Assignment 2

Navneel Singhal

2018CS10360

Contents

1 Problem 1

1.1 Statement

Consider the following scheduling problem: there are n jobs to be scheduled on a single machine, where each job j has a processing time p_j, a weight w_j, and a due date d_j, $j = 1, \ldots, n$. The objective is to schedule the jobs so as to maximise the total weight of jobs that complete by their due date. First prove that there always exists an optimal schedule in which all on-time jobs complete before all late jobs and the on-time jobs complete in the earliest due date order. Use this structural result to show how to solve this problem using Dynamic Programming in $O(nW)$ time where $W = \sum_j w_j$. Now use this result to derive a fully polynomial-time approximation scheme.

1.2 Solution

Claim 1.1

There exists an optimal schedule in which all on-time jobs complete before all late jobs and the on-time jobs complete in the earliest due date order.

Proof. For the first part, let S be an optimal schedule with jobs s_j in increasing order of starting/ending time (both these orders are equivalent since scheduling is done on a single machine without overlaps).

If for some $i < j$, s_i is a late job and s_j is an on-time job, we call (i, j) to be an *inversion*.

Among all schedules, wlog that S is one with the minimum number of inversions. If the number of inversions in S is not 0, then there must be at least one inversion in S. It then follows that for some index i, there is an inversion of the form $(i, i+1)$ (since if there wasn't, then either both i and $i+1$ would have been late or on-time, or i would have been on-time and $i+1$ would have been late, in which case there is no inversion by induction on the index i).

Consider the order $S' = s_1, \ldots, s_{i-1}, s_{i+1}, s_i, s_{i+2} \ldots, s_n$ (the prefix and suffix may be empty) which we get from swapping s_i and s_{i+1} in S.

Note that this is also an optimal schedule, since we are moving an on-time job to earlier, so it stays on-time in the new schedule as well, and the late job remains a late job since we're pushing it to a later time.

Note that after this swap, the only pair of indices whose relative order changes is $(i, i+1)$. So the number of inversions reduces by strictly 1, but this is a contradiction since S was an optimal schedule with the smallest number of inversions.

For the second part, let S be an optimal schedule with all the on-time jobs completing before all the late jobs. We call a pair of indices (i, j) a date-inversion if $i < j$ and $d_{s_i} > d_{s_j}$, and s_i, s_j are both on-time jobs. Again, as before, suppose that the least number of date inversions for any optimal schedule with on-time jobs before late jobs is non-zero. Then there exists a date-inversion of the form $(i, i+1)$ as well. Suppose the time at which job s_i started was t. Then we have $t + p_{s_i} \leq d_{s_i}$ and $t + p_{s_i} + p_{s_{i+1}} \leq d_{s_{i+1}}$. Upon swapping jobs s_i and s_{i+1} in the schedule, note that $t + p_{s_{i+1}} \leq t + p_{s_i} + p_{s_{i+1}} \leq d_{s_{i+1}}$ and $t + p_{s_i} + p_{s_{i+1}} \leq d_{s_{i+1}} < d_{s_i}$, the new schedule so formed is also a valid (and optimal) schedule. Hence we can reduce the number of date-inversions in this case as well, which is a contradiction.

This proves the claim. □

Using this result, we come up with the following algorithm to compute the maximum total weight of jobs we can get; it is straightforward to restore the schedule using the dynamic programming table constructed in the solution (note that we can remove all jobs which take more time than their due date, so we can also assume $p_i \leq d_i$):

```
1:  function SCHEDULE(n, p[1...n], w[1...n], d[1...n])
2:      Let W = sum_{i=1}^{n} w[i]
3:      Let dp[0...n][0...W] be a matrix initialized to all ∞s  ▷ dp[i][j] = minimum time taken to get a total
        weight of at least j from the first i elements
4:      dp[0][0] ← 0
5:      for i = 1...n do
6:          for j = 0...W do
7:              dp[i][j] ← dp[i − 1][j]
8:              if dp[i − 1][max(0, j − w[i])] + p[i] ≤ d[i] then
9:                  dp[i][j] ← min(dp[i][j], dp[i − 1][max(0, j − w[i])] + p[i])
10:             end if
```

11: **end for**
12: **end for**
13: **return** the largest w such that $dp[n][w] \neq \infty$ ▷ To restore the schedule from this, we can reconstruct our choices using the dp-matrix
14: **end function**

For a proof of correctness, note that we have the following induction to show that $dp[i][j]$ indeed is the quantity mentioned in the comment.

