A Fast Algorithm for Multi-Machine Scheduling Problems with Jobs of Equal Processing Times

Alejandro López-Ortiz[1] and Claude-Guy Quimper[2]

1 **University of Waterloo**
Waterloo, Canada
alopez-o@uwaterloo.ca
2 **Université Laval**
Québec, Canada
claude-guy.quimper@ift.ulaval.ca

Abstract

Consider the problem of scheduling a set of tasks of length p without preemption on m identical machines with given release and deadline times. We present a new algorithm for computing the schedule with minimal completion times and makespan. The algorithm has time complexity $O(\min(1, \frac{p}{m})n^2)$ which improves substantially over the best known algorithm with complexity $O(mn^2)$.

1998 ACM Subject Classification Algorithms and data structures

Keywords and phrases Scheduling

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

We consider the problem of scheduling a set of equal length tasks without preemption on m identical machines with given release and deadline times. The goal is to produce a schedule, if one exists, that minimizes the sum of the completion times. We later prove that this simultaneously minimizes the makespan. This scheduling problem is known as $Pm|r_j; p_j = p; D_j|\sum C_j$ in the notation used by Pinedo [8].

The scheduling problem we study is formally defined as follows. There are n jobs labeled from 1 to n with integer release times r_i and latest starting times u_i such that $r_i < u_i$ for $i \in 1..n$. A job can start on or after time r_i but must start strickly before time u_i. Each job has an integer processing time p and needs to be allocated on one of the m identical machines. Jobs are not allowed to be preempted and only one job at a time can be executed on a machine. The deadline of job i is therefore given by $u_i + p - 1$. We therefore need to find for each job i a starting time s_i such that $r_i \leq s_i < u_i$ and that for any time t, no more than m jobs are being executed. Moreover, we would like to minimize the total completion time, i.e. the sum of the completion time of each job. Formally, we have the following system to solve.

$$\min \sum_{i=1}^{n} s_i \tag{1}$$

$$r_i \leq s_i < u_i \qquad \forall i \in 1..n \tag{2}$$

$$|\{i \mid t \leq s_i < t + p\}| \leq m \qquad \forall t \tag{3}$$

Simons [9] proposed the first polynomial time algorithm running in $O(n^3 \log\log n)$ for solving this problem. Simons and Warmuth [10] further improved this bound to $O(mn^2)$. Vakhania [11] presented an algorithm that runs in $O(d_{\max}^2(m + d_{\max})n \log n)$ where $d_{\max}$ is the latest deadline. Note

Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that in any feasible instance we have $d_{\max} \geq p \left\lceil \frac{n}{m} \right\rceil$. Vakhania's algorithm is therefore competitive when the processing time p is small and the number of machines is proportional to the number of jobs. Dürr and Hurand [3] gave an algorithm that runs in $O(n^3)$. Even though their algorithm does not improve over the best time complexity, it deepens the understanding of the problem by reducing it to a shortest path in a graph. The algorithm presented in this paper is based on this reduction while improving substantially its time complexity.

There exist specializations to the problem with more efficient algorithms. For instance, on a single machine ($m = 1$) and with a unit processing time ($p = 1$), the problem consists of a matching in a convex bipartite graph. Lipski and Preparata [7] designed an algorithm running in $O(n\alpha(n))$ time where α is the inverse of Ackermann's function and where it is assumed that the jobs are presorted by deadlines. Gabow and Tarjan's [4] showed how to reduce this complexity to $O(n)$ using their union-find data structure. The algorithm can be easily adapted without altering the complexity for multiple machines even in the case where the number of machines fluctuates over time. Finally, Garey et al. [5] solve the scheduling problem in $O(n \log n)$ time for jobs of equal processing times on a single machine ($m = 1$).

2 Reduction to a Shortest Path

Suppose that an oracle provides the number x_t of jobs starting at time t. A solution can be constructed using a matching in a convex bipartite graph. For each job j, we create a node j and for each time t, we create x_t duplicates of a node t. There is an edge between a job node j and a time node t if $r_j \leq t < u_j$. A matching in such a graph associates to each job a starting time. Since the graph is convex, Lipski and Preparata [7] show how to find such a matching in $O(n\alpha(n))$ time.

Thus we have reduced the scheduling problem to finding how many jobs start at any given time t. We answer this question by computing a shortest path in a graph in a manner similar to what Dürr and Hurand [3] did. Their solution consists of building a graph with $O(n)$ nodes and $O(n^2)$ edges and to compute the shortest path using the Bellman-Ford algorithm in $O(n^3)$ time. We propose a similar approach with a graph having more nodes. These additional nodes make the computation of a solution easier. However, the main contribution of our technique is presented in Section 3 to 5 where we identify and exploit the properties of the graph to obtain a substantially faster algorithm.

