COMP 251 - Fall 2017 - Assignment 3

Due: 11:59pm Nov 3rd

**General rules:** In solving these questions you may consult your book; You can discuss high level ideas with each other, but each student must find and write his/her own solution. You should upload the pdf file (either typed, or a clear scan) of your solution to mycourses.

1. (15 points) We are given a tree $T$ (not necessarily binary). Design a greedy algorithm that finds the largest number of nodes in $T$ such that no two of them are adjacent. (You do not need to optimize the running time of your algorithm, as long as your running time is polynomial).

**Solution:**

**key observation.** If node $v$ is a leaf, there exists a max cardinality independent set containing $v$.

**proof:** Consider a max cardinality independent set $S$. If $v \in S$ we are done. Else, let $(u, v)$ be some edge.

(a) if $v, u \notin S$, then $S \cup v$ is a bigger independent set. "CONTRADICTION"

(b) if $u \in S$, then $S \cup v - u$ is still the maximum independent set.

Using the above theorem, it makes sense to recursively include leaves in the independent set:

<table>
<thead>
<tr>
<th>Algorithm 1: Maximum Independent Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \leftarrow \emptyset$</td>
</tr>
<tr>
<td>while Graph $G$ has at least one edge do</td>
</tr>
<tr>
<td>$e \leftarrow (u, v)$ v is leaf.</td>
</tr>
<tr>
<td>$S \leftarrow S \cup v$</td>
</tr>
<tr>
<td>$G \leftarrow G - e$ it removes nodes $v, u$ and all their incident edges.</td>
</tr>
<tr>
<td>close;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return $S$</td>
</tr>
</tbody>
</table>

2. (15 points) We have a single processor, and we are given a sequence of jobs with processing times $t_1, \ldots, t_n$. Each job has to be processed in a continuous time interval, and these intervals cannot overlap. We want to minimize the average finishing times of these jobs. Prove that the optimal strategy is to process them in increasing order according to their processing times.

**Solution:** Let’s assume that ordering the jobs in increasing order according to the processing time ($t_i$ for job $i$) is not the optimal solution and call this solution $S$. Then there must exist a solution $O$ that is optimal and for which the ordering is different than $S$.

Since we know that the increasing order $S$ has that $t_i < t_j \forall i, j$ where $i < j$, then there must exist in $O$ at least one adjacent couple $i, j$ where $t_i > t_j$ and $i < j$. The finishing times of jobs $i, j$ is the following:
Where $T$ is the earliest starting time between job $i$ and $j$. Let’s consider swapping those two elements and name this solution $O’$. We note that by doing so, the finishing time of all the other jobs, before and after, is not altered. The new finishing times of jobs $i, j$ are:

\[ f’_j = T + t_j, \quad f’_i = T + t_i + t_j \]

We notice that $f’_i = f_j$, however $f’_j < f_i$ since $t_j < t_i$. If $O$ is the optimal solution, we have proven that $O’$ not less optimal, yet $O’$ is closer to $S$ than $O$. Therefore, by repeating these inversions we can see that $O \geq O’ \geq O’’ \geq \ldots \geq S$, which proves that $S$ is also optimal.

3. (15 points) Consider the same setting as the previous question, but now we have 3 processors instead of 1. Each job has to be processed in a continuous time interval on one of the processors, and the intervals on each one of the processors cannot overlap. Again we want to minimize the average finishing times. Is the greedy algorithm still optimal? In the greedy algorithm, every time a processor becomes available, among the unprocessed jobs we assign the one that has the smallest processing time to that processor. Either prove that this algorithm is optimal or give a counter-example showing that it does not always minimize the average of the finishing times.