Proof.

1. Base case: $i = 0$: In this case, we can only get a weight of 0, so $dp[0][0] = 0$ and $dp[0][w] = \infty$ for $w > 0$, which matches.
2. Inductive step: $i > 0$. We have only two choices:
   - In the case where we choose the i^{th} element as an on-time job, we need to check if the minimum time taken to finish this job while having a total weight at least j is still $\leq$ its due date (else this won't be an on-time job), and this condition is sufficient as well. In the case that this condition is true, we only need to minimum time taken to complete jobs with total weight $\geq j - w[i]$ from the first $i-1$ elements. Upon noting that time taken can't be negative, we get the transition as in the algorithm by the inductive hypothesis.
   - In the case where we don't choose it, we can simply use the minimum time taken for some of the first $i-1$ jobs to reach a total weight of j (infinite if impossible), and this leads to the transition as in the algorithm by the inductive hypothesis.

Hence this completes the proof. □

Now the answer is the largest w which doesn't take infinite time to be achieved, and we are done. The time taken by the algorithm is $O(nW)$.

The approximation algorithm for a $(1-\varepsilon)$ approximation is as follows:

1. Let w' be the maximum weight of a job.
2. Replace w_i by $\lfloor \frac{nw_i}{w'\varepsilon} \rfloor$
3. Output the schedule returned by the previous algorithm on this instance.

Note that this schedule is valid, and since we didn't change the processing time or the due dates, the jobs which are on-time in this solution also remain on-time in the solution corresponding to the original problem.

Claim 1.2

This is a $1-\varepsilon$ approximation.

Proof. Let **OPT** be the ordering of the on-time jobs in an optimal schedule for the original problem. Let $\mathbf{OPT'}$ be defined similarly for the modified problem that we return.

Then we have the following:

$$\begin{aligned}
\sum_{j \in \mathbf{OPT'}} w_j &\geq \frac{w'\varepsilon}{n} \sum_{j \in \mathbf{OPT'}} \left\lfloor \frac{nw_j}{w'\varepsilon} \right\rfloor \\
&\geq \frac{w'\varepsilon}{n} \sum_{j \in \mathbf{OPT}} \left\lfloor \frac{nw_j}{w'\varepsilon} \right\rfloor \\
&\geq \frac{w'\varepsilon}{n} \sum_{j \in \mathbf{OPT}} \left(\frac{nw_j}{w'\varepsilon} - 1 \right) \\
&= \sum_{j \in \mathbf{OPT}} w_j - w'\varepsilon \frac{|\mathbf{OPT}|}{n} \\
&\geq \sum_{j \in \mathbf{OPT}} w_j - w'\varepsilon
\end{aligned}$$

Here the first inequality comes from rounding, the second from the fact that $\mathbf{OPT}'$ is an optimal solution to the modified problem, the third from rounding again, and the last one from the fact that we have at most n jobs in $\mathbf{OPT}$.
Now note that we can always take the maximum element as an on-time job and schedule everything else to be a late job (this is possible since we assume $p_i \leq w_i$ for all i). This shows that $\sum_{j \in \mathbf{OPT}} w_j \geq w'$. Plugging this into the above inequality, we have: $\sum_{j \in \mathbf{OPT}'} w_j \geq (1-\varepsilon) \sum_{j \in \mathbf{OPT}} w_j$, and we are done. □

Claim 1.3

The time complexity is $O(n^3/\varepsilon)$ (and hence this algorithm is a FPTAS).