We know that at most m jobs can start in any window of size p. We can already state the equations. Let $r_{\min} = \min_i r_i$ and $u_{\max} = \max_i u_i$ be the earliest release time and latest allowed starting time.

$$\sum_{j=t}^{t+p-1} x_j \leq m \qquad \forall\, r_{\min} \leq t \leq u_{\max} - p \tag{4}$$

$$x_t \geq 0 \qquad \forall\, r_{\min} \leq t < u_{\max} \tag{5}$$

Condition 4 states that at most m processes may start on a given interval of length p. Let $u_{\max} = \max_i u_i$ be the latest time when a job can start. Given two (possibly identical) jobs i and j defining a non-empty semi-open interval $[r_i, u_j)$, the set $\{k \mid r_i \leq r_k \wedge u_k \leq u_j\}$ denotes the jobs that must start in this interval, hence

$$\sum_{t=r_i}^{u_j-1} x_t \geq |\{k \mid r_i \leq r_k \wedge u_k \leq u_j\}| \qquad \forall\, i, j \in 1..n \tag{6}$$

▶ **Lemma 1.** *The scheduling problem has a solution if and only if Equations* (4) *to* (6) *are satisfiable.*

Proof. ($\Rightarrow$) Given a valid schedule, we set x_t to the number of jobs starting at time t, i.e $x_t = \{i \mid s_i = t\}$. By definition of the problem, all equations are satisfied.
($\Leftarrow$) Consider a bipartite graph $G = \langle J \cup T, E\rangle$ such that $J = \{1, \ldots, n\}$ are the nodes associated to the jobs and T is a multiset of time points such that time t occurs x_t times in T. There is an edge from the job-node i to the time-node t if $t \in [r_i, u_i]$. Note that the bipartite graph is convex, i.e. if there is an edge (i, t_1) and an edge (i, t_3) then there is an edge (i, t_2) for all $t_2 \in [t_1, t_3] \cap T$. A maximum matching in this convex bipartite graph gives a valid assignment of jobs to time points. Equation (4) ensures that no machines are overloaded and the schedule is feasible. From Hall's [6] marriage theorem, there exists a matching if and only if for any set of jobs S, there are at least $|S|$ time nodes that are adjacent to the nodes in S. Let $i \in S$ be the job with the earliest release time and $j \in S$ be the job with the latest deadline. Inequality (6) ensures that there are at least $|\{k \mid r_i \leq r_k \wedge u_k \leq u_j\}| \geq |S|$ time-nodes adjacent to the nodes in S which meets Hall's condidtion. ◀

We perform a change of variables to simplify the form of the equations. Let $y_t = \sum_{i=r_{\min}}^{t-1} x_i$ for $r_{\min} \leq t \leq u_{\max}$. Equations (4) to (6) are rewritten as follows.

$$y_{t+p} - y_t \leq m \qquad \forall\, r_{\min} \leq t \leq u_{\max} - p \tag{7}$$

$$y_t - y_{t+1} \leq 0 \qquad \forall\, r_{\min} \leq t < u_{\max} \tag{8}$$

$$y_{r_i} - y_{u_j} \leq -|\{k \mid r_i \leq r_k \wedge u_k \leq u_j\}| \qquad \forall\, r_i < u_j \tag{9}$$

Equations (7) to (9) form a system of difference constraints, which can be solved creating a graph with one node per variable and an edge (a, b) of weight w for each constraint $b - a \leq w$. For the equations above, we obtain a graph $G = \langle T, E\rangle$ where $T = r_{\min}..u_{\max}$ is the set of nodes, one for each integer time point. We consider three sets of edges: *forward edges* $E_f = \{(t, t+p) \mid r_{\min} \leq t \leq u_{\max} - p\}$, *backward edges* $E_b = \{(u_j, r_i) \mid r_i < u_j\}$, and *null edges* $E_n = \{(t+1, t) \mid r_{\min} \leq t < u_{\max}\}$. The edges of the graph are the union of these three sets of edges $E = E_f \cup E_n \cup E_b$ that are directly derived from Equations (7), (8), and (9). The following weight function maps every edge to an integer weight. Let $(a, b) \in E$, then

$$w(a,b) = \begin{cases} m & \text{if } a < b \\ -|\{k \mid b \leq r_k \wedge u_k \leq a\}| & \text{otherwise} \end{cases} \tag{10}$$

We call the graph G the *scheduling graph*. The following theorem shows the connexion between a shortest path in the scheduling graph and a solution to the system of difference constraints. The proof is taken from Cormen et al. [2] who credit it to R. Bellman.

▶ **Theorem 2.** *Let $\delta(a, b)$ be the shortest distance between node a and node b in the scheduling graph. The assignment $y_t = n + \delta(u_{\max}, t)$ is a solution to Equations* (7) *to* (9)*.*

Proof. Suppose there is an inequality $y_b - y_a \leq w(a, b)$ that is not satisfied by the assignment, we therefore have $n + \delta(u_{\max}, b) - n - \delta(u_{\max}, a) > w(a, b)$. The inequality $\delta(u_{\max}, b) > \delta(u_{\max}, a) + w(a, b)$ contradicts that $\delta(u_{\max}, b)$ is the shortest distance from $u_{\max}$ to b. ◀

Let $|T| \in O(u_{\max} - r_{\min})$ be the number of nodes and $|E| \in O(n^2 + u_{\max} - r_{\min})$ the number of edges in the scheduling graph. Here we could directly apply a shortest path algorithm such as Bellman-Ford to compute a shortest path from $u_{\max}$ to all other nodes in the graph. These algorithms run in time polynomial in T and $|E|$, or in other words, pseudo-polynomial on the term $u_{\max}$. In the next section we show properties of the scheduling graph and use them to propose a much more efficient method based on a speed up version of Bellman- Ford's algorithm.

