1 *IKEA Grill*

IKEA is growing in popularity across the US, however their stores are only found in a handful of larger metropolitan areas. While their main product is furniture, they have become known for their signature meatballs. To increase profits and make their delicious food more accessible, they have decided to open local IKEA Grill restaurants in towns across the country. Restaurant storefronts are expensive to rent and maintain, so they are happy with customers needing to drive at most to the next town over to get their IKEA meatball fix. Specifically, their goal is that every town in America either has an IKEA Grill, or its neighboring town has an IKEA Grill.

Given a graph representing the towns (as vertices) and roads between them (edges connecting neighboring towns), the *IKEA Grill* problem is to decide whether k IKEA Grill locations can be placed in order to ensure that each town or its neighbor has an IKEA Grill location. Show that *IKEA Grill* is NP-Complete.

Note: You are not being asked to explicitly solve the *IKEA Grill* problem; you are only required to show that it is NP-Complete.

Solution: The *IKEA Grill* problem is essentially the min-vertex cover problem. But by converting it to a problem of "whether k IKEA Grill locations can be placed in order to ensure that each town or its neighbor has an IKEA Grill location" we convert it into the Verification Problem form of vertex cover. This makes sense, given a graph all we need to do is check that any set of k vertices will touch all the edges in the graph.

The harder part is to prove that the k Vertex Cover problem is NP-Complete. This comes in two parts. The first part is that we have to prove that this problem is NP and the second part is that we have to prove that the problem is NP-Hard.

To prove that this problem is NP, we have to prove that a solution can be verified in Polynomial time (making it NP). To solve said problem, we can use the following polynomial time algorithm. Where V' is the solution set of k vertices that encompasses $G(V, E)$.

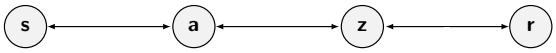
```
kIKEAGrillChecker(V', V, E, k)
    int count = 0
    for v in V'
        for e in E
            if v has edge e:
                remove v.e(u) from V // the node that v connects to
        count++

    if count == k and V.empty()
        return True
```

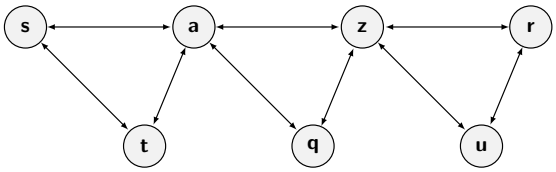
```
    else:
        return False
```

Solution: This algorithm takes either O(V') time or O(E) time and is therefore polynomial time. Because we can verify a solution to our problem in Polynomial time, we can conclude that k Vertex Cover is indeed NP. Now we have to show that IKEA Grill is NP-Hard. The easiest way to go about this is to show that a different NP-Hard problem reduces to our problem. This enables us to show that IKEA Grill is NP-Hard because if all NP problems reduce to an NP-Hard problem that reduces to IKEA Grill, then by transitivity we can reduce all NP-Hard problems to IKEA Grill.

The way we go about this is that we use a solver for IKEA Grill to solve the vertex cover problem. But the process of doing so poses some difficulty. Namely, before passing in a Vertex Cover input for processing in IKEA Grill, we perform the modification that per edge, we add a node that connects to both nodes on either side of the edge. So the following example input into vertex cover:



Solution: Would instead look like this:



Solution: We then pass the above graph into IKEA Grill. Note that IKEA Grill is guaranteed to select vertex cover as a solution. This is for two reasons. The first reason is that IKEA grill will never select any of the nodes that we've added. In this case, IKEA Grill will never select nodes t, q, or u. This is because selecting nodes t, q, or u cannot satisfy a Vertex cover problem. Further, t and q or u would never be selected because at most they can only cover at most two edges (in the case of Vertex cover) and two nodes (in the case of IKEA Grill) whereas other selections (namely a) can account for 4 edges and 4 nodes. We can also emplace an artificial limitation that we are unable to select the nodes we added when determining the solution (nodes t, q, u). This handles corner cases where we have only two nodes or three nodes and IKEA Grill accidentally selects a node that was artifically added. The second reason is that IKEA grill will only select nodes that are at most one node apart from one another. So in the example above, IKEA Grill would only select nodes s and z or a and t or nodes a and z. This is because selecting nodes s and r leaves node q unserviced. This generates a valid solution to vertex cover because all pair-wise selections of the set of (s, a, z, and r) work except for the selection of s and r. We can then return the selection set that IKEA Grill produces as a valid answer to vertex cover thereby demonstrating the reducibility of vertex cover to IKEA Grill.

The reduction cost here is linear with the number of edges. As we add a node per edge (effectively).

PROBLEM 2 *Backpacking, Revisited*

After your first successful backpacking adventure (Unit C's Advanced, Problem 1), you have decided to return to Shenandoah National Park. Similar to before, you and your friend have completed your packing list, and you need to bring n items in total, with the weights of the items given by $W = (w_1, \ldots, w_n)$. Your goal this time is to divide the items between the two of you such that the difference in weight is as small as possible. There is no longer a restriction on the total number of items that each of you should carry. Here, we will define a decisional version of this BACKPACKING problem:

> BACKPACKING: Given a sequence of non-negative weights $W = (w_1, \ldots, w_n)$ and a target weight difference t, can you divide the items among you and your friend such that the weight difference between backpacks is at most t?