Proof. The largest weight is reduced to $\lfloor \frac{n}{\varepsilon} \rfloor$, and since the mapping is a non-decreasing function, all weights are at most $\frac{n}{\varepsilon}$. So $W \in O(\frac{n^2}{\varepsilon})$, and the algorithm becomes $O(n^3/\varepsilon)$, and we are done. □

2 Problem 2

2.1 Statement

Consider the following scheduling problem: there are n jobs to be scheduled on a constant number of machines m, where each job j has a processing time p_j, and a weight w_j, $j = 1, \ldots, n$. Once started, each job must be processed to completion on that machine without interruption. For a given schedule, let C_j denote the completion time of job j, $j = 1, \ldots, n$ and the objective is to minimise $\sum_j w_j C_j$ over all possible schedules. First show that there exists an optimal schedule where, for each machine, the jobs are scheduled in non-decreasing p_j/w_j order. Then use this property to derive a dynamic programming algorithm that can be used to obtain a fully polynomial-time approximation scheme.

2.2 Solution

I will assume that the phrase "constant number of machines" implies that the number of machines is not in the input, but rather a part of the problem description.

Claim 2.1

Given a machine, there is an optimal schedule where the jobs on it are scheduled in non-decreasing p_j/w_j order.

Proof. Suppose not. Then any optimal schedule with the least number of inversions (in the array of p_j/w_j) has at least one inversion. This also implies that there is an inversion where the relevant indices are adjacent (else by induction, the array of p_j/w_j induced by this ordering is sorted, which is a contradiction).

Suppose that there is an inversion at indices $i, i+1$, i.e., if job k is at the i^{th} position, and job l is at the $(i+1)^{\text{th}}$ position, then we have $p_k/w_k > p_l/w_l$. Consider what happens when we swap their positions. Since the set of jobs in the prefix of size j where $j < i$ or $j > i+1$ doesn't change, the change in the objective function is only due to the terms arising from indices i and $i+1$.

The change is $-((C + p_k) \cdot w_k + (C + p_k + p_l) \cdot w_l) + ((C + p_l) \cdot w_l + (C + p_l + p_k) \cdot w_k) = -p_k \cdot w_l + p_l \cdot w_k = w_k w_l \cdot \left(-\frac{p_k}{w_k} + \frac{p_l}{w_l}\right) \leq 0$.

Hence, swapping leads to one less inversion (since there are no extra inversions added/removed by swapping consecutive positions) and doesn't increase the objective function. This is a contradiction to the fact that we have at least one inversion in any optimal schedule, which completes the proof. □

Now we will show a dynamic programming algorithm that can find an optimal solution using this lemma. What follows will work for processing times being real numbers, but we will be able to give a good bound on time complexity only when the processing times are integers (or rationals). For integers, rather than a red-black tree, we can use an $m-$dimensional array to make the implementation faster.

function SCHEDULE(n, $p[1 \ldots n]$, $w[1 \ldots n]$)
  Let $id[1 \ldots n]$ be the order of indices sorted by $p[j]/w[j]$
  Let $T = \sum_{i=1}^{n} p[i]$
  Let $dp[0 \ldots n]$ be an array of red-black trees supporting key-value mappings initialized to all empty trees.
    ▷ $dp[i]$ consists of key-value pairs where the key is the m-tuple of completion times of each machine in sorted order, and the value is the minimum objective value corresponding to any configuration with jobs in $id[1 \ldots i]$ with that key. Ordering is lexicographical.
  Let $parent[1 \ldots n]$ be an array of red-black trees supporting key-value mappings initialized to all empty trees.
    ▷ $parent[i]$ consists of key-value pairs where the key is the $m-$tuple of completion times of each machine in sorted order, and the value is the tuple of completion times, which when appended by the $id[i]^{\text{th}}$ job, leads to the key. Ordering is lexicographical here again
    ▷ We will represent these balanced binary search trees as sets.
  $dp[0] \leftarrow \{((0, \ldots, 0), 0)\}$
  for $i = 1 \ldots n$ **do**
    for $((t_1, t_2, \ldots, t_m), v)$ in $dp[i-1]$ **do**
      for $j = 1 \ldots m$ **do**
        Let $(t'_1, \ldots, t'_m) = (t_1, \ldots, t_m)$
        $t'_j \leftarrow t_j + p[id[i]]$
        Let $v' = v + w[id[i]] \cdot t'_j$