Figure 1 presents a scheduling graph with 2 machines, 5 jobs, and a processing time of 3.

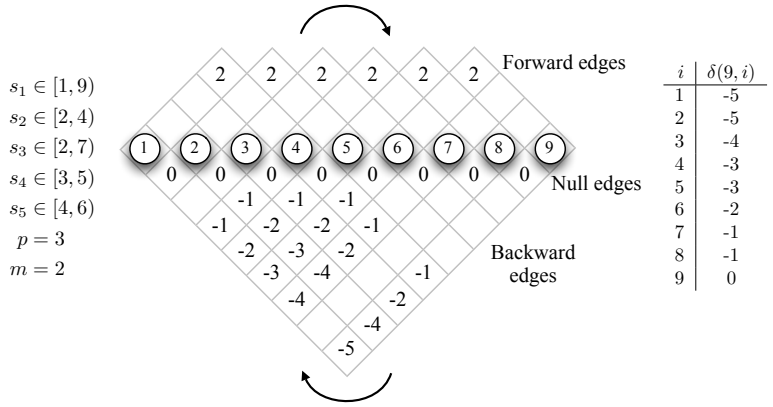

Figure 1 A scheduling graph with 9 nodes. The weight on an edge between two nodes appear at the intersection of the two diagonals passing by these nodes. The weights of the null and backward edges appear bellow the nodes while the weights of the forward edges appear above the nodes. Empty cells indicate the absence of an edge. For instance, the weight of backward edge $(9, 2)$ is -4. The shortest path between node 9 and 6 passes by the nodes 9, 1, 4, 7, 2, 5, 3, 6.

3 Properties of the Scheduling Graph

▶ **Lemma 3.** *Let $e < f < g < h$ be four nodes in a scheduling graph without negative cycles. If the edges (h, f) and (g, e) lie on a shortest path then there exists an equivalent path of same length that does not include these edges.*

Proof. Suppose that the edges (h, f) and (g, e) lie on a same shortest path and that (h, f) precedes (g, e) on this path. Since any sub-path of a shortest path is also a shortest path, we have

$$w(h, e) \geq w(h, f) + \delta(f, g) + w(g, e) \tag{11}$$

Recalling that $-w(y, x)$ for $x < y$ is the number of jobs that must start in the time interval $[x, y)$, we know that the following relationship holds on backward edges.

$$w(h, e) \leq w(g, e) + w(h, f) - w(g, f) \tag{12}$$

Subtracting (12) from (11) shows that the cycle passing by (f, g) is negative or null and since there are no negative cycles, we obtain the equality $0 = w(g, f) + \delta(f, g)$ Substituting this equality back in (12) shows an equality in (11). The edge (h, e) is therefore an equivalent path that does not contain the edges (h, f) nor (g, e).

Alternatively, suppose that (h, f) succeeds to (g, e) on the path. We have

$$w(g, f) \geq w(g, e) + \delta(e, h) + w(h, f) \tag{13}$$

Adding (12) to (13) gives $0 \geq w(h, e) + \delta(e, h)$. Since there are no negative cycles in the scheduling graph, the inequality is tight. Substituting the equality into (12) shows that (13) is tight and that (g, f) is an equivalent shortest path. ◀

A backward edge (b, a) is associated to the interval $[a, b)$. If two backward edges have disjoint intervals, we say that the backward edges are *disjoint*. If the interval of one backward edge is contained in the interval of another backward edge, we say that the backward edges are *nested*. Otherwise, we say that the backward edges are *crossed*.

▶ **Lemma 4.** *The shortest path which also minimizes the number of edges does not have crossed backward edges.*

Proof. By applying Lemma 3 on a shortest path, one obtains a shortest path with two crossed edges and one forward edge replaced by one backward edge. One can repeat the process until there are no more crossed edges in the path. Since each time we apply Lemma 3, the number of edges in the path diminishes, the process is guaranteed to finish. ◀

Let d be the distance vector such that $d[t] = \delta(u_{\max}, t)$ is the shortest distance from node $u_{\max}$ to node t. The vector d is monotonically increasing.

