CMSC651 Assignment 3

Yancy Liao, Fan Yang, Bowen Zhi

March 2018

Problem 1

(a) Given as input $V = F \cup D$, distance function d, and k, consider the algorithm:

```
S ← ∅
pick c_1 ∈ D at random
for i = 1, ..., k :
    find s_i = argmin_{s∈F} d(s, c_i)
    if s_i ∈ S, break
    S ← S ∪ {s_i}
    find c_{i+1} = argmax_{c∈D} (min_{s∈S} d(s, c))
return S
```

We now show that this is a 3-approximation. Consider the optimal solution of cost OPT, consisting of $\{o_1, o_2, \ldots, o_\ell\}$, $\ell \leq k$. Note that we can assume $\ell = k$ without a loss of generality (since this solution is also optimal for $k = \ell$). Now, consider the area covered by k circles of radius OPT centered at each o_j, $j \in \{1, 2, \ldots, k\}$. Clearly all of D is contained in these circles, including $\{c_i\}_{i=1}^k$ picked by our algorithm.

Consider the case when there is exactly one c_i in each circle L_i. Note that since d satisfies the triangle inequality, the distance between c_i and any other customer in L_i is $\leq 2 \cdot \text{OPT}$. Further note that c_i has closest supplier s_i by construction of our algorithm, and hence $d(s_i, c_i) \leq \text{OPT}$ (if this weren't the case, then c_i is no longer the closest supplier). Using the triangle inequality, we see that the distance between s_i and any customer in L_i is $\leq 3 \cdot \text{OPT}$. Since this holds for every circle, the distance from any customer to the nearest provider in S also satisfies this bound.

For all other cases, there exists some circle L containing at least two customers c_i, c_j picked by our algorithm, with $i < j$. Note then that $d(c_i, c_j) \leq 2 \cdot \text{OPT}$. Moreover, we have $d(s_i, c_i) \leq \text{OPT}$ and thus by the triangle inequality, $d(s_i, c_j) \leq 3 \cdot \text{OPT}$. But recall that c_j was picked to be the customer furthest from $S_{j-1} = \{s_1, s_2, \ldots, s_{j-1}\}$, hence $\max_{c \in D} \left(\min_{s \in S_{j-1}} d(s, c)\right) \leq d(s_i, c_j) \leq 3 \cdot \text{OPT}$.

We have shown that in both cases our algorithm satisfies $\max_{c \in D} (\min_{s \in S} d(s, c)) \leq 3 \cdot \text{OPT}$, and thus is a 3-approximation.

(b) Suppose we are given an undirected graph $G(T, E), T = \{t_i\}_{i=1}^n$. Create suppliers $F = \{s_i\}_{i=1}^n$ and customers $D = \{c_i\}_{i=1}^n$. Construct a distance function d over $D \cup F$ such that $d(s_i, c_i) = 0$ and $\forall\, i \neq j$, $d(s_i, c_j)$ is 1 if there is an edge $\in E$ connecting t_i to t_j, and 3 otherwise. In order to satisfy the triangle inequality, set $d(s_i, s_j) = d(c_i, c_j) = 2$, $i \neq j$. To see why we need this, consider when two suppliers/customers A, B are both 1 away to the same customer/supplier C. Then the distance between A and B is at most 2. If A is 3 away from C instead, then the distance between A and B is at least 2. Thus we need a distance of 2 between any pair of suppliers or customers. Note that the remaining case (when A and B are both 3 away from C) is also satisfied by this.

Note that if G has a dominating set of size k, then $\exists\, S \subset F$, $|S| = k$ s.t. $\max_{c \in D} (\min_{s \in S} d(s, c)) = 1$. Otherwise, $\forall\, S \subset F$, $|S| = k$, $\max_{c \in D} (\min_{s \in S} d(s, c)) = 3$. Now, suppose we have an algorithm that is a α-approximation to k-suppliers with $\alpha < 3$. Running this algorithm on the above F, D, d yields a cost < 3 if G has a dominating set, and ≥ 3 otherwise. Hence, we could solve the NP-complete dominating set decision problem in polynomial time with such an algorithm. Thus if there is a α-approximation to the k-suppliers problem with $\alpha < 3$, then P = NP.

Problem 2

Suppose we have n machines and k jobs, with the jobs' requirements labeled $p_1, p_2, \ldots, p_k$. Since the requirement of each job is strictly greater than one-third of the total job time, each machine can have no more than two jobs. It follows that $k \leq 2n$. WLOG, we arrange them so that $p_1 \geq p_2 \geq \ldots \geq p_k$. And WLOG, let's pad our list of jobs with jobs of requirement zero until we have exactly $2n$ jobs, so that $p_1 \geq p_2 \geq \ldots \geq p_{2n}$ where $p_i = 0$ for $i > k$.