1. Show that the BACKPACKING problem defined above is NP-complete (namely, you should show that BACKPACKING $\in$ NP and that BACKPACKING is NP-hard). For this problem, you may use the fact that the SUBSETSUM problem is NP-complete:

   > SUBSETSUM: Given a sequence of non-negative integers $a_1, \ldots, a_n$ and a target value t, does there exist a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = t$?

   Solution: For a problem to be NP-Complete, it must be both NP and NP-hard. First, we will focus on guaranteeing that this problem is indeed NP. This is most easily proved by showing that a solution to this problem can be verified as a solution to this problem in polynomial time. But, this is a decision problem, so given a set W, an upper bound on the difference between bags (u), an output (true/false), and a satisfying solution subset (representing one of the bags), we would need to process W so that we can verify that two subsets can be made with a difference in weight that is less than u. This would just entail iterating over the bag that we're given, taking the sum of its contents to get the total weight in the bag, subtracting from the weight of all the objects to get the weight of the two bags. We can then observe the difference between the two bags and check whether or not the difference is bounded by u ($|(Bag - weight - 1) - (bag - weight - 2)| \leq u$).

```
backpackChecker(W, u, output, bag-set-1):
    bag-weight-1 = 0
    for i in bag-set-1
        bag-weight-1 += i

    totalWeight = 0
    for w in W
        totalWeight += w

    bag-weight-2 = totalWeight - bag-weight-1

    if ( |bag-weight-1 - bag-weight-2| <= u == output ) {
        return SolutionWorks
    } else {
        return SolutionIsWrong
    }
```

Solution: This both determines the validity of the answer to the decision problem and runs in polynomial time. The reason that this decision-checker works is because it iterates over the solution in O(size(W)) time and can check if the given solution set and answer (u) correspond to a real solution to the problem. After computing the weights of both bags, it checks if the difference is indeed bound by u and correspondingly returns whether or not all the given inputs are valid provided the conditions of the problem. Namely, that the difference in the two bag weights are indeed bounded by t and the "output" field (whether or not the difference between the two bags can be bounded by u) is possible.

Now, we have to show that the Backpacking decision is NP-Hard. This is most easily done by reducing another NP-Hard problem to our Backpacking decision problem. This is because the definition of NP-Hard is that all NP problems can reduce to said problem. And if we reduce a known NP-Hard problem to our problem, then by transitivity all problems in NP are reducible to our problem in polynomial time. How can we reduce SubsetSum (a known NP-complete problem, and therefore an NP-Hard problem) to our Backpacking problem?

We can simply convert the input sequence of $a_1, \ldots, a_n$ to a series of Weights and add two additional elements, of two weights: $2 * A - t$ and $A + t$ where A is equal to the sum of all the weights/a's and t is the amount we want one of the bags to weigh in total. The reason that this transformation is optimal is because a solution to backpacking allocates both bags so that the difference between the bags is minimal. The reason that this works is because the weights of both bags will be approximately equal to 2A each after backpacking is finished allocating them as backpacking selects the backpack allocation that has the minimum difference between bags. Therefore we have a 3-step process to reducing SubsetSum to the Backpacking problem:

(a) Add two additional elements to the set A, namely 2 * sum(A) - t and sum(A) + t

(b) Perform backpacking on the generated set with a conditional check to see if the difference between the two bags is less than or equal to 0 or not

(c) Take the return value of backpacking and directly output it back as a solution of SubsetSum

The reason that this works is that Backpacking will allocate the backpacks such that their weight difference is minimized. By adding the 2A - t term and the A + t term, we add a little correcting factor so that we can actually compute whether or not a subset of A adds up t. If there is such a subset, that subset will be part of the 2A - t backpack and the total weight of the backpack will be 2A. The leftover part of A will be added to the A + t backpack and because the leftover A has mass equal to A - t, the weight of that backpack will also add up to 2A and the difference between the two backpacks will be 0 and we backpacking will return true (provided that u = 0). We can then use backpacking's output as SubsetSum's output.

Namely, when the backpacking problem returns true (that the two sets are equal, both add up to 2A), we can take the two backpack sets and observe the set that has the element 2A - t, remove that element from that set, and we get a solution set for subsetSum that adds up to t. Otherwise, in the case that Backpacking returns false, we can just return false from SubsetSum.

The reduction cost here is linear, as we iterate over the elements of A and add them up.

2. Your solution to Unit C Advanced's Problem 1 can be adapted to solve this version of the BACKPACKING problem in time that is *polynomial* in n and M, where $M = \max(w_1, \ldots, w_n)$ is the maximum weight of all of the items. Why did this not prove P = NP? (*Conversely, if you did prove that P = NP, there's a nice check waiting for you at the Clay Mathematics Institute.*)

Solution: We actually cannot solve the Backpacking problem in polynomial time. Instead, we solve it in pseudo-polynomial time. The algorithm that I wrote runs in $\Theta(n^2 * m)$ time. While this may appear to be polynomial time at first glance, we find that m is not a size, instead it is a value. But when we convert m to a size, we find that the runtime is closer to the form of $\Theta(n^2 * log(m))$. Because we have created a pseudo-polynomial solution to this problem instead of a polynomial solution, we cannot apply the implications of showing that a NP-Complete problem is solvable in polynomial time and cannot say that P = NP. M itself is exponential to the number of bits needed to represent M as the size of the input (M), therefore our solution to backpacking is not polynomial and cannot be used to prove that P = NP.

PROBLEM 3 *Gradescope Submission*

Submit a version of this `.tex` file to Gradescope with your solutions added, along with the compiled PDF. You should only submit your `.pdf` and `.tex` files.