▶ **Lemma 5.** *The distance vector d is monotonically increasing.*

Proof. Consider the nodes t and $t+1$, the null edges E_n guarantees that $d[t] \leq d[t+1]+w(t+1,t)$ or simply $d[t] \leq d[t+1]$. ◀

▶ **Lemma 6.** *If the scheduling graph has no negative cycles, $d[r_{\min}] = -n$.*

Proof. Lemma 4 implies that there is a shortest path from $u_{\max}$ to $r_{\min}$ that do not have forward edges. Because two consecutive backward edges are no shorter than one longer backward edge ($w(c,a) \leq w(c,b) + w(b,a)$ for any time point $a < b < c$) we conclude that the edge $(u_{\max}, r_{\min})$ is a single-segment shortest path from $u_{\max}$ to $r_{\min}$ with distance $-n$. ◀

Lemma 5 and Lemma 6 implies that the distance vector d contains the values $-n..0$ in non-decreasing order. Keeping a structure in memory of the n time points where the vector d is strictly increasing is sufficient to retrieve all components of vector d. This is a first step towards a strongly polynomial algorithm.

4 The Algorithm

We present an algorithm based on the Bellman-Ford algorithm [1] for the single-source shortest path problem. We encode the distance vector with a vector d^{-1} of dimension $n+1$. The component $d^{-1}[i]$ is the rightmost node reachable at distance at most $-i$. For example, if $n = 10$ and $d^{-1}[3] = 4$, there is a path from $u_{\max}$ to 4 of weight at most -3.

An iteration of the Bellman-Ford algorithm applies the relaxation $d[b] \leftarrow \min_{(a,b)\in E} d[a] + w(a,b)$ for all nodes b and assumes that there is an edge (b,b) of null weight on all nodes. After sufficiently many iterations. the algorithm converges to a distance vector d such that $d[a]$ is the shortest distance between the source node and the node a.

We develop two procedures. One that applies the relaxation to the edges in $E_n \cup E_f$ and one that applies it to the edges in $E_n \cup E_b$. Yen [12] introduced the technique of partitioning edges between forward and backward edges to reduce the number of iterations of the Bellman-Ford algorithm to the number of times a shortest path alternates between a backward edge to a forward edge. In a scheduling graph, the number of alternations can be reduced to $\min(n, \lceil \frac{n}{m} \rceil p)$ as we will prove in Section 5. The algorithm for finding the starting times adapts the Bellman-Ford algorithm to the scheduling graph. If the distance vector of the algorithm does not converge after a sufficient number of iterations there exists a negative cycle in the graph proving that the problem is unsolvable. The

algorithm then returns an error message.

Algorithm 1: FindStartingTimes($\vec{r}, \vec{u}, m, p$)

$B \leftarrow sort(\{r_i \mid i \in 1..n\} \cup \{u_i \mid i \in 1..n\})$;
for $i = 1$ *to* n **do** $l_i \leftarrow index(B, r_i)$;
for $i = 1$ *to* n **do** $v_i \leftarrow index(B, u_i)$;
$d_0 \leftarrow [u_{\max}, \underbrace{r_{\min}, \ldots, r_{\min}}_{n \text{ copies}}]$;
for $k \leftarrow 1$ *to* $\min(n, \lceil \frac{n}{m} \rceil p) + 1$ **do**
  $\vec{d_k} \leftarrow RelaxForwardEdges(\vec{d_{k-1}}, m, p)$;
  $\vec{d_k} \leftarrow RelaxBackwardEdges(\vec{d_k}, \vec{l}, \vec{v}, B)$;
  if $\vec{d_k} = \vec{d_{k-1}}$ **then return** $[d_k[n-1], d_k[n-2], \ldots, d_k[0]]$;
return Failure;

Relaxing the Forward Edges.

Relaxing forward edges is done in $O(n)$ time by iterating over the distance vector d^{-1}. It ensures that if there is a path of distance i that goes to node x, then there is a path of distance $i + m$ that reaches node $x + p$. For all possible distances i spanning from $-n$ to $-m$, we apply the relaxation $d^{-1}[-i - m] \leftarrow \max(d^{-1}[-i] + p, d^{-1}[-i - m])$.

Algorithm 2: RelaxForwardEdges($\vec{d}, m, p$)

for $i \leftarrow -n$ *to* $-m$ **do**
  $d[-i - m] \leftarrow \max(d[-i] + p, d[-i - m])$;
return $\vec{d}$;

Relaxing the Backward Edges.

Processing backward edges in $O(n)$ time is more complex. Assume that jobs are sorted in non-decreasing order of release times ($r_i \leq r_{i+1}$). The algorithm is based on the similarity between the backward edges incoming to node r_i and backward edges incoming to node r_{i+1}.

▶ **Lemma 7.** *Let J_i be the set of jobs sharing the same release time as r_i. The backward edges incoming to r_i and r_{i+1} are linked by the relation $w(t, r_i) = w(t, r_{i+1}) - |\{k \in J_i \mid u_k \leq t\}|$.*

Proof. Recall that $w(a, b)$ is the negation of the number of jobs that must start in the interval (b, a). The number of jobs that must start in the time interval $[r_i, t)$ is the number of jobs that *must* start in the interval $[r_{i+1}, t)$ plus the number of jobs that *can* start in $[r_i, r_{i+1})$ but must start before t. Hence $-w(t, r_i) = -w(t, r_{i+1}) + |\{k \in J_i | u_k \leq t\}|$ as claimed. ◀

Let $d[u_j]$ be the best distance found so far by the Bellman-Ford algorithm from node $u_{\max}$ to node u_j. Relaxing the backward edges consists of computing the value $\min_j d[u_j] + w(u_j, r_i)$ for all r_i. The following Lemma shows that not all edges need to be processed.

