3 Scheduling All Intervals

We analyze the greedy algorithm for scheduling all intervals using the minimum possible number of processors. The algorithm sorts the intervals according to the start times (break ties arbitrarily). For each interval in the order, the algorithm first tries to use an existing processor to process it, if there are multiple processors that can do the job, it assigns the interval to an arbitrary one among them. If none of the existing processor can process the current interval, the algorithm assigns the interval to a newly added processor.

To prove the correctness of the algorithm, we introduce the concept of "depth at time t" $depth(t)$ - the number of intervals which contain t. The depth of the problem is the maximum depth at any time. We first argue that any valid solution will need at least $depth = \max_t depth(t)$ processors. This is because at any time, we need at least $depth(t)$ processors to process the $depth(t)$ active intervals at time t. We then show that our algorithm returns a solution that uses at most $depth$ many processors (see the claim below). If we combine the two statements above, we get the following inequality : $depth \leq$ optimal solution $\leq$ algorithm's solution $\leq depth$. Therefore $depth =$ optimal solution $=$ algorithm's solution.

Claim 5 *The algorithm uses at most $depth$ many processors.*

Proof: Let us prove the claim by contradiction. Assume for contradiction that the algorithm uses at least $(depth + 1)$ processors. We can further assume that when the algorithm allocates the $(depth + 1)$-st processor, it is for I_j. At that moment, there are already $depth$ many processors, namely $P_1, \ldots, P_{depth}$. For each processor ($1 \leq k \leq depth$), we have that P_k cannot process I_j, i.e., there exists an interval I_{q_k} which has been assigned to P_k, and overlaps with I_j. Since our algorithm assigns each interval to a processor in the non-decreasing order of the interval's start time, we have that $s_{q_k} \leq s_j$. Since I_{q_k} overlaps with I_j, we further have $f_{q_k} > s_j$. Now if we set $t = s_j + \delta$ for some infinitesimal $\delta > 0$, we see that I_{q_k} is active at time t for every $1 \leq k \leq depth$. Furthermore, since I_j is also active at time t, we see that there are at least $(depth + 1)$ active jobs at time t, i.e., $depth(t) \geq depth + 1$. Contradiction. □

For the runtime of the algorithm, the simple implementation uses $O(nd)$ time where d is the $depth$ of the problem. In the worst case, d can be as big as $\Theta(n)$ and therefore the algorithm could use $\Theta(n^2)$ time. In the homework, you are asked for a more efficient implementation.

Homework 3 *The algorithm can be implemented using $O(n \log n)$ time.*

4 Dijkstra's Shortest Path Algorithm [KT 4.4]

The algorithm: See the figure below. For a node $v \in V$, $d(v)$ is the shortest distance from s to v, and $d'(v)$ is the shortest distance from s to v passing nodes only in the explored subgraph. As we keep exploring the graph, we will eventually have $d'(v) = d(v)$.

```
Dijkstra's Algorithm (G, ℓ)
Let S be the set of explored nodes
    For each u ∈ S, we store a distance d(u)
Initially S = {s} and d(s) = 0
While S ≠ V
    Select a node v ∉ S with at least one edge from S for which
        d'(v) = min_{e=(u,v):u∈S} d(u) + ℓ_e is as small as possible
    Add v to S and define d(v) = d'(v)
EndWhile
```

A snapshot of the execution of the Dijkstra's algorithm.

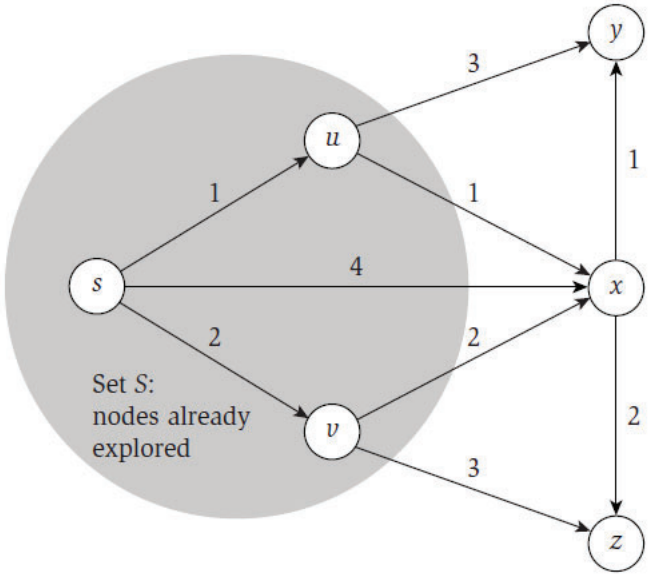

Figure 4.7 A snapshot of the execution of Dijkstra's Algorithm. The next node that will be added to the set S is x, due to the path through u.