16: Sort the tuple $(t'_1, \ldots, t'_m)$
17: **if** $(t'_1, \ldots, t'_m) \notin dp[i]$ **then**
18: Insert $((t'_1, \ldots, t'_m), v')$ into $dp[i]$
19: Insert $((t'_1, \ldots, t'_m), (t_1, \ldots, t_m))$ into $parent[i]$.
20: **else if** the value of the key $(t'_1, \ldots, t'_m)$ in $dp[i]$ is $> v'$ **then**
21: Update its value to v'
22: Update the value of the key $(t'_1, \ldots, t'_m)$ in $parent[i]$ to $(t_1, \ldots, t_m)$.
23: **end if**
24: **end for**
25: **end for**
26: **end for**
27: Let l be an empty list of m-tuples.
28: Let $(t_1, \ldots, t_m)$ be a key in $dp[n]$ with the largest value.
29: **for** $i = n \ldots 1$ **do**
30: Let v be the value of $(t_1, \ldots, t_m)$ in $dp[i]$.
31: **for** $j = 1 \ldots m$ **do**
32: Let $(t'_1, \ldots, t'_m)$ be the tuple we get after subtracting $p[id[i]]$ from t_j in $(t_1, \ldots, t_m)$.
33: Let $v' = v - w[id[i]] \cdot t_j$.
34: Sort this tuple.
35: **if** the value of $(t'_1, ..., t'_m)$ in $dp[i-1]$ is v' **then**
36: Add $(t_1, \ldots, t_m)$ to the head of l.
37: Replace $(t_1, \ldots, t_m)$ with $(t'_1, \ldots, t'_m)$.
38: Continue
39: **end if**
40: **end for**
41: **end for**
42: ▷ The i^{th} tuple in l is the tuple of sorted completion times of all machines in an optimal solution when considering only jobs numbered $id[1 \ldots i]$
43: Assign jobs to the correct machines by simulating the addition and sorting procedure by maintaining a permutation of machine indices at each step.
44: Return the solution so formed.
45: **end function**

For a brief proof of correctness, note the following:

1. The correctness of the order of processing the jobs lies in the fact that in the kind of optimal solution we are interested in, for all machines, there is a maximal prefix which has jobs with numbers among $id[1 \ldots i]$ only.

2. The proof of correctness of the state transitions in dp is straightforward, and can be done using an induction that relies upon the fact that all configurations of machines come from some previous configuration by appending a job to some prefix, and that the increase in objective function in appending a job with processing time t and weight w to a machine with completion time C is precisely $w \cdot (C + t)$, and that we are essentially finding the minimum of the objective function over all 'optimal' ways (for jobs $id[1 \ldots i-1]$) to reach to a candidate optimal parent configuration and then transition to a new configuration.

3. The part of the program from lines 27 to 43 is merely to reconstruct the assignment of the jobs to machines.

For now, we will assume that the times $p[i]$ are all integers. Let $T = \sum_{i=1}^{n} p[i]$. We show that the time taken by the algorithm is polynomial in n and T.

Firstly note that for any $dp[i]$, since all machines have sorted tuples of integers between 0 to T, and each tuple is associated with at most one value, and the sum of tuples is constant (and equals $\sum_{j=1}^{i} p[id[i]]$), the number of elements in $dp[i]$ is $O(T^{m-1})$ (in fact it would be better than this in practice, and would be at most $O(\binom{T+m-2}{m-1})$ which is asymptotically the same but has a better constant factor). The time complexity is strictly dominated by the loop that fills in $dp[1 \ldots n]$. Searching/inserting/deletion in a red-black tree takes $O(m \log |dp[i]|) \in O(m^2 \log T)$ time. The rest of the operations in a single iteration of the loop at line 12 take a total of $O(m \log m)$ time.

Since there are $m \cdot |dp[i-1]|$ iterations for each i, one iteration of the loop at line 10 takes $O(m^3 T^{m-1} \log T)$ time. The time taken over all loops is thus $O(nm^3 T^{m-1} \log T)$, which is polynomial in n and T, keeping in mind that m is a problem-specified constant.

For this problem, my rounding ideas didn't work out (since the error term is proportional to $T \sum_i w_i$ which might be much larger in order of magnitude than **OPT**). However, I tried to scale and round values of each solution into certain intervals similar to those made in bin-packing to get a not-too-large deviation from **OPT**. For that, it is necessary to bound the number of keys we can have for each value, else the algorithm still remains pseudo-polynomial in the input.