▶ **Lemma 8.** *Given two latest starting times $u_a < u_b$, if $d[u_a] + w(u_a, r_{i+1}) \geq d[u_b] + w(u_b, r_{i+1})$ then $d[u_a] + w(u_a, r_i) \geq d[u_b] + w(u_b, r_i)$.*

Proof. Using the set J_i as defined in Lemma 7, we have $|\{j \in J_i \mid u_j \leq u_a\}| \leq |\{j \in J_i \mid u_j \leq u_b\}|$. From Lemma 7, we obtain $w(u_a, r_{i+1}) - w(u_a, r_i) \leq w(u_b, r_{i+1}) - w(u_b, r_i)$. Subtracting this inequality from the hypothesis proves the Lemma. ◀

Lemma 8 is fundamental to obtain a fast algorithm relaxing the backward edges. It says that when processing the backward edges ingoing to r_{i+1}, if the edge (u_b, r_{i+1}) is a better or equivalent candidate for a shortest path than (u_a, r_{i+1}) then the edge (u_b, r_i) is also a better or equivalent candidate than (u_a, r_i). By transitivity, any backward edge outgoing from u_a and ingoing to a node smaller than r_{i+1} can be ignored.

Let B be the set containing all the release times $r_1, \ldots, r_n$ and latest starting times $u_1, \ldots, u_n$. This set contains no duplicates and its elements are labeled from b_1 to $b_{|B|}$.

We construct a singly linked-list that we call the *list of representatives*. The list initially contains the elements of B in increasing order. Each element of the list is a *representative* of a set that initially only contains itself. The representative is always the largest element of its set. Each set is represented in the data structure by a node labeled by its representative that has a link to the previous node $b_1 \leftarrow b_2 \leftarrow \cdots$. The link between b_{j+1} and b_j has weight $d[b_{j+1}] - d[b_j]$. If the weight of a link (b_{j+1}, b_j) is null, we merge the node b_j and the node b_{j+1} together to form a larger set for which b_{j+1} is the representative. The node b_j disappears from the list of representatives since Lemma 8 ensures that b_{j+1} will always be a better candidate than b_j.

On the running example of Figure 1, the vector $\vec{d}$ is initialized to $\vec{d} = [9, 1, 1, 1, 1, 1]$. After the forward edge relaxation stage, its value becomes $\vec{d} = [9, 7, 4, 4, 1, 1]$. The representatives are $B = \{1, 2, 3, 4, 5, 6, 7, 9\}$ which gives the vector $d = [-5, -3, -3, -3, -1, -1, -1, 0]$ that maps each element in B to a distance. After merging the sets connected with a null link, we obtain the following chain where representatives are highlighted in bold.

$$\{\mathbf{1}\} \xleftarrow{2} \{2, 3, \mathbf{4}\} \xleftarrow{2} \{5, 6, \mathbf{7}\} \xleftarrow{1} \{\mathbf{9}\} \tag{14}$$

Initially, the data structure allows us to compute $d[b_j]$ for any j. As we shall show in Lemma 9, one only needs to visit the nodes from b_j to b_1 and sum up the weights on the links to obtain $d[b_j]+n$. Subtracting n from the sum of the links gives $d[b_j]$. The data structure can be updated to compute the values $d[b_j] + w(b_j, r_i)$ for each backward edge (b_j, r_i) in the scheduling graph. We process the tasks in non-increasing order of release time starting with r_n. When processing task i, we first look in the data structure for the representative of u_i which we call b_q. Assume that the node b_q points to b_t, we decrement the weight of the link (b_q, b_t). If the weight of the link becomes null after decrementing, we delete b_t from the list of representatives and merge the set containing b_t with the set represented by b_q. The element b_q remains the representative of the merged set.

Continuing with the running example, processing job 5 decreases by one the weight on the link between node 7 and node 4. Processing job 4 decreases once more the weight of this link and fix it to zero. The data structure then looks as follows.

$$\{\mathbf{1}\} \xleftarrow{2} \{2, 3, 4, 5, 6, \mathbf{7}\} \xleftarrow{1} \{\mathbf{9}\} \tag{15}$$

▶ **Lemma 9.** *After processing the last job with release time r_i, the sum of the weights on the links from the representative of u_j to node b_1 is equal to $d[u_j] + w(u_j, r_i) + n$.*

Proof. Let b_k be the representative of u_j. Initially, the weights on the links of the data structure from b_k to b_1 are equal to the telescopic sum $\sum_{l=0}^{k-1} (d[b_{l+1}] - d[b_l])$ which simplifies to $d[b_j] - d[b_1] = d[b_j] + n$. After processing the last job with release time r_i, all jobs that must start at or after r_i and before b_k have been processed. Each of these $|\{a \mid r_i \leq r_a \wedge u_a \leq u_j\}|$ jobs decremented by one a link on the path between b_k and b_1. Therefore, the sum of the links on a path between u_j and b_1 is $d[u_j] + w(u_j, r_i) + n$. ◀

The Bellman-Ford algorithm requires to find the backward edge incoming into r_i that minimizes the value $d'[b_j] + w(b_j, r_i)$ where b_j can be r_i. Lemma 10 shows how the data structure finds the optimal edge.

