Problem Set 1

> *We are given as input a set of n requests (e.g., for the use of an auditorium), with a known start time s_i and finish time t_i for each request i. Assume that all start and finish times are distinct. Two requests conflict if they overlap in time — if one of them starts between the start and finish times of the other. Our goal is to select a maximum-cardinality subset of the given requests that contains no conflicts. (For example, given three requests consuming the intervals [0,3], [2,5], and [4,7], we want to return the first and third requests.) We aim to design a greedy algorithm for this problem with the following form: At each iteration we select a new request i, including it in the solution-so-far and deleting from future consideration all requests that conflict with i.*
>
> *Which of the following greedy rules is guaranteed to always compute an optimal solution?*
>
> - *At each iteration, pick the remaining request with the earliest finish time.*
> - *At each iteration, pick the remaining request which requires the least time (i.e., has the smallest value of $t_i - s_i$) (breaking ties arbitrarily).*
> - *At each iteration, pick the remaining request with the earliest start time.*
> - *At each iteration, pick the remaining request with the fewest number of conflicts with other remaining requests (breaking ties arbitrarily).*

ANSWER: Consider the following requests:

```
i | start | finish | duration
-----------------------------
1 | 0     | 3      | 3
2 | 2     | 4      | 2
3 | 3     | 6      | 3
```

If we pick the request with the smallest $t_i - s_i$, we can only pick r_1, whereas just eyeballing shows that the maximum subset should contain r_1 and r_3. Therefore, option 2 is incorrect.

Consider the following requests:

```
i | start | finish
-----------------
1 | 0     | 6
2 | 2     | 4
3 | 3     | 6
```

If we pick the request with the earliest time (r_1), we cannot pick any from the table above, but if we somehow don't pick r_1, we can pick either one of r_2 or r_3. Therefore, option 3 is incorrect.

It is shown here (https://walkccc.github.io/CLRS/Chap16/16.1/#161-3) that option 4 is incorrect as well, but I honestly don't understand how they came up with such an example, so I'm going to refrain from copy-pasting it here.

Option 1 is correct. To prove it, let S be a set of requests and $r_m \in S$ a request with the earliest finish time. Let $R \subseteq S$ be the maximum subset of mutually compatible requests, and $r_k \in R$ a request with the earliest finish time. Clearly, $finish(r_m) \leq finish(r_k)$. If they are equal, we are done with the proof. Otherwise, let $R' = R - \{r_k\} \cup \{r_m\}$. Since r_k is the request with the earliest finish time in R, and $finish(r_m) < finish(r_k)$, therefore, none of the other requests in R conflict with r_m, and hence $|R'| = |R|$. Therefore, the maximum subset R contains r_m, the request with the earliest finish time.

> *We are given as input a set of n jobs, where job j has a processing time p_j and a deadline d_j. Recall the definition of completion times C_j from the video lectures. Given a schedule (i.e., an ordering of the jobs), we define the lateness l_j of job j as the amount of time $C_j - d_j$ after its deadline that the job completes, or as 0 if $C_j \leq d_j$. Our goal is to minimize the maximum lateness, $\max_j l_j$.*
>
> *Which of the following greedy rules produces an ordering that minimizes the maximum lateness? You can assume that all processing times and deadlines are distinct.*
>
> - *None of the other answers are correct.*
> - *Schedule the requests in increasing order of deadline d_j*
> - *Schedule the requests in increasing order of processing time p_j*
> - *Schedule the requests in increasing order of the product $d_j \times p_j$*

ANSWER: Let S be a sequence in increasing order of processing time p_j. Let the position of the job with the maximum lateness be j. If we swap job at j with the one at $j - 1$, since $C_{j-1} < C_j$, we get an even lower value for lateness. Thus, option 3 is incorrect.

By similar argument, option 4 is incorrect.

Let S be a sequence in increasing order of deadline d_j. If we swap job at j with the one at $j-1$, the lateness for the new job at position j increases. Thus, the swap makes the sequence worse, and hence, the sequence is optimal to begin with.

