CSE 551 Homework 2 Solutions

Sep 2021

1 Question 1

Consider the proof of correctness for Dijkstra's Algorithm. If we allow negative weights in a graph, does the algorithm still find the shortest path? If not, where does the proof of correctness fail?

Solution: Dijkstra's algorithm does not necessarily find the shortest path when negative weights are allowed. Let the set of visited nodes in Dijkstra's algorithm be denoted S. J. Kleinberg and E. Tardos argue the following

Proof. We prove this by induction on the size of S. The case $|S| = 1$ is easy, since then we have $S = s$ and $d(s) = 0$. Suppose the claim holds when $|S| = k$ for some value of $k \geq 1$; we now grow S to size $k + 1$ by adding the node v. Let (u, v) be the final edge on our $s - v$ path P_v.

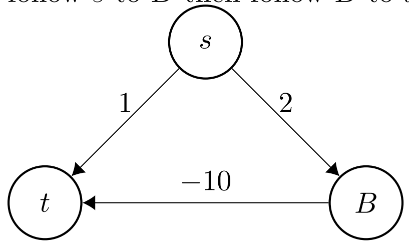
By the induction hypothesis, P_u is the shortest $s - u$ path for each $u \in S$. Now consider any other $s - v$ path P; we wish to show that it is at least as long as P_v. In order to reach v, this path P must leave the set S somewhere; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y. ☐

The key is that immediately after this, they argue

> ... the crux of the proof is very simple: [any other $s - v$ path] P cannot be shorter than P_v because it is already at least as long as P_v by the time it has left the set S.

but P can be shorter if there is a negative weight as one can actually decrease the current weight by following more edges. Figure 1 illustrates a graph where Dijkstra's algorithm fails.

Figure 1: Dijkstra's algorithm may fail to find a shortest $s - t$ path when negative weights are allowed. When running Dijkstra's algorithm on this graph, S is initialized to $\{s\}$. t is the next node added to S and the shortest path so far from s to t has a distance of 1. Next, B is added to S and the distance of the path from s to B is updated to 2. The algorithm does not consider any more edges because $S = V$ and Dijkstra's algorithm gives that going from s to t is the best $s - t$ path. However, the optimal path is to follow s to B then follow B to t.



2 Question 2

We have shown, given a connected graph G, where every edge cost is distinct, if you take any subset of nodes S such that $\varnothing \subset S \subset V$ and if you consider the minimum cost edge e with one vertex in S and the other vertex in $V(G) - S$, every minimum spanning tree contains the edge e. Consider the following argument:

> Let T be a spanning tree that does not contain e. Since T is a spanning tree, it must contain an edge f with one end in S and another $V(G) - S$. Then $T' = T - \{f\} \cup \{e\}$ is a spanning tree with a smaller cost than T as $c_e < c_f$.

Does this argument work? Why or why not?

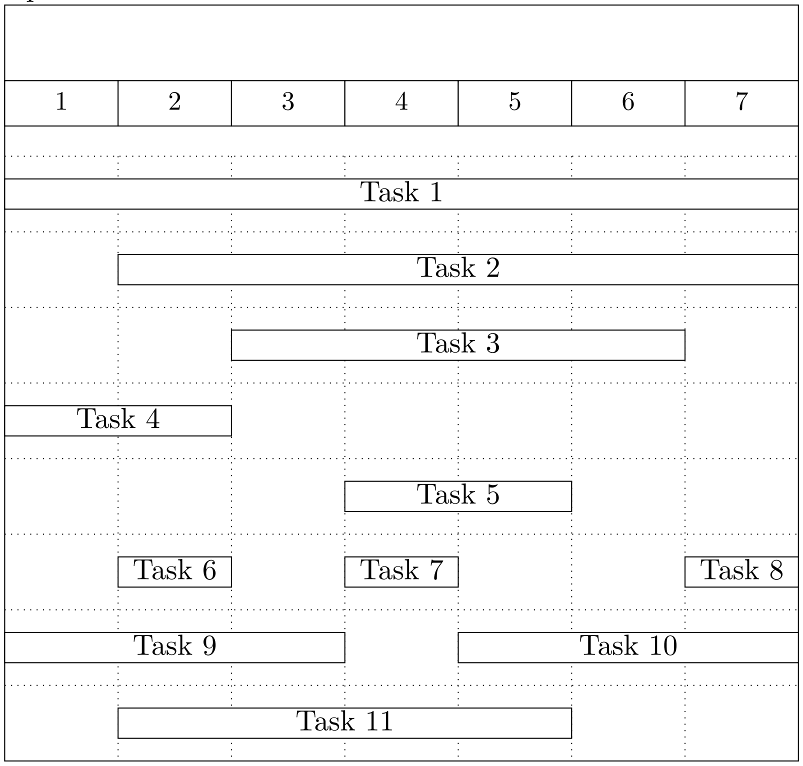
Solution: This is mentioned in the book on page 145. We are not gaurenteed to have a spanning tree if we swap f with e. You must illustrate that the swap between f and e (1) does not disconnect the graph, and that (2) the graph remains acyclic. Removing f may, for instance, disconnect the tree. (See Figure 4.10 page 146 for an example).