▶ **Lemma 10.** *Let b_j be the representative of r_i after processing all jobs with release times greater than or equal to r_i. The backward edge (b_j, r_i) is the one minimizing the value $d'[b_j] + w(b_j, r_i)$.*

Proof. The backward edge (b_j, r_i) that minimizes $d'[b_j]+w(b_j, r_i)$ also minimizes $d'[b_j]+w(b_j, r_i)+n$. Lemma 9 guarantees that the later value is equal to the weights on the path from b_j to b_1. Since all weights on the path are positive, the smallest representative that is greater than or equal to r_i is the one minimizing $d'[b_j] + w(b_j, r_i)$. This representative is necessarily the representative of r_i.

We prove that values that are not representatives are not optimal. If u_b is the representative of u_a, then for some release time $r_c \geq r_i$, the link from node r_b to r_a was decremented to zero. From Lemma 9, we have $d'[u_a] + w(u_a, r_c) = d'[u_b] + w(u_b, r_c)$. From Lemma 8 we conclude that $d'[u_a] + w(u_a, r_i) \geq d'[u_b] + w(u_b, r_i)$. Therefore, the representative u_b is as good or better than the non-representative u_a. ◀

The algorithm RelaxBackwardEdges uses the data structure discussed above to relax the backward edges. The first for loop on line 1 converts the vector d to the vector d'. Recall that $d[i]$ is the largest node in the graph reachable at distance at most $-i$ and $d'[i]$ is the smallest distance found so far to reach node $B[i]$ where B is the sorted vector of release times r_i and latest starting times u_i.

The algorithm then initializes the data structure. Each node is a set in a union-find data structure T whose representative is the largest element. The weight of the link pointing to a representative b_i is stored in $c[i]$. We store in $k[i]$ the number of jobs j that have been processed so far and for which the latest starting time u_j is represented by b_i. The for loop on line 2 processes each job in non-increasing order of release time. The data structure is updated as explained above. Line 3 computes the value $d'[b_e] + w(b_e, r_i)$ where b_e is the representative of r_i. Note that $k[e]$ is the number of processed jobs with latest starting time smaller than or equal to b_e which is equal to $-w(b_e, r_i)$.

5 Analysis

The following lemmas show the correctness of the algorithm and give the conditions to bound its time complexity.

▶ **Lemma 11.** *There is a shortest path from $u_{\max}$ to all other nodes with at most $\lceil \frac{n}{m} \rceil$ disjoint backward edges.*

Proof. Suppose a shortest path has k disjoint edges (b_i, a_i) for $1 \leq i \leq k$. We assume that these backward edges are interleaved with forward edges since two backward edges connected with null edges can be replaced by a single backward edge of cost equal or smaller than the sum of the weights of the backward edges. Since the intervals $[a_i, b_i)$ are disjoint, we have $\sum_{i=1}^{k} w(b_i, a_i) \geq -n$. It is safe to assume that the path has negative weight since a path of null weight can be entirely constituted of null edges without any backward edges. The path has $k-1$ sequences of forward edges whose weights sum to at most $n-1$. Each sequence of forward edges must be at least of weight m. To maximize the number k, we assume that each sequence of forward edges has a single edge. We have $(k-1)m \leq n-1$ which implies $k \leq \lfloor \frac{n-1}{m} \rfloor + 1 = \lceil \frac{n}{m} \rceil$. ◀

Lemma 12 gives an upper bound on the number of nested backward edges lying on a shortest path. Lemma 13 gives an upper bound on the number of backward edges lying on a shortest path.

▶ **Lemma 12.** *A shortest path can have at most p nested backward edges (b_i, a_i) such that $a_i < a_{i+1}$ and $b_i > b_{i+1}$.*

Proof. In such a path, there must be a sequence of forward edges that connects a_i to b_j for some i and some j. This sequence of forward edges cannot pass by a node a_k for $k > i$ since each node is

