CENG567: Homework #3

Yiğit Sever

December 2, 2020

1 Job Scheduling

First we will state the variables we will use. A job $n_i \in n$ is split into preprocessing $p_i \in p$ and indexing $f_i \in f$ per the question text. With the given question text, we will ignore any kind of utilisation or efficiency concerns. In other words, we will not care about the number of PCs we employ nor the time they will stay idle. So, every indexing job $f_i \in f$ will be run on a separate PC.

Since we cannot pipeline the preprocessing jobs $p_i \in p$, any kind of real choice we have will be done on the $f_i \in f$. The trivial case is when the preprocessing jobs are the "bottleneck" of the system, where;

$$\forall p_i \in p > \forall f_i \in f$$

The completion time in this case is minimized by sorting the $f_i \in f$ and placing the *shortest* f_i last.

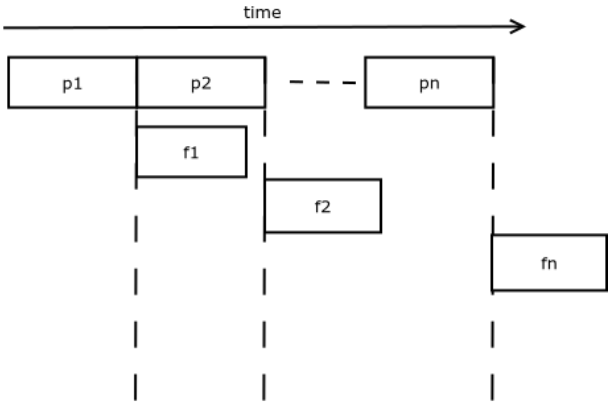


Figure 1: The trivial case where preprocessing jobs are all longer than the indexing jobs

This approach extends to the case where the preprocessing time can be shorter than the indexing time as well. In the example given in Figure 2, two preprocessing jobs p_1 and p_2 both take 1 unit of time. Whereas, the indexing jobs tied to the preprocessing jobs f_1 and f_2 take 5 and 1 units of time, respectively. By scheduling the preprocessing task that is tied to the longest indexing job p_1 first, we can finish the whole computation in 6 time units which takes 7 time units in the other case.

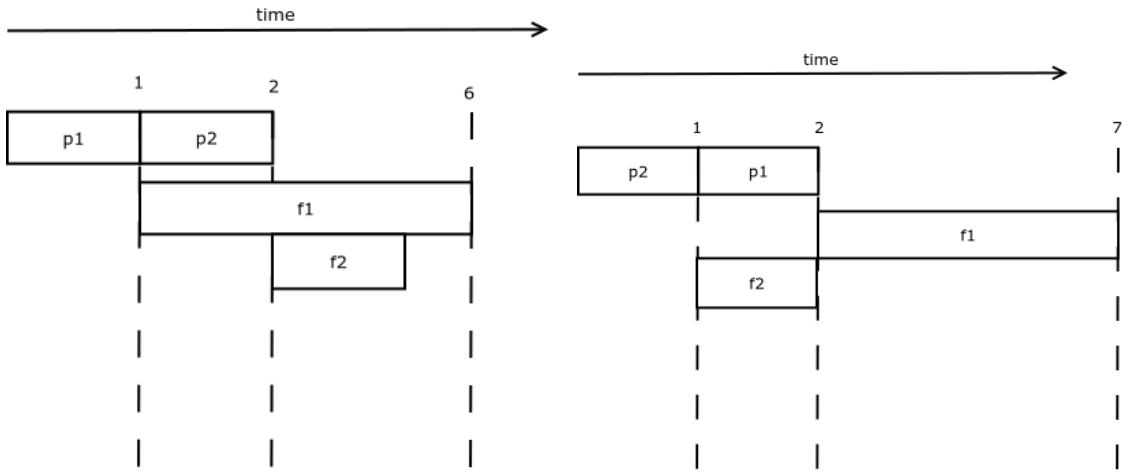
Formally, the algorithm we propose simply sorts the $f_i \in f$ in $\mathcal{O}(n \log(n))$ time and schedules the jobs with respect to $f_i \in f$ (so the $p_i \in p$ tied to $f_i \in f$) from longest to shortest.