The LPT algorithm will assign jobs to the machines as follows: $(p_1, p_{2n}), (p_2, p_{2n-1}), \ldots, (p_n, p_{n+1})$. Basically the largest job gets paired with the smallest job, the second-largest job gets paired with the second-smallest job, etc. We call this the "LPT assignment" and we need to show that this particular way of assigning jobs produces an optimal solution.

Suppose we have an optimal solution, with the slowest machine having jobs i and j.

We introduce terminology: a slow job is one whose index lies between 1 and n (its time requirement is high), and a fast job is one whose index lies between $n+1$ and $2n$ (its time requirement is low).

Claim: one of i and j is a fast job and the other is a slow job. Suppose not and they are both slow. Then, by pigeonhole principle, there must be a machine with two jobs that are fast. It follows that if we did a job swap between a job from the first machine and a job from the second machine, each of the resulting machines would have run-time quicker than the machine that previously had the two slow jobs. Since the machine that previously had the two slow jobs was the slowest machine of the schedule, we have hence improved the schedule, which is a contradiction of our original arrangement being optimal. Similarly, if we suppose that i and j both are fast, we also get a contradiction because we can find a machine with two slower jobs, which contradicts the assumption that i and j belong to the longest-running machine.

Claim: we can make every other machine in our solution also have one slow job and one fast job, without disturbing the optimality condition. This is because, for each machine with two fast jobs, we can find another machine with two slow jobs, and apply the "swapping" argument from above.

So far, we have shown that any optimal solution can be expressed with each machine having one slow job and one fast job.

Claim: if, in a schedule, each machine has one slow job and one fast job, the schedule will either have the same run-time as the "LPT assignment," or we can improve it by making it look like the "LPT assignment." First take the machines and order them by the slow jobs they were given, so we have machines ordered $1 \ldots n$. We argue that if their paired "fast" jobs are not $2n \ldots n+1$, then we can improve the schedule time. Take the machine with the longest run-time, call the jobs i and j, where i is the slow job and j is the fast job. If i and j correspond to one of the pairings of the LPT assignment, then we know that this arrangement has the run-time of the LPT assignment and we stop there. If i and j do not come from one of the pairs of the LPT assignment, then we know that the ordering of fast jobs is not $2n \ldots n+1$. Note then that we can either find a slow job i_* slower than i paired with a fast job j_* slower than j, or a slow job i_* faster than i paired with a fast job j_* faster than j. Since i and j came from the machine with longest run-time, the latter must be the case. Note then that we can do a job swap and improve the run-time of the resulting machines, and hence of the schedule. Moreover, this implies that so long as the ordering of such a schedule is not the "LPT assignment," we can always turn it into the "LPT assignment" via swaps that either decrease or preserve the longest run-time.

Hence, to wrap things up, any optimal solution can be expressed with each machine having one slow job and one fast job. Further, any solution with one slow job and one fast job will have the same run-time as the "LPT assignment" or can be improved it by turning it into the "LPT assignment." Hence the LPT assignment is indeed optimal.

Problem 3

Let's call the jobs $x_1, \ldots, x_n$. The way the algorithm will work is that it will start packing in jobs "tightly," i.e., in the usual greedy fashion with no gaps, until it comes across a "dominant" job which all remaining jobs depend on - the algorithm cannot proceed with the scheduling until this job has finished. If x_{*i} is a dominant job, that means for all remaining jobs x, $x_{*i} \prec x$. The algorithm will then place x_{*i} down (if it has not already been placed) according to the next available machine, and then there will be a gap across all machines until this dominant job is finished. The max gap length in this situation is the time requirement of x_{*i}. Once the dominant job is finished, the algorithm will begin packing in jobs tightly until it comes across another dominant job, by which another gap appears.

Let's say the schedule that the algorithm produces is a bunch of tightly packed areas except for gaps determined by dominant jobs labeled $x_{*1}, \ldots, x_{*p}$. Firstly, since the tightly packed areas are filled with jobs while entertaining no gaps, they cannot be longer than a full schedule of this problem, and hence cannot be longer than the optimal schedule. Secondly, the gaps also cannot be longer than the optimal schedule. This is because the sum of the gap lengths is at most the sum of the time requirements of $x_{*1}, \ldots, x_{*p}$. However, note that $x_{*1} \prec \ldots \prec x_{*p}$, and so therefore this non-time-overlapping sequence of jobs must appear in any feasible schedule of the problem, and hence in the optimal schedule. So the optimal schedule must also be longer than the sum of the time requirements of $x_{*1}, \ldots, x_{*p}$. Finally, we have shown that both parts of the schedule produced by the algorithm (the tightly packed areas and the gaps) can be no longer than the optimal schedule, and therefore the algorithm is a 2-approximation for the problem. Note: the algorithm is clearly polynomial.