Algorithm 3: RelaxBackwardEdges($\vec{d}, \vec{l}, \vec{v}, B, W$)

```
Construct a vector d' s.t. the distance between u_max and B[i] is d'[i];
d' ← [ ];                                                    // Empty vector
j ← −n;
1 for b ∈ B in increasing order do
    while b > d[−j] do j ← j + 1;
    append(d', j);
T ← UnionFind(|B|) ;                                  // |B| disjoint sets
k ← [0, ..., 0];              // Create a null vector of dimension |B|
for i ← 1 to |B| − 1 do
    c[i] ← d'[i + 1] − d'[i];
    if c[i] = 0 then Union(T, i, i + 1);
2 for i ∈ [1, n] in non-increasing value of l_i do
    q ← FindMax(T, v_i);
    t ← FindMax(T, FindMin(T, v_i) − 1);
    c[t] ← c[t] − 1;
    k[q] ← k[q] + 1;
    if c[t] = 0 then
        Union(T, t, q);
        k[q] ← k[q] + k[t];
    e ← FindMax(T, l_i);
3   a ← d'[e] − k[e];
    if a < d'[l_i] then
        d'[l_i] ← a;
        d[−a] ← B[l_i];
return d⃗;
```

visited only once on a path. This implies that a_i and a_k are not congruent modulo p for every $k > i$. Consequently, $a_i \neq a_j \mod p$ for all $i \neq j$. The maximum set of values satisfying this property has cardinality p. ◀

▶ **Lemma 13.** *There is a shortest path with at most* $\min(n, \lceil \frac{n}{m} \rceil p)$ *backward edges.*

Proof. The number of backward edges on a path is bounded by n since there are at most n nodes r_i to which they lead to. Lemma 11 guarantees that there are at most $\lceil \frac{n}{m} \rceil$ disjoint backward edges. Each disjoint backward edge can have at most p nested backward edges. Therefore, there are at most $\lceil \frac{n}{m} \rceil p$ backward edges on a shortest path. The number of backward edges is bounded by the smallest of both bounds. ◀

▶ **Theorem 14.** *The algorithm for finding the starting times is correct.*

Proof. The correctness of the forward and backward edge relaxation algorithms follows from the discussions and Lemmas in the previous section. The correctness of algorithm for finding the starting times follows from the Bellman-Ford algorithm. We however need to justify why $\min(n, \lceil \frac{n}{m} \rceil p)$ iterations are sufficient. A path is necessarily an alternation of forward edges in E_f and backward or null edges in $E_b \cup E_n$. Yen [12] shows that the number of iterations can be bounded to the number of alternations. An alternation between a forward edge, a sequence of null edges, and another forward edge can be replaced by an equivalent path of two forward edges and a sequence of null edges (or

a sequence of null edges followed by two forward edges). Consequently, we can assume that the sequences of null edges occur before or after a backward edge. Lemma 13 gives an upper bound of $\min(n, \lceil \frac{n}{m} \rceil p)$ on the number of backward edges which is also an upper bound on the number of alternations. ◀

▶ **Theorem 15.** *The algorithm for finding the starting times completes in $O(\min(1, \frac{p}{m})n^2)$ steps.*

Proof. The running time complexity of the forward edge relaxation stage is clearly $O(n)$. The complexity of the backward edge relaxation stage depends on the implementation of the Union-Find data structure. There are $O(n)$ calls to the functions $FindMin$, $FindMax$, and $Union$. Using path compression, each call can be executed in $O(\alpha(n))$ time where α is the inverse of Ackermann's function. However, since the disjoint sets always contain consecutive values in B, Gabow and Tarjan [4] propose a data structure where each call executes in constant amortized time which makes the backward edge relaxation stage run in $O(n)$ steps. Finally, the algorithm for finding the starting times performs $min(n, \lceil \frac{n}{m} \rceil p)$ calls to the forward and backward edge relaxation stages which results in a running time complexity of $O(\min(1, \frac{p}{m})n^2)$. ◀

Observe that the running time complexity $O(\min(1, \frac{p}{m})n^2)$ is strongly polynomial in all parameters. While the presence of the variable p might suggest pseudo-polynomiality, the term $\min(1, \frac{p}{m})$ is always bounded by a constant. In the particular case where p is considered to be a small bounded value, the resulting complexity $O(\frac{n^2}{m})$ decreases as the number of machines m increases.

The algorithm has better performance in some special cases of interest. For instance, when $p = 1$, a shortest path cannot have nested backward edges as stated in Lemma 12. Neither can the path have disjoint backward edges. To wit, suppose that (a, b) and (c, d) are two disjoint backward edges such that (a, b) occurs before (c, d) on a shortest path. If $b \geq c$ then the edge (d, a) is an equivalent or shorter path. If $b < c$ then the forward edges need to pass by node c before reaching d creating a loop. With shortest paths including only one backward edge, the algorithm converges after one iteration.

We showed that the algorithm computes in polynomial time a shortest path and thus a valid schedule. We show that it also detects infeasibility in polynomial time. By Lemma 13, if there is no negative cycle, a shortest path has at most $\min(n, \lceil \frac{n}{m} \rceil p)$ alternations between backward and forward edges. By Yen's theorem, if there are no negative cycles, the Bellman-Ford algorithm will converge after $\min(n, \lceil \frac{n}{m} \rceil p)$ iterations. On the other hand, if the graph has a negative cycle, the Bellman-Ford never converges since the cost function tends to minus infinity. To detect infeasibility, the algorithm first iterates $\min(n, \lceil \frac{n}{m} \rceil p)$ times. If the graph has no negative cycles, the algorithm should have converged. To test if it did converge, the algorithm iterates one more time. If the distance vector has changed, then the algorithm has not converged and will never do. The algorithm detected infeasibility in exactly $\min(n, \lceil \frac{n}{m} \rceil p) + 1$ iterations which is strongly polynomial.