> *Consider an undirected graph $G = (V, E)$ where every edge $e \in E$ has a given cost c_e. Assume that all edge costs are positive and distinct. Let T be a minimum spanning tree of G and P a shortest path from the vertex s to the vertex t. Now suppose that the cost of every edge e of G is increased by 1 and becomes $c_e + 1$. Call this new graph G'. Which of the following is true about G'?*
>
> - *T may not be a minimum spanning tree and P may not be a shortest $s - t$ path.*
> - *T may not be a minimum spanning tree but P is always a shortest $s - t$ path.*
> - *T is always a minimum spanning tree and P is always a shortest $s - t$ path.*
> - *T must be a minimum spanning tree but P may not be a shortest $s - t$ path.*

ANSWER: Since Prim's MST algorithm picks the edge with the lowest cost at each iteration, increasing the cost of every edge by 1 wouldn't change anything. However, the shortest path may change. Consider the graph $D \overset{7}{\longleftarrow} A \overset{2}{\longrightarrow} B \overset{2}{\longrightarrow} C \overset{2}{\longrightarrow} D$. The shortest path from A to D is $A \rightarrow B \rightarrow C \rightarrow D$. Adding 1 to every edge gives us the new graph $D \overset{8}{\longleftarrow} A \overset{3}{\longrightarrow} B \overset{3}{\longrightarrow} C \overset{3}{\longrightarrow} D$. Now $A \rightarrow B \rightarrow C \rightarrow D$ is no longer the shortest path between A and D. Thus, option 4 is correct, and the other ones are not.

> *Suppose T is a minimum spanning tree of the connected graph G. Let H be a connected induced subgraph of G. (I.e., H is obtained from G by taking some subset $S \subseteq V$ of vertices, and taking all edges of E that have both endpoints in S. Also, assume H is connected.) Which of the following is true about the edges of T that lie in H? You can assume that edge costs are distinct, if you wish. [Choose the strongest true statement.]*
>
> - *For every G and H, these edges form a minimum spanning tree of H*
> - *For every G and H, these edges are contained in some minimum spanning tree of H*
> - *For every G and H and spanning tree T_H of H, at least one of these edges is missing from T_H*
> - *For every G and H, these edges form a spanning tree (but not necessary minimum-cost) of H*

ANSWER: Consider a triangle graph $G = C \overset{3}{\longleftarrow} A \overset{1}{\longrightarrow} B \overset{1}{\longrightarrow} C; \therefore T = (A, B), (B, C)$. If $H = (A, C), T \cap H = \emptyset$, which rules out options 1 and 4.

Option 2 is correct. To prove it, let's establish the *Light-Edge Property* of a MST.

Light-Edge Property: *Let G = (V, E) be a connected undirected weighted graph with distinct edge weights. For any cut of G, the minimum weight edge that crosses the cut is in the minimum spanning tree T of G.*

Proof: Suppose $e(v, w) \in E, e \notin T$ be the minimum weight edge. If we take a cut (A, B) s.t. $v \in A, w \in B$, then there must be another edge $e' \in T, e' \neq e$ that connects v and w (since by definition T is a connected subgraph). Let $T' = T - \{e'\} \cup \{e\}$; since, $weight(e') > weight(e), weight(T') < weight(T)$. Since T is a MST, this brings us to a contradiction, and hence $e \notin T$ cannot be true.

Back to the original question, suppose T' be a MST of H and an edge $e \in T \cap H, e \notin T'$. Arguing on the same lines as the *Light-Edge Property*, we can show that this contradicts the assumption T is a MST, and e must be in T'. Since e is an arbitrary edge, all edges in $T \cap H$ must be included in T'.

Since option 2 is correct, option 3 is not.