**Solution:**
Yes, it is. Here is the proof:
Assume we have $n$ jobs: $j_1, j_2, j_3, \ldots, j_n$. These jobs are ordered from the smallest to the largest processing time. Let’s assume $n = 3k$ for some $k$ (the other two cases are very similar to prove by adding minor changes). Let’s call the processors A, B, C. Without loss of generality, assume the algorithm gives $j_1$ to A, $j_2$ to B and $j_3$ to C. The set of jobs assigned to the processors A, B, C until the end of the task are called $S_A, S_B, S_C$ respectively. Due to the characteristics of the greedy algorithm, which assigns a job immediately to the next free processor, it is easy to check that at the end of the task we have:

- $S_A = j_1, j_4, \ldots, j_{3k-2}$
- $S_B = j_2, j_5, \ldots, j_{3k-1}$
- $S_C = j_3, j_6, \ldots, j_{3k}$

Now, the average finishing time of the tasks are going to be:

\[
AVG = \frac{S_A + S_B + S_C}{3k} = \frac{\sum_{i=0}^{k-1} (k-i) \times t(j_{3i+1}) + \sum_{i=0}^{k-1} (k-i) \times t(j_{3i+2}) + \sum_{i=1}^{k} (k-i+1) \times t(j_{3i})}{3k}
\]

In the above formula, the syntax $t(j)$ is the time it takes to finish job $j$. As it can be seen, the algorithm has given the largest weight to the smallest time consuming jobs and ordered the weights accordingly. This is exactly what we expect from an optimal algorithm to do. Note that, as it was shown in the question 2, if we switch any task inside a processor of greedy algorithm, it is going to increase the overall finishing time. Switching the tasks between processors is also increases the finishing time, as the algorithm does not allocate a job to a processor as soon as it becomes available. This delay will add extra term to the nominator of AVG. So, the greedy algorithm is optimal.

4. (15 points) We are given an undirected graph $G = (V, E)$ as an input, and our goal is to find the minimum of vertices whose deletion will remove all the edges in $G$ (when we delete a vertex, the edges incident to it will be removed). Is the following greedy algorithm optimal?
• While there are still edges left:
  • pick a vertex with current maximum degree and remove it.
• endwhile

Solution: It is not optimal. The following counter-example shows this

5. (20 points) We are given the coordinates of \( n \) points on the plane: \((x_1, y_1), \ldots, (x_n, y_n)\). Give a polynomial time algorithm that finds the smallest circle that is centered at one of these points and contains at least half of the points.

Solution: Note that one of the \( n \) points will be on the perimeter of the optimal circle. Hence for every two points \( i, j \) and we check whether the circle centered at \( i \) that passes through the point \( j \) (radius is distance between \( i \) and \( j \)) contains at least half of the points. Among all such circles we output the one that has the smallest radius.

6. (20 points) Let \( G = (V, E) \) be an undirected weighted graph, and let \( F \subset E \) be a collection of edges in \( G \) that contain no cycles (i.e. they form a forest). Design an efficient algorithm to find a spanning tree in \( G \) that contains all the edges in \( F \) and has minimum cost among all such spanning trees.

The algorithm is basically by initializing Kurskal’s algorithm to include all the edges in \( F \) and then running it to add more edges until we obtain a tree. To be more precise, we include all the edges in \( F \), and then we consider the remaining edges (i.e. the edges in \( E - F \)), one by one, in an increasing order according to their weights, and add them to the tree if they do not create a cycle.

To prove that this algorithm works correctly, we can change the costs of the edges in \( F \) to 0, and obtain a new weighted graph \( H \). Now what is described above is indeed just a normal execution of Kruskal’s algorithm on \( H \), and thus finds an MST \( T \) in \( H \). Note that if \( R \) is a spanning tree in \( G \) that contains all the edges in \( F \), then

\[
\text{cost}_G(R) = \text{cost}_H(R) + \sum_{e \in F} c_e,
\]

where \( c_e \) denotes the cost of an edge \( e \), and \( \text{cost}_H \) and \( \text{cost}_G \) denote the costs of a spanning tree in \( H \) and \( G \), respectively. This equation implies that since \( T \) is a minimum spanning tree in \( H \), it also has the smallest cost among all spanning trees in \( G \) that contain all the edges in \( F \).