Now we need to construct a FPTAS based on this solution. Note that this solution works for $m \leq M = 2$, I wasn't able to solve for $m > M$.

Firstly, we claim that if $m \leq M$, it is possible to impose a total-order on tuples for a given value.

For that, we first claim that for any partition of the remaining jobs $A_1, \ldots, A_m$ into m sets, with the total time of each set on a single machine being $a_1, \ldots, a_m$ respectively, it is optimal to assign the highest a_i to the lowest t_j and so on.

For a proof, note that the final answer is $O + \sum a_i + \sum_{(i,j) \in S} a_i \cdot t_j$ where S is the set of pairs whose first component is the index of the set A_i and second component is the index of the machine to which it is assigned. So by the rearrangement inequality, the result follows.

Consider two tuples $(t_1, t_2, \ldots, t_m)$ and $(t'_1, t'_2, \ldots, t'_m)$ both in sorted order. Suppose $a_1 \geq \ldots \geq a_m$, without loss of generality. Then the objective value taking the first tuple is $\sum_i a_i t_i$ and that for the second tuple is $\sum_i t'_i$. Note that $\sum_i a_i t_i = \sum_i \left((a_i - a_{i+1}) \cdot \sum_{j=1}^{i} t_j \right)$, where $a_{m+1} = 0$.

Note that $a_i - a_{i+1} \geq 0$ for all i. So in order to claim that the first tuple is better than the second tuple, it is sufficient to say that $\sum_{j=1}^{i} t_j \geq \sum_{j=1}^{i} t'_j$ for all i. Since the tuple sums are the same, by subtracting this inequality from the tuple sums, it is sufficient to check if the second tuple majorizes the first tuple (when reversed).

For $m = 2$ this is clear, since we need to only check if $t_2 \leq t'_2$ and $t_1 + t_2 = t'_1 + t'_2$, which is equivalent to a lexicographical ordering on the original tuples. For $m = 3$, this might not be possible since the tuples $(6, 3, 2)$ and $(5, 5, 1)$ are not comparable under the majorization relation.

The case of $m = 1$ was settled earlier. So for $m = 2$, after we have computed for the answer for all tuples, we can remove all tuples other than the lexicographically smallest tuple for each value, and get one tuple per value.

Then ...

3 Problem 3

3.1 Statement

In the *directed Steiner tree* problem, we are given as input a directed graph $G = (V, E)$, non-negative costs $c_{ij} \geq 0$ for arcs $(i, j) \in E$, a root vertex $r \in V$, and a set of terminals $T \subseteq V$. The goal is to find a minimum-cost tree such that for each $i \in T$ there is a directed path from r to i. It is NP-hard to approximate set-cover to a factor better than $c \log n$, for some constant c, where n is the number of elements. Use this fact to argue that for some constant d there can be no $d \log |T|$-approximation algorithm for the directed Steiner tree problem, unless P = NP.

3.2 Solution

We claim that any $d < c$ satisfies this property.

Suppose that there is a $d \log |T|$-approximation algorithm for the directed Steiner tree problem. We shall show that such an algorithm will directly lead to a $d \log n$ algorithm for an instance of set cover with m sets $S_1, \ldots, S_m$ and n elements $e_1, \ldots, e_n$.

1. Construct nodes v_0, $v_1, \ldots, v_m$ and $u_1, \ldots, u_n$ in a graph G.
2. Join v_0 to each v_i with an edge of cost $cost(S_i)$.
3. For a set $S_i = \{e_{j_1}, \ldots, e_{j_k}\}$, join v_i to $u_{j_1}, \ldots, u_{j_k}$ each with edges of cost 0.
4. Solve the directed Steiner tree problem for the graph G with the costs assigned as above, root vertex v_0 and the set of terminals being $\{u_1, \ldots, u_n\}$.
5. For each edge of the form (v_0, v_k), add set S_k into the solution.
6. Return the solution.

Claim 3.1

This solution is a valid set cover.