We now characterize the solution returned by the algorithm and prove that it minimizes both the sum of the completion times and the makespan.

▶ **Theorem 16.** *The solution returned by the algorithm for finding the starting times minimizes the sum of the completion times.*

Proof. Minimizing the sum of the completion times is equivalent to minimizing the sum of the starting times. We recall that x_t is the number of jobs starting at time t. Minimizing the sum of the starting times is equivalent to minimizing $\sum_{t=r_{\min}}^{u_{\max}-1} t x_t$. Performing the change of variables from x_t

to $y_{t+1} - y_t$ leads to a telescopic sum that we solve in (16) and simplify in (17).

$$\sum_{t=r_{\min}}^{u_{\max}-1} t(y_{t+1} - y_t) = (u_{\max} - 1)y_{u_{\max}} - r_{\min}y_{r_{\min}} - \sum_{t=r_{\min}+1}^{u_{\max}-1} y_t \tag{16}$$

$$= r_{\min}(y_{u_{\max}} - y_{r_{\min}}) + \sum_{t=r_{\min}+1}^{u_{\max}-1} (y_{u_{\max}} - y_t) \tag{17}$$

The difference $y_{u_{\max}} - y_{r_{\min}}$ is equal to n for all solutions since this is the number of jobs executed between the beginning and the end of the schedule. The first term to optimize is therefore a constant and can be ignored leaving only the expression $\sum_{t=r_{\min}+1}^{u_{\max}-1}(y_{u_{\max}} - y_t)$ to minimize or $\sum_{t=r_{\min}+1}^{u_{\max}-1}(y_t - y_{u_{\max}})$ to maximize.

Let $(a_1, a_2), (a_2, a_3), \ldots, (a_{k-1}, a_k)$ with $a_1 = u_{\max}$ and $a_k = t$ be the edges on the shortest path from $u_{\max}$ to t with total weight $\delta(u_{\max}, t)$. By substituting the inequalities (7)-(9), we obtain this relation.

$$\delta(u_{\max}, t) = \sum_{i=1}^{k-1} w(a_i, a_{i+1}) \geq \sum_{i=1}^{k-1} y_{a_{i+1}} - y_{a_i} = y_t - y_{u_{\max}} \tag{18}$$

By setting $y_{u_{\max}} = 0$ and $y_t = \delta(u_{\max}, t)$, we maximize the difference $y_t - y_{u_{\max}}$ up to reaching the equality. Since we maximize the difference for all values t, this maximizes $\sum_{t=r_{\min}}^{u_{\max}-1}(y_t - y_{u_{\max}})$ which is equivalent to minimizing $\sum_{t=r_{\min}}^{u_{\max}-1} t x_t$. ◀

▶ **Theorem 17.** *The algorithm for finding the starting times minimizes the makespan.*

Proof. Suppose the algorithm makes the latest job start at time m. We have $-\sum_{t=m}^{u_{\max}-1} x_t = y_m - y_{u_{\max}} < 0$. A schedule with smaller makespan would have $y_m - y_{u_{\max}} = 0$ which means this quantity would be greater than the one produced by the algorithm. However, following the argument in Theorem 16, the inequality (18) is maximized up to equality. Therefore, a schedule with a smaller makespan violates (7)-(9). ◀

6 Conclusion

We gave an algorithm which substantially improves over the previous best known ones for the problem of makespan and completion times minimization for multi-machine scheduling with tasks of equal length. We observed that the running time complexity depends on the relative sizes of m and p. An open question is to show whether this relationship is tight or whether there exists a better complexity in terms of n, m, and p.

References

1. R. E. Bellman. On a routing problem. *Quarterly Applied Mathematics*, 16:87–90, 1958.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. S. Stein. *Introduction to Algorithms*. MIT Press, 2001.
3. C. Dürr and M. Hurand. Finding total unimodularity in optimization problems solved by linear programs. *Algorithmica*, 2009. Published online on April 22nd, 2009, DOI 10.1007/s00453-009-9310-7.
4. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, 1983.

5 M.R. Garey, D.S. Johnson, B.B. Simons, and R.E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal on Computing*, 10(2):256–269, 1981.

6 P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, pages 26–30, 1935.

7 W. Lipski and F. P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15:329–346, 1981.

8 Michael Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, third edition, 2008.

9 B. Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal of Computing*, 12(2):294–299, May 1983.

10 B. Simons and M. K. Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM Journal of Computing*, 18(4):690–710, August 1989.

11 N. Vakhania. Scheduling equal-length jobs with delivery times on identical processors. *International Journal of Computer Mathematics*, 79(6):715–728, 2002.

12 J. Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general network. *Quarterly Applied Mathematics*, 27:526–530, 1970.