As a remark, this is why, in order to swap the two paths e' and $e = (v, w)$, it is essential to show that there is a path from v to w going through e'. Because if there is a path, swapping the edges does not disconnect any components.

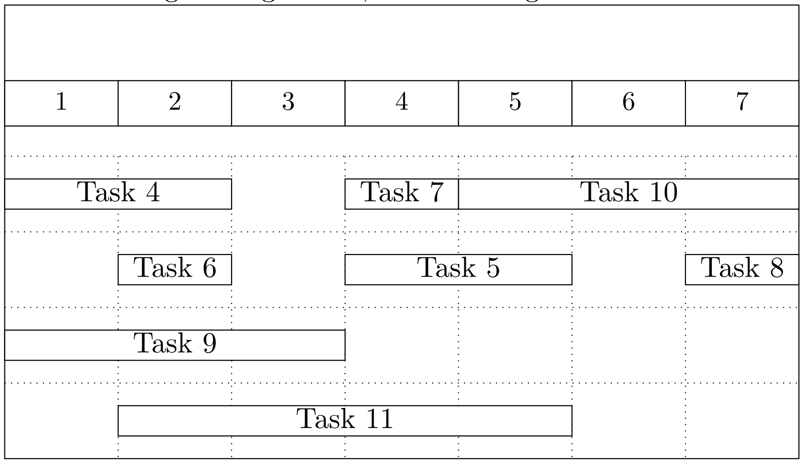
3 Question 3a

Consider arranging all jobs in non-decreasing order of their finish times. Select all jobs (in sorted order) that are compatible with each other starting from the first job. Assign these jobs to the first core and mark them as to be executed. Next, select all jobs (in sorted order) that are compatible with each other starting from the first job that has yet to be executed. Assign these jobs to the second core and mark them as to be executed. Repeat this process for all n cores. Does this algorithm ensure the maximum number of jobs are completed? If so, justify your answer. If not, provide a counterexample.

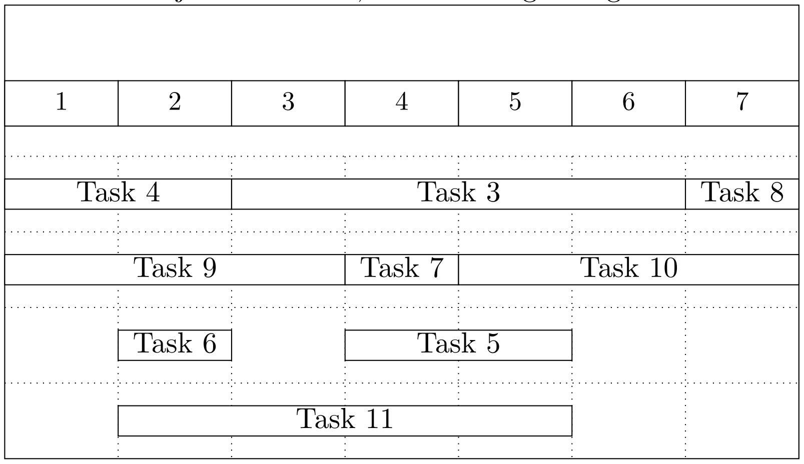
Solution: The algorithm does not ensure the maximum number of jobs are completed. Here a counter example:



After running the algorithm, the resulting chart is:



which runs 8 jobs. However, the following configuration runs 9 jobs:

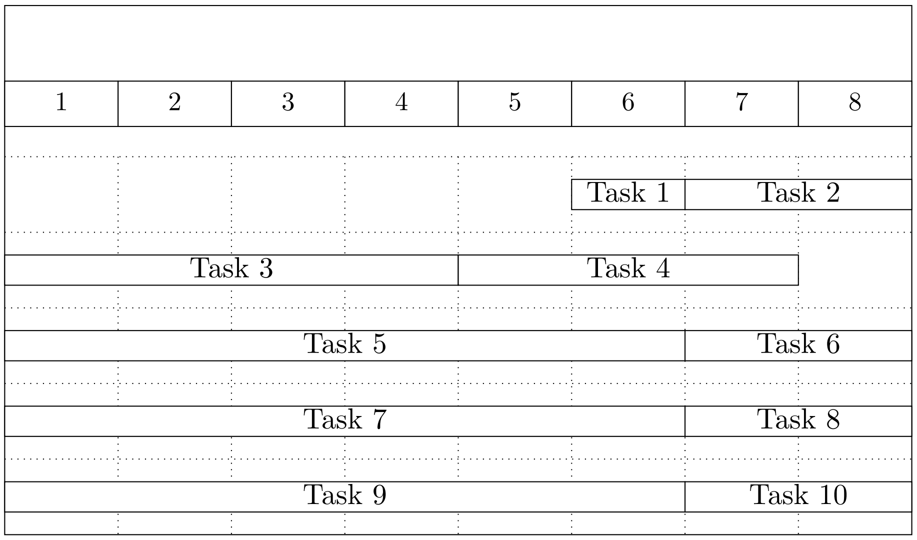


4 Question 3b

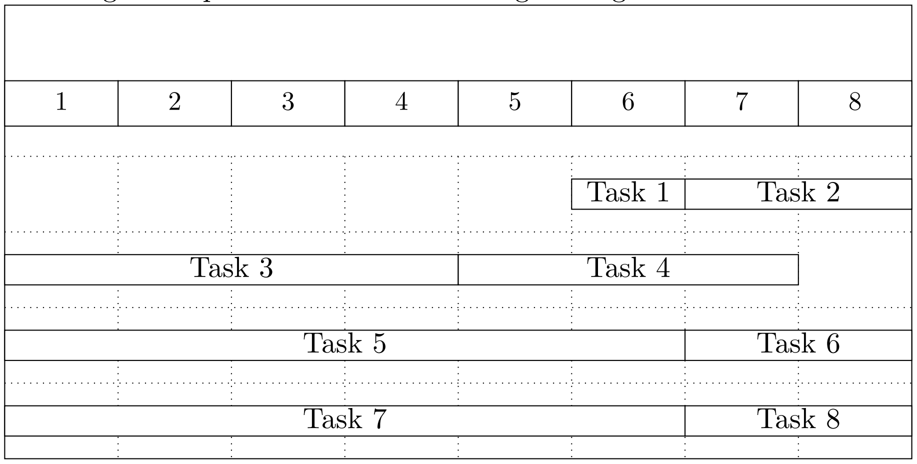
Interval partitioning gives the minimum number of cores required to run all jobs in the queue. In practice, we may only have k cores, but n ($n > k$) cores are required to run every task. In order to run as many jobs as possible, the following scheme is proposed. Use the interval partition scheme described in class to compute the minimal number of cores required to complete all the jobs. Sort the cores from the most number of jobs completed to the least number of jobs completed using the interval partition method. Run the top k cores to complete the most jobs possible. Does this scheme work? If not, provide a counterexample.

Solution: The algorithm does not give the maximum number of tasks possible.