> *Consider an undirected graph $G = (V, E)$ where edge $e \in E$ has cost c_e. A minimum bottleneck spanning tree T is a spanning tree that minimizes the maximum edge cost $\max_{e \in T} c_e$. Which of the following statements is true? Assume that the edge costs are distinct.*
>
> - *A minimum bottleneck spanning tree is not always a minimum spanning tree and a minimum spanning tree is not always a minimum bottleneck spanning tree.*
> - *A minimum bottleneck spanning tree is always a minimum spanning tree and a minimum spanning tree is always a minimum bottleneck spanning tree.*
> - *A minimum bottleneck spanning tree is always a minimum spanning tree but a minimum spanning tree is not always a minimum bottleneck spanning tree.*
> - *A minimum bottleneck spanning tree is not always a minimum spanning tree, but a minimum spanning tree is always a minimum bottleneck spanning tree.*

ANSWER: Recall the *Light-Edge Property*; it implies that every other spanning tree has maximum edge cost at least as large. Therefore, a MST is also a minimum bottleneck spanning tree. But the reverse is not true; to see that, consider the graph

$G = C \xleftarrow{4} A \xrightarrow{4} B \xrightarrow{-1} C; MST(G) = \{(A, B), (B, C)\}$. Consider the spanning tree $\{(A, B), (A, C)\}$; it doesn't increase the bottleneck (4), and hence is a minimum bottleneck spanning tree, but isn't a MST.

> *In this problem you are given as input a graph $T = (V, E)$ that is a tree (that is, T is undirected, connected, and acyclic). A perfect matching of T is a subset $F \subset E$ of edges such that every vertex $v \in V$ is the endpoint of exactly one edge of F. Equivalently, F matches each vertex of T with exactly one other vertex of T. For example, a path graph has a perfect matching if and only if it has an even number of vertices.*
>
> *Consider the following two algorithms that attempt to decide whether or not a given tree has a perfect matching. The degree of a vertex in a graph is the number of edges incident to it. (The two algorithms differ only in the choice of v in line 5.)*

Algorithm A:

```
1 While T has at least one vertex:
2   If T has no edges:
3     halt and output "T has no perfect matching."
4   Else:
5     Let v be a vertex of T with maximum degree.
6     Choose an arbitrary edge e incident to v.
7     Delete e and its two endpoints from T.
8 [end of while loop]
9 Halt and output "T has a perfect matching."
```

Algorithm B:

```
1 While T has at least one vertex:
2   If T has no edges:
3     halt and output "T has no perfect matching."
4   Else:
5     Let v be a vertex of T with minimum non-zero degree.
6     Choose an arbitrary edge e incident to v.
7     Delete e and its two endpoints from T.
8 [end of while loop]
9 Halt and output "T has a perfect matching."
```

> *Is either algorithm correct?*
>
> - *Algorithm A always correctly determines whether or not a given tree graph has a perfect matching; algorithm B does not.*
> - *Neither algorithm always correctly determines whether or not a given tree graph has a perfect matching.*
> - *Both algorithms always correctly determine whether or not a given tree graph has a perfect matching.*
> - *Algorithm B always correctly determines whether or not a given tree graph has a perfect matching; algorithm A does not.*

ANSWER: The algorithms are understated: When a vertex is deleted, *all* edges incident to that vertex are also deleted, otherwise we're left with dangling edges.

Consider the path tree $A - B - C - D$. Algo A chooses vertex B, and say, edge (B, C). After deleting vertices B and C, and all the edges, we're left with vertices A and D with no edges. Algo A halts, and incorrectly declares that T doesn't have perfect matching (it does, $\{(A, B), (C, D)\}$).

Lets prove the correctness of algo B by induction. For $|V| = 1$, it correctly identifies that T doesn't have perfect matching. For $|V| = 2$, it correctly identifies that T has perfect matching. Clearly, for T to have perfect matching, $|V|$ must be even. Assume algo B works for $|V| = k$. Let us add a new vertex v and a new edge (v, w) to T, where w is some existing vertex in T. Clearly, for T to have perfect matching, it must include edge (v, w) (since that's the only way to match vertex v). By construction, algo B picks v and removes edge (v, w) in some iteration, leaving $|V| = k - 1$, which by inductive hypothesis, is correctly solved by algo B.

Therefore, algo B is correct.

COMMENTS