Homework #3

Advanced Algorithms Spring 2021

Name: Lilo Heinrich

Time Spent: 8 hours

Problem 1

You are running a relatively new fancy restaurant that is determined to stay afloat during the pandemic by processing online orders. You employ m chefs $C_1, ..., C_m$ and have a set of n online orders. Each order has a cooking time of t_j. Your job as the manager is to assign each order to one of the chefs so that the loads placed on all the chefs are as "balanced" as possible since you are a nice boss and understand that burnout is totally a thing.

In other words, you seek to **minimize a quantity known as the makespan**, which is simply the **maximum load on any chef**. The load for a given chef can be determined by *finding the total cooking time needed to complete all the online orders assigned to that chef*.

Since you have a ton of other things on your plate (pun intended), you want to create an algorithm that can do all this assigning work for you. However, finding an assignment of orders to minimize the makespan is an NP-hard problem. As a result, your goal is to **find an approximation algorithm that runs in polynomial time and outputs solutions (i.e. makespans) that are guaranteed to be at most twice the optimal solution**.

Questions:

a) Find a 2-approximation algorithm for this problem, and show why is it a 2-approximation algorithm. (no need to prove just carefully explain why). **Hint: think in terms of a simple greedy algorithm**. Please include an explanation of:

- How your algorithm works

  Arrange the orders by their cooking time, labelling the highest cook time order as t_1 and the lowest cook time order as t_n. Go through the orders from longest to shortest, assigning each order the cook with the lowest cook time so far. Repeat until all orders have been assigned.

- Pseudocode

```
orders = a queue of (ID, time) sorted by time

def assign_orders(m, orders):
    assm = [[] for i in range(m)]
    times = [0 for i in range(m)]

    while orders:
```

```
        order = orders.pop()
        idx = times.index(min(times))
        assm[idx].append(order.ID)
        times[idx] += order.time
    return assm, times

assm, times = assign_orders(m, orders)
max_time = max(times)
```

- An explanation of why it's a 2-approximation algorithm

  First, the minimal makespan must be greater than or equal to the longest order:

$$C^* \geq \max_j t_j \tag{1}$$

  And secondly, the minimal makespan must be greater than or equal to the average time per chef, which is also the sum of order times so far divided by the number of chefs:

$$C^* \geq \frac{1}{m} \sum_{i=0}^{m} C_i = \frac{1}{m} \sum_{j=0}^{n} t_j \tag{2}$$

  Consider a machine with minimum makespan C^*. Let j be the last order left and let it get assigned to chef k. Using the algorithm above, we know that the cooking time of chef C_k must have been less than or equal to the average so far before t_j was assigned:

$$C_k - t_j \leq \frac{1}{m} \sum_{i=0}^{m} C_i \tag{3}$$

  Also, the order j will have a cooking time less than or equal to the maximum order cook time:

$$t_j \leq \max_i t_i \tag{4}$$

  Therefore, the final cooking time of chef C_k can be bounded by:

$$C_k = (C_k - t_j) + t_j \leq \frac{1}{m} \sum_{i=0}^{m} C_i + \max_i t_i \tag{5}$$

  Since C^* must be greater than or equal to both the average time per cook and the maximum order time, as described above in equations (1) and (2), the following must be true:

$$\frac{1}{m} \sum_{i=0}^{m} C_i + \max_i t_i \leq 2C^* \tag{6}$$

  Combining these two inequalities to relate the cook time for any chef C_k to the optimal maximum cook time C^* shows that this algorithm will assign all chefs a cook time that is less than $2C^*$, making it a 2-approximation algorithm.

$$C_k \leq 2C^* \tag{7}$$

b) **Optional:** Find a 1.5-approximation algorithm for this problem, and show why it is a 1.5-approximation algorithm. **Hint: think in terms of a simple greedy algorithm, should be just a minor change from 2-approximation algorithm**

Problem 2

a) An independent set of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident on at most one vertex in V'. The independent set problem is to find a maximum-size independent set in G. Recall that a **clique** of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. The clique problem is to find a maximum-size clique in G. Suppose that you were given an approximation algorithm A for the independent set problem that has a ratio bound of two. **Describe how you can use A to obtain a 2-approximation algorithm for the clique problem. Hint: Think reduction!**

Let V' be an independent set of graph $G = (V, E)$, and let graph H be the inverse of graph G. Take the inverse of graph G by removing every edge between vertices that were adjacent in G, then adding an edge between each pair of vertices that were not adjacent in G.

Now look at the same subset of vertices V' in graph H. Since none of the edges between each pair of vertices in the subset V' were adjacent in G, they must all become adjacent in H, making V' a fully connected subgraph of H also known as a clique. Therefore, for any independent set V' of some graph G, there is a corresponding clique of the same size and set of vertices in H.