Problem 4

Consider the relaxed knapsack problem in which the knapsack is allowed to take fractions of the items. Let C_{greedy} denote the value returned by the described greedy algorithm for the relaxed problem. We know:

$$C_{greedy} = \sum_{i=1}^{k} v_i + \alpha v_{k+1},$$

where $\alpha = \frac{B - \sum_{i=1}^{k} s_i}{s_{k+1}} \in [0, 1)$.
Note that the items have been sorted in decreasing order of their unit price v_i/s_i, so replacing any portion of the items in C_{greedy} will unavoidably decrease the unit price associated with that space, thus decrease the value of the answer. Hence, C_{greedy} is optimal for the relaxed problem.
Let OPT denote the optimal value of the original problem, then OPT $\leq C_{greedy}$. We have the following chain of inequalities:

$$\text{OPT} \leq \sum_{i=1}^{k} v_i + \alpha v_{k+1} < \sum_{i=1}^{k} v_i + v_{k+1} \leq \sum_{i=1}^{k} v_i + \max_{i \in I} v_i.$$

Thus, the larger one of $\sum_{i=1}^{k} v_i$ and $\max_{i \in I} v_i$ is at least as large as OPT/2, and the claim that the greedy algorithm is a 1/2-approximation for the knapsack problem is proved.

Problem 5

First, run the greedy algorithm from Problem 4, which returns a solution with value G. Now run the approximation scheme shown in Algorithm 3.2 from Williamson-Shmoys, except with $\mu = \epsilon\, G/n$. Suppose the optimal solution is O with value OPT. Clearly we must have $G \leq \text{OPT}$. Note then that the solution S obtained satisfies:

$$\sum_{i \in S} v_i \geq \mu \sum_{i \in S} v_i' \geq \mu \sum_{i \in O} v_i' \geq \sum_{i \in O} \mu v_i' - n\mu = \text{OPT} - \epsilon\, G \geq \text{OPT} - \epsilon\, \text{OPT} = (1-\epsilon)\text{OPT}.$$

Thus this modification to the algorithm does not change its approximation factor. Let us analyze the runtime of the algorithm, which consists of the greedy and the dynamic programming portions. As the former simply consists of a sort and iteration on the items, it can be done $\mathcal{O}(n \log(n))$ time. The latter can be done with Algorithm 3.1 from Williamson-Shmoys on the downscaled problem, which we now analyze. Suppose the optimal solution of the downscaled problem has value OPT'. Note that the size of A after each dominated pair removal can never exceed $B+1$ or $\text{OPT}'+1$, whichever is smaller. The latter bound holds since none of the pairs in A can have value above OPT' (otherwise OPT' would no longer be optimal). Further, since the size of A at worst doubles between removals, $|A|$ is always $\mathcal{O}(\min(B, \text{OPT}'))$. Thus the latter portion of the algorithm takes $\mathcal{O}(n \min(B, \text{OPT}'))$ time. Since Algorithm 3.1 yields the optimal solution, we have:

$$\text{OPT}' = \sum_{i \in S} v_i' \leq \sum_{i \in S} \frac{v_i}{\mu} = \sum_{i \in S} \frac{n v_i}{\epsilon\, G} \leq \frac{n}{\epsilon\, G} \text{OPT}.$$

As shown in Problem 4, we know that $G \geq \text{OPT}/2$. Thus, we have $\text{OPT}' \leq 2n/\epsilon$. Hence, the algorithm has overall runtime of $\mathcal{O}(n \log(n) + n\, \text{OPT}') = \mathcal{O}(n^2/\epsilon)$: a factor of n faster than the original Algorithm 3.2.

Problem 6

Suppose there exists an optimal schedule in which some ontime jobs finish after late jobs, and/or the ontime jobs do not complete in an earliest due date order. We know that the late jobs do not contribute to the value of the answer, because the objective of the problem is to maximize the total weight of the jobs that complete by their due date. Hence, whenever we see a late job that completes before an ontime job, we can simply move the late job to the end of the schedule and shift the jobs after its original position forward accordingly. This way we will yield a modified schedule in which all ontime jobs finish before all late jobs. Note that all ontime jobs in the original optimal schedule remain ontime in this modified schedule, therefore the modified schedule remains optimal.

Now assume there exist two jobs i and j in the modified optimal schedule such that i's due date is earlier than j but i finishes after j. We observe that if we re-insert job j to the position right after i and shift the jobs after j's original position forward accordingly, we will have a new modified schedule in which both i and j finish before i's due date and i finishes before j. This new modified schedule remains optimal because i, j and the jobs that finish before i remain ontime, and the jobs that finish after j (in its new position) are not affected. By repeating this process until no modification can be made, we will achieve a new optimal schedule in which all ontime jobs complete in an earliest due date order (and all ontime jobs finish before all late jobs as already proved in the previous paragraph).

