**CS787: Advanced Algorithms**

**Topic:** Scheduling with Precedence Constraints **Presenter(s):** James Jolly, Pratima Kolan

## 17.5.1 Motivation

### 17.5.1.1 Objective

Consider the problem of scheduling a collection of weighted tasks with precedence constraints, where the objective is produce a schedule that minimizes the sum of their weighted completion times.

Generating an optimal single-machine schedule for this task is NP-Hard. We discuss an algorithm that approximates the optimal schedule in polynomial time within a factor of 2. We then discuss a second approximation that converts a single-machine schedule into a multi-machine schedule in polynomial time.

### 17.5.1.2 A Precedence Graph as Tasks Waiting for Input

Many scheduling tasks in databases and operating systems involve reasoning about tasks that depend on the output of other tasks. A collection of these related tasks can be thought of as a graph, where each task is represented by a vertex and each depedency, or *precedence constraint*, is represented by an edge.

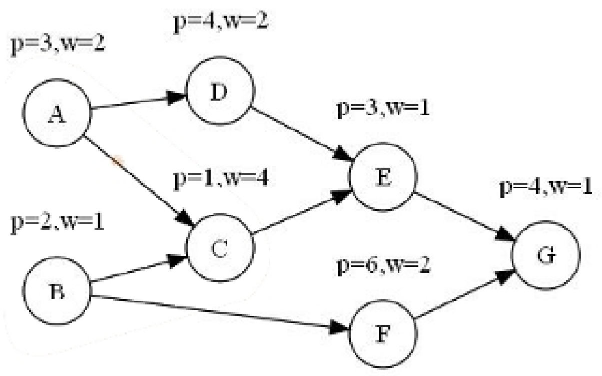


Figure 17