Proof. Suppose that this is not the case. Then there exists an element e_i which is not covered by the chosen sets. The corresponding vertex u_i is hence not reachable from the root, since any path from v_0 to u_i should pass through a v_j such that e_i is in S_j, and had we chosen any of these, this would had implied that S_j was in the solution, contradicting that e_i is not covered. So u_i is not reachable from r, which is a contradiction to the fact that the algorithm gets a minimum cost tree such that each vertex in the terminal set is reachable from the root v_0. □

Claim 3.2

An optimal solution to the constructed instance of directed Steiner tree corresponds to an optimal solution to the associated set cover problem.

Proof. We claim that for each solution of set cover with cost C, we can construct a solution of this instance of directed Steiner tree with cost C and vice versa.

For the first part, let $S_{i_1}, \ldots, S_{i_k}$ be the sets chosen for the set cover. Then add the edges (v_0, v_{i_r}) and the edges (v_{i_r}, u_j) for each element e_j in S_{i_r}. Note that this is a directed tree since it has no cycles. Now since for each e_r there is a set S_{i_j} such that $e_r \in S_{i_j}$ (by the definition of set cover), there is a path $v_0 \to v_{i_j} \to u_r$ from v_0 to u_r for each r. The cost of this solution is clearly C, since the edges between v_i's and u_j's has contribution 0 to the overall cost, and the contribution of the edge (v_0, v_{i_j}) is $cost(S_{i_j})$, and summing it gives the conclusion.

For the second part, construct any solution for this instance of directed Steiner tree problem which has cost C. Construct the solution in the same way as done in the algorithm above. Since for every vertex u_i, there is a path from v_0 to u_i, there must be a vertex v_j on this path, which corresponds to including the set S_j in the solution. Hence the corresponding solution is a set cover instance. Now note that by a very similar argument as in the previous part, the solution has the same cost.

Hence this shows that the optimal solution to the constructed instance of the directed Steiner tree corresponds to an optimal solution to the associated set cover solution. □

Claim 3.3

This solution is a $d \log n$ approximation to the set cover problem.

Proof. By the assumption on the algorithm used to solve the directed Steiner tree problem, we know that the cost of the solution to the constructed directed Steiner tree instance is at most $d \log n$ times the cost of the optimal solution to the constructed directed Steiner tree instance. By the claim above, and the claim about the cost of the associated set cover solution having the same cost as the solution to the directed Steiner tree instance, we get the conclusion. □

However, since this approximation algorithm is a polynomial time algorithm, and it is NP-hard to approximate set cover to a factor better than $c \log n$, this shows that P = NP.

4 Problem 4

4.1 Statement

The *k-suppliers* problem is similar to the k-center problem discussed in class. The input to the problem is a positive integer k, and a metric on a set of points V, $|V| = n$. However, now the points are partitioned into suppliers $F \subseteq V$ and customers $C = V \setminus F$. The goal is to pick k suppliers such that the maximum distance between a customer and its nearest picked supplier is minimized. In other words, we wish to find $S \subseteq F$, $|S| \leq k$ that minimizes $\max_{j \in C} d(j, S)$ where $d(j, S)$ is the distance between j and the nearest point in S. Give a 3-approximation algorithm for the k-suppliers problem.

4.2 Solution

The solution will closely mirror the one for the k-center problem as done in class.

Our algorithm will use a predicate on a non-negative real r as follows:

1. Initialize the solution set to $\{\}$.
2. While C is non-empty:
   - Pick a vertex v from C, and consider the closest vertex s to it in F. If the distance is more than r, return failure.
   - Otherwise, consider all vertices in C at a distance of at most $3r$ fom s, and remove them from C. Add s to the solution set.
3. Return the solution set.

Claim 4.1

Let r^* be the maximum distance between a customer and its nearest supplier in an optimal solution. If the algorithm returns failure or a solution with $> k$ suppliers, then $r < r^*$, otherwise it returns a solution with at most k suppliers, with the maximum distance between a customer and a supplier being at most $3r$.

Proof. We break the proof into two parts. The second part is obvious, since the solution is valid, and the vertices removed all have a distance of $\leq 3r$ from the supplier chosen in that step, so the closest supplier to them in the finally chosen set of suppliers has a distance at most $3r$ as well.