If V' is the maximum clique of some graph H, then V' must also be the maximum independent set of its' inverse G. For every clique in H there is a corresponding independent set of the same size in G, and vice versa, so the maximum of one must also be the maximum of the other.

Therefore, an approximation of the maximum independent set problem for some graph G translates directly into an approximation of the maximum clique problem for its' inverse H. Apply algorithm A to the inverse of the clique graph to find the 2-approximation of the clique problem.

b) Can we use an approximation algorithm for the minimum-size vertex cover to design a comparably good approximation algorithm for the maximum-size independent set?

Let J be a vertex cover of graph $G = (V, E)$, and let I be the subset of vertices in G but not in J.

For each edge e in graph G, at least one of the two vertices that e connects must be in J. In other words, at least one of the two vertices that e connects must not be in I, making I an independent set. If I was not an independent set, then it would mean that there is some edge e' that connects two vertices in I. This would invalidate J as a vertex cover as it would not cover edge e'.

Considering isolated vertices (vertices with no connecting edges), they do not have any edges that need covering and therefore should be excluded by the minimum vertex cover. They are independent from all other vertices and so they belong in the maximum independent set.

Apply the approximation algorithm for the minimum-size vertex cover to some graph G to get a subset of vertices J which is a vertex cover. Create the independent set I that is the subset of vertices that are in G but not in J. The smaller the approximated minimum vertex cover, the larger the independent set, and vice versa.

Problem 3

Let's get our hands dirty! It's nice to devise algorithms and discuss how they are theoretically attractive, but it's good to also work with actual implementations and see how those perform. So, we're asking y'all to code up an implementation of a 2-approximation (or better!) for the metric TSP problem. *(Hint: use the implementation discussed in class.)* Specifically, hit the following targets:

1. Make a working implementation of a 2-approximation algorithm for the TSP using the Minimum Spanning Tree (MST) approach from class. (We provide starter code and tests here).

   GitHub link: https://github.com/liloheinrich/MST-TSP
   I've also pasted in the most relevant sections of code I worked on:

```
def tsp(adjList, start):
    tour = []

    for v in adjList:
        v.visited = False

    stack = []
    stack.append(start)

    while stack:
        curr = stack.pop()
        curr.visited = True
        tour.append(curr.rank)

        for n in curr.mstN:
            if n.visited is False:
                stack.append(n)

    tour.append(start.rank)
    return tour
```

2. Make a brute-force implementation to find the true optimal solution to the TSP.

```
def getTSPOptimal(self):
    final_cost = float("+inf")
    vertices_left = copy.deepcopy(self.adjList[1:])
    circuits = list(itertools.permutations(vertices_left))

    for circ in circuits:
        c = list(circ)
        c.insert(0, self.adjList[0])
        temp_cost = 0
        for i in range(len(c)):
            temp_cost += self.adjMat[c[i].rank][c[(i+1) % len(c)].rank]

        if temp_cost < final_cost:
            final_cost = temp_cost
            for i in c:
                self.tourOpt.append(i.rank)

    return self.tourOpt
```

3. Show that the MST approach is truly a 2-approximation with respect to the optimal solution found via brute-force. This part is a theoretical question - we want you to prove that the approximation method chosen is truly a 2-approximation or better! And of course, it's absolutely fine to regurgitate what we discussed in class.

   The MST approximation is to take the minimum spanning of a graph that follows the rules of the metric TSP problem. Pick a node on the MST to use as a root and draw it out as a tree T. Then do a DFS traversal of T, keeping track of the list of nodes as they are visited, and call this C. The cycle C will have twice the cost (sum of edge weights) of tree T because it must pass through each edge of the graph exactly twice during the traversal:

   $$c(C) = 2c(T)$$

   Read through C from start to end, removing duplicates as they are encountered to yield a Hamiltonian circuit C'. The cost of C' will be less than the cost of C, and since the cost of C is equal to twice the cost of T, we can create the following inequality:

   $$c(C') \leq 2c(T)$$

   Considering the optimal solution which we'll call H^*, remove any one edge e from H^* to turn it into a spanning tree T'. We know that T' is not the minimum spanning tree and so its' cost must be greater than or equal to the cost of T. We also know that the cost of H^* must be greater than the cost of T' since T' is just the optimal with one edge removed, $H^* - e$. Expressing this as one big inequality:

   $$c(H^*) \geq c(H^* - e) \geq c(T)$$

   Combining these two inequalities to relate the cost of the proposed solution C' to the cost of the optimal solution H^*, we find that:

   $$c(C') \leq 2c(H^*)$$

   Therefore, the circuit C' has a total cost of at most twice the optimal solution H^*, confirming that this algorithm produces a 2-approximation.