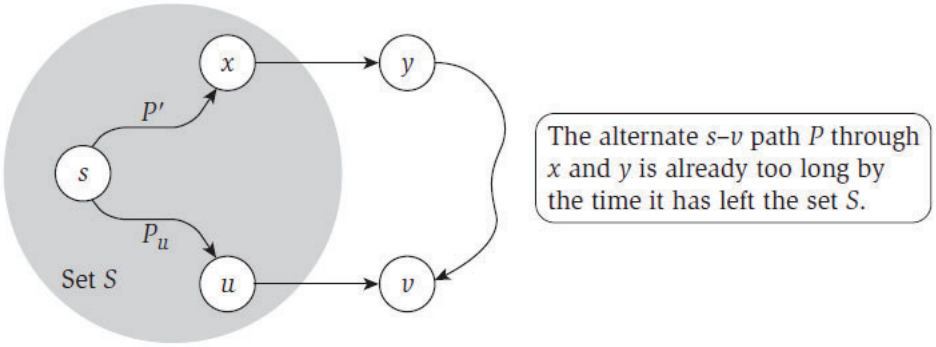

Figure 4.8 The shortest path P_v and an alternate s-v path P through the node y.

Proof of correctness:

Proof: We prove by induction, and use the "stay ahead" argument. We show

> At any point in the algorithm's execution, for each $u \in S$, the path P_u is a shortest $s \rightsquigarrow u$ path.

The base case is trivial: when $S = \{s\}$, we have $d(s) = 0$ which is the shortest path from s to s, given that there is no negative edges.

Suppose this is true when $|S| = k$. We now consider growing S by adding v. Let (u, v) be the edge that we add according to Dijkstra's algorithm, and let $P_v = s \rightsquigarrow u \to v$. We need to show that $d(v) = len(P_v)$ is also the shortest path distance from s to v.

Suppose P is another $s \rightsquigarrow v$ path, and P leaves S at node y. Then $len(P) > len(P_v)$ since $len(s \rightsquigarrow y) \geq len(P_v)$ (and there is no negative edges), since otherwise Dijkstra's algorithm will add y instead of v. □

A few comments:

1. Dijkstra can compute (s, v) for any $v \in V$, and of course for the particular (s, t) pair.
2. Does not work for negative edges. Which part of the proof breaks?

   Homework 4 *Give a counter example showing that Dijkstra's algorithm does not work for graphs with negative edges. Please example why Dijkstra's algorithm fails in this case.*

   Later we will discuss the Bellman-Ford algo which works for negative edges.
3. Dijkstra is essentially BFS on weighted edges: Imagine what will happen if we replace each edge e with a path of $w(e)$ unit edges. Motivated by "waterfall".

Implementation: use priority queue PQ. Insert each v to the PQ with priority $d'(v)$. Call `ExtractMin` when need a new v.

For a new node v added into S, for each neighbour w connected by edge $e = (v, w)$, we update $d'(w) = \min\{d'(w), d(v) + len(e)\}$, using `DecreaseKey` ([KT] call it `ChangeKey`)

Running time: the above implementation needs $O(n)$ `ExtractMin` and $O(m)$ `DecreaseKey`. This turns out to be $O(m \log n)$ time if we use a binary heap, and $O(m + n \log n)$ time if we use the Fibonacci heap.

Rebuild the shortest path: Along the algorithm, we can construct an additional array $from()$ so that $from(u)$ points to the last node before reaching u in the shortest path from s to u. We update the $from()$ array whenever there is an update to the $d'()$ array (i.e., when update $d'(w) = \min\{d'(w), d(v) + len(e)\}$).