For the first part, there are two cases:

1. The algorithm returns a failure: In this case, there is a vertex which is at a distance of $> r$ from its nearest supplier. Hence $r^* > r$.
2. The algorithm returns a solution with $> k$ suppliers: Suppose for the sake of contradiction that there is indeed a solution where the maximum distance between a customer and its nearest supplier is at most r, and the number of suppliers chosen is at most k. We show that the minimum number of suppliers that need to be chosen in any solution with maximum distance between a customer and supplier being $\leq r$ is $> k$, which will lead to the desired contradiction.

   Call any solution with the maximum distance between a customer and a supplier being $\leq r$ an $r-$covered solution.

   To this end, we show that for any r-covered solution S, we can assign a unique supplier belonging to this solution to each of the suppliers in the solution our algorithm returns.

   Firstly note that at each step, we choose a different supplier, since we always clear out all vertices which are at a distance of at most $3r$ from that supplier, so any other vertex chosen will be at a distance of $> 3r$ from this supplier, and hence this supplier won't be chosen again.

   Note that at each step, we choose a vertex v corresponding to a new customer. In S, there must be a supplier s' for this vertex. We will associate this supplier to the supplier s chosen at this step.

   Suppose that this supplier is chosen in two distinct steps (i.e., associated to two distinct suppliers picked by the algorithm), where the customers chosen were u and v respectively. Without loss of generality, suppose v was picked before u. Then we have $d(u, s) \leq d(u, s') + d(s', v) + d(v, s) \leq 3r$, where the first two terms are bounded using the fact that S is an r-covered solution, and the last term

arises from the choice of s when v was picked. This inequality implies that u is indeed removed when s is chosen as the supplier, and this is a contradiction.

So we have shown a surjective partial function from the set of suppliers chosen in S to the set of suppliers chosen by the algorithm. Hence the number of suppliers chosen in S is at least the number of suppliers chosen by the algorithm, which is $> k$. Hence for any r-covered solution, the number of suppliers in it is more than k. Now note that by our assumption, there is a solution where the maximum distance between a customer and a supplier is at most r and the number of suppliers chosen is at most k. This is a clear contradiction.

Now if $r \geq r^*$, any r^*-covered solution is also r-covered, hence it must have more than k suppliers. This applies to the optimal solution of the original problem as well, so there should be no solution to the original problem in this case, which is a contradiction. Hence we must have $r < r^*$, since there exists at least one solution to the problem.

This completes the proof of the first part of the claim as well, and we are done. □

Call the previous algorithm SolveOrFail(r).

The final algorithm becomes the following:

1. For each possible distance r between suppliers and customers (sorted in increasing order):
   - Call SolveOrFail(r).
   - If it returns failure, continue.
   - Else, return the solution it returns.

Claim 4.2

The optimal solution of the problem is equal to one of the distances between the suppliers and the customers.

Proof. This is true since r^* is in fact a distance between a customer and a supplier by definition in the previous claim. □

Claim 4.3

The algorithm always returns a 3-approximation for the k-suppliers problem.

Proof. Note that there is always an answer: just consider a single supplier; then the answer is in the set of distances between a customer and a supplier, and it is upper bounded by the maximum distance of this supplier from a customer.

Now consider the first r where the algorithm doesn't give a failure. Since no previous distance returned a solution, we must have $r^* \geq r$. However, since the solution returned has the distance between any customer to its nearest supplier at most $3r$, the value of $\max_{j \in C} d(j, S)$ is $\leq 3r \leq 3r^*$, which is 3 times the optimum. Thus, this is a 3-approximation, as needed. □

Claim 4.4

This is a polynomial-time algorithm.

Proof. Note that there are at most $O(n^2)$ distinct distances, and sorting would take roughly $O(n^2 \log n)$ (a better bound can be found by considering the size of F etc., but the main goal is just to show that this is a polynomial time algorithm). At each step in the function SolveOrFail, we do $O(n)$ work to find the closest supplier to a customer, and remove all customers that are at a distance of at most $3r$ from this supplier. So overall, this algorithm is $O(n^3)$, which is polynomial time in the problem size. □