2 Spanning Tree

In this question, leveraging the discussion on the lecture forum, we know that there are $n = |E|$ edges in $G = (V, E)$ in which $k + \delta$ edges are red and the rest are white. k is not given as an input in the question text but we will assume so to continue with the discussion, otherwise the algorithm cannot be properly stated.

Given $G = (V, E)$ where $r \leq |E|$ edges are red, can we find k such edges that form a spanning tree.

We will start by running Kruskal's algorithm on G with a modification; we will assign a positive weight γ on every *white* edge and 0 on every *red* edge. Kruskal's algorithm will pick only red edges as a result. This will either;



(a) The longest task is scheduled first

(b) The longest task is scheduled last

Figure 2: The worked example of the cases

- yield a MST with less than k edges. In this case, we can state no such tree with k red edges exists and report this per case (ii) in the question text.
- yield a MST with *exactly* k edges, which we can return immediately per case (i) in the question text.
- yield a MST with more than k edges, which we will have to continue operating with.

Since we are continuing with the algorithm, we can name the minimum spanning tree we got from the steps above T_r.

At this point, we should prepare another MST using only white edges. Since the MST we are aiming to return at the end of the algorithm does not have to consists of only red edges, we can substitute white edges with respect to either Cut Property or Cycle Property as given in the Greedy Algorithms II lecture notes.

We will prepare a MST with only white edges by setting the weights of white edges on G to 0 and red edges on G to β and running Kruskal's algorithm on G. Kruskal's algorithm will pick only white edges as a result and we will call this tree T_w. If T_w does not contain at least $k - E'$ for $T_r = (V, E')$ edges we will conclude that no such tree exists and report per case (ii).

Now, with T_r and T_w, we are aiming to reduce the number of red edges in T_r to k by removing a red edge from it and connecting the tree back using a white edge from T_w. Since both T_r and T_w are MST, they have $V - 1$ edges, removing any edge will disconnect the tree and adding any edge will introduce a cycle, per the definition of trees.

Our algorithm is then as follows;

1. Remove an edge $e \in E(T_r)$ that is not in T_w.
2. We can add e to T_w but this will introduce a cycle (per above). However, we can pick an edge in the newly introduced cycle in T_w that is not in T_r and add that edge e' to T_w. We are able to do this since trees cannot contain cycles hence at least one of the edges in the newly introduced cycle has to be missing from T_r.
3. When we have k red edges (and $V - k - 1$ white edges) in T_r, then we can report it per case (i).

The cycle checking step, step 2 in the algorithm above uses the Union Find. Since we did not go into the detail of Union Find in the lectures we will use the naive approach which uses $\mathcal{O}(n^2)$ operations. On top of that, we have to run Kruskal's algorithm twice for T_r and T_w which runs in order $\mathcal{O}(k \log(n))$ (k can be substituted with the number of white edges, if higher). Overall, the runtime of the algorithm is $\mathcal{O}(n^2 k \log(n))$.

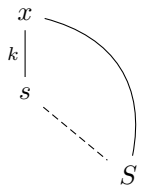
3 Shortest-paths and min spanning trees

3.1 a

We will assume the tree of shortest paths is a spanning tree (otherwise node s on its own is a tree and the answer is trivial).

For a graph $G = (V, E)$, we can have $T = (V, E')$ which is a tree of shortest paths and a minimum spanning tree $T' = (V, E'')$. Take any edge incident to s and call it k. $k = (s, x)$ is the minimum weighted edge from s to x per definition of the tree of shortest paths. Suppose that the MST T' does not include k. This implies that there exists a path in T' $s \rightsquigarrow x$ that does not include k. Adding k to the MST will introduce a cycle but we can break this cycle by using the Cycle property given in the Greedy Algorithms II lecture notes. Since $k = (s, x)$ is the minimum weighted edge, it will not get deleted which means MST should have included k to begin with.

So it is not possible for a tree of shortest paths and a MST to not share any edges.



3.2 b

Given the acyclic digraph $G = (V, E)$ and an arborescence $A \subset E$ with the given conditions in the question text, A itself is then a minimum cost arborescence in G. Take any $e = (v, t) \in A$. If e is part of some minimum cost arborescence in G then it is the cheapest incoming edge for t per the lecture notes. A is built up from such e and the algorithm for building an arborescence in G per the lecture notes is picking the cheapest incoming edge for every node and the graph G is acyclic meaning that we cannot form accidental cycles by including every $e \in E$, A is a minimum-cost arborescence in G.