We now show that the problem can be solved using DP in $\mathcal{O}(nW)$ time. First we sort the jobs in the earliest due date order. Similar to Algorithm 3.1 in Williamson-Shmoys, we maintain an array entry $A(j)$ for $j = 1, \ldots, n$. Each entry $A(j)$ is a list of pairs (t, v) and each (t, v) pair in $A(j)$ indicates that there is a feasible set of jobs $S \subseteq \{1, \ldots, j\}$ such that $\sum_{i \in S} p_i = t$ and $\sum_{i \in S} w_i = v$. We shall call a set of jobs feasible if all of its jobs can complete on time when sorted in due date order and placed at the front of the schedule. If there are two pairs (t, v) and (t', v') in $A(j)$ such that $t \leq t'$ and $v \geq v'$, then we say (t', v') is dominated by (t, v) and we remove (t', v') from $A(j)$. We set $A(0)$ to $\{(0, 0)\}$ to indicate that $t = v = 0$ when the subset of jobs is empty. The DP algorithm is shown on the next page.

```
sort the jobs in earliest due date order
A(0) ← {(0, 0)}
for j ← 1 to n do
    A(j) ← A(j − 1)
    for each (t, v) ∈ A(j − 1) do
        if t + p_j ≤ d_j then
            add (t + p_j, v + w_j) to A(j)
    remove dominated pairs from A(j)
return max_{(t,v)∈A(n)} v
```

The correctness of the above DP algorithm can be proved by showing that $A(j)$ contains all non-dominated pairs corresponding to feasible sets of $\{1, \ldots, j\}$. This trivially holds for $j = 0$ because both t and v are 0 when the set is empty. Now suppose it is true for $A(j-1)$. Let S denote one of the feasible sets of $\{1, \ldots, j\}$, and let $t = \sum_{i \in S} p_i$ and $v = \sum_{i \in S} w_i$. We claim that there is some pair $(t', v') \in A(j)$ that dominates (t, v). First, suppose that $j \notin S$. Then the claim follows by the fact that we initially set $A(j) \leftarrow A(j-1)$ and only removed dominated pairs. Now suppose $j \in S$. Then for $S' = S - \{j\}$, there is some $(\hat{t}, \hat{v}) \in A(j-1)$ that dominates $(\sum_{i \in S'} p_i, \sum_{i \in S'} w_i)$. Then the algorithm will add $(\hat{t}+p_j, \hat{v}+w_j)$, which dominates (t, v), to $A(j)$. Thus, we have proved $A(j)$ contains all non-dominated pairs corresponding to feasible sets $S \subseteq \{1, \ldots, j\}$.

Assume that the pairs contained in $A(j)$ are $(t_1, v_1), \ldots, (t_k, v_k)$ with $t_1 < t_2 < \cdots < t_k$. We must also have $v_1 < v_2 < \cdots < v_k$ since all of the pairs in $A(j)$ are undominated. This implies that $A(j)$ contains at most $W = \sum_j w_j$ pairs if p_j for every job is an integer. Thus, running time of the inner for-loop is $\mathcal{O}(W)$. The algorithm takes $\mathcal{O}(nW)$ time to finish as the inner for-loop will execute n times. Note that the first line of the algorithm is a sort of the jobs taking $\mathcal{O}(n \log n)$ time. $\mathcal{O}(n \log n)$ is smaller than $\mathcal{O}(nW)$ because $\log n < n \leq W$.

A FPTAS for this problem is similar to Algorithm 3.2 in Williamson-Shmoys. Let I be the set of all jobs that might be ontime, i.e., jobs that satisfy $p_j \leq d_j$, and let $n = |I|$. The FPTAS is shown below:

```
M ← max_{i∈I} w_i
μ ← εM/n
w'_i ← ⌊w_i/μ⌋ for all i ∈ I
run the above DP algorithm with values w'_i
```

The proof for this FPTAS is also given in the book. Below is a summary of it.

Let S be the set of jobs returned by the above approximation scheme, and let O be be an optimal set of jobs for the problem. Certainly $M \leq \text{OPT}$ and $w_i - \mu \leq \mu w'_i \leq w_i$. Combining this knowledge with the fact that S is an optimal solution for the values w'_i, we can derive the following chain of inequalities:

$$\begin{aligned}
\sum_{i \in S} w_i &\geq \mu \sum_{i \in S} w'_i \\
&\geq \mu \sum_{i \in O} w'_i \\
&\geq \sum_{i \in S} w_i - |O|\mu \\
&\geq \sum_{i \in S} w_i - n\mu \\
&= \sum_{i \in S} w_i - \epsilon M \\
&\geq \text{OPT} - \epsilon M = (1 - \epsilon)\text{OPT}.
\end{aligned}$$

Thus, the correctness of the FPTAS is proved.