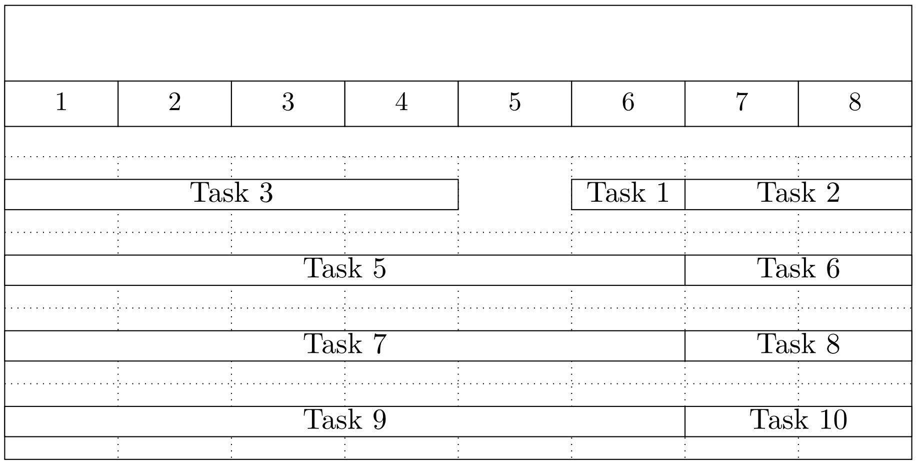
The following is a counterexample:



Selecting the top four cores after running the algorithm it can be seen that we get the diagram:



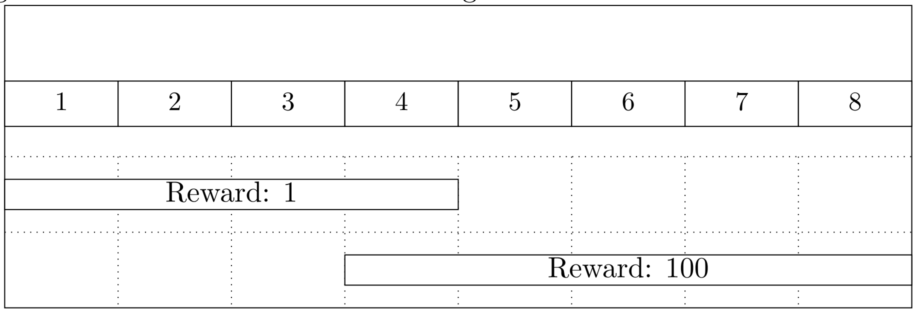
which completes 8 jobs. However, the following selection of jobs allows 9 jobs to be run on the four processors:



5 Question 4

Consider the scheduling problem but now there are rewards associated with each job that one can receive after the job has been completed. Instead of trying to complete the most jobs, our goal is to now maximize the reward. Consider the algorithm discussed in class where jobs are sorted by their finish time. Does following this algorithm always give the optimal solution for this modified problem? If not, provide a counterexample.

Solution: Each reward can be modeled as a number such that the more substantial the reward, the larger the number is. Then the following should work:



Running the algorithm will give you a reward of 1, however, the optimal solution is to finish the job that gives a reward of 100.

6 Question 5

Consider the problem of sending a file over a network from one computer to another. For this problem, we will represent a network as a graph where computers and routers are vertices and the bandwidth between two vertices is the edge weight associated with them (if such an edge exists). The bandwidth of a path is the minimum weighted edge in the path. Such an edge is a *bottleneck*. Can you adapt Dijkstra's algorithm to find a path between two computers such that the bandwidth of the path is maximized?

Solution:

We adapt Dijkstra's algorithm to the following:

Algorithm 1 Bottleneck Dijkstra

```
1: procedure BOTTLENECK-DIJKSTRA($(G, c)$)          ▷ $c$ assigns a bandwidth to each edge.
2:     $S = \{s\}$, $b : V \to \mathbb{R}$          ▷ $b$ keeps track of the bandwidth from $s$ to some vertex $v$
3:     $b(s) = \infty$
4:     while $S \neq V$ do
5:         Select $v \notin S$ such that
6:         $\pi(v) = \max_{e=(u,v),u\in S} \min\{b(u), c(e)\}$.
7:         $b(v) = \pi(v)$.
8:         Add $v$ to $S$.
```

To see this works, we proceed with nearly the same to the proof of the usual Dijkstra's algorithm as described in the book.

Theorem 6.1. *At any point in the algorithm, for every $u \in S$, the path P_u maximizes the bandwidth.*

Proving this establishes that the algorithm is correct, as $S = V$ at the end.

Proof. By convention let's assume that there is no connection from a node to itself so the bandwidth is ∞.

Induct on $|S|$. When $|S| = 1$, $S = \{s\}$ and $d(s) = \infty$. So the base case holds. Now assume the claim holds for $|S| = k$. Then grow S to size $k + 1$ by adding the node v. Let (u, v) be the final edge in the $s - v$ path P_v.

By the induction hypothesis P_u is the $s - u$ path with the most bandwidth for any $u \in S$. Consider any $s - v$ path P besides P_v; we wish to show it has at most the bandwidth of P_v. In order to reach v, this path P must leave the set S somewhere; let y be the first node on P that is not in S, and let $x \in S$ be the node just before y. Indeed, in the $k + 1$th iteration, Dijkstra's Bottleneck algorithm must have considered adding node y to S via the edge (x, y). After consideration, Dijkstra's Bottleneck algorithm rejected it in favor of P_v. And so $\min\{b(x), c((x, y))\} \leq \min\{b(u), c((u, v))\}$.

But observe that the bandwidth can only decrease or stay the same as new edges are explored, so let let z be the vertex before v on the path P. Then

$$\min\{b(z), c((z, v))\} \leq \min\{b(x), c((x, y))\} \leq \min\{b(u), c((u, v))\}$$

So there cannot be any path from s to y that goes through x and has more bandwidth than P_v. By the inductive hypothesis the claim holds. □

7 Question 6

Which of the following algorithms (if any) produce a minimum spanning tree? Justify your answer.

7.1 6a

Algorithm 2 MST Variant 1

```
1: procedure MST-VARIANT-1($G, w$)
2:     $E' = \text{sorted}(E(G), key = weight)$                ▷ By non-decreasing order.
3:     $T \leftarrow G$
4:     for $e \in E'$ do
5:         if $T - \{e\}$ is connected then
6:             $T \leftarrow T - \{e\}$
       return T
```

Solution: No. You should instead sort by non-increasing order. Consider Figure 2:

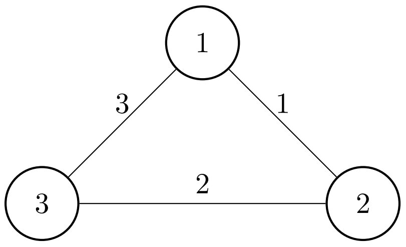


Figure 2: Weighted triangle.

The algorithm would remove {1} (since non-decreasing order sorts the vertices as $[1, 2, 3]$. But this results in Figure 3.

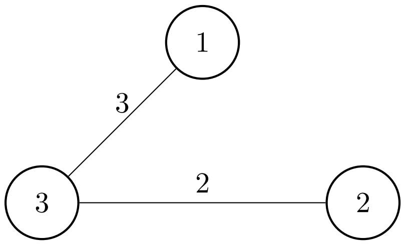
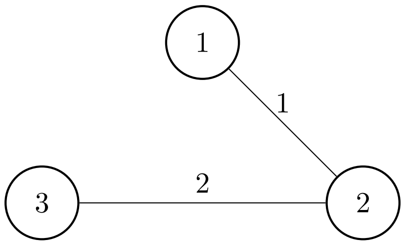


Figure 3: Unoptimal spanning tree.

when the optimal is Figure 4.

Figure 4: Optimal Spanning Tree



7.2 6b

Algorithm 3 MST Variant 2

```
1: procedure MST-VARIANT-2($G, w$)
2:     $T = \{\}$
3:     for $e \in E(G)$ do                                  ▷ Unsorted.
4:         if $T \cup \{e\}$ does not create a cycle. then
5:             $T \leftarrow T \cup \{e\}$
       return T
```

Solution: No. You should sort the edges by non-increasing order. Unsorted edges can result in the same problem as 6a.

7.3 6c

Algorithm 4 MST Variant 3

```
1: procedure MST-VARIANT-3($G, w$)
2:     $T = \{\}$
3:     for $e \in E(G)$ do                                  ▷ Unsorted.
4:         $T \leftarrow T \cup \{e\}$
5:         if $T$ has a cycle $c$. then
6:             Remove the minimum weight edge from $c$.
       return T
```

Solution: No. Consider Figure 2. All the edges will be added to T. Then T will remove the minimum weighted edge, but that will result in the unoptimal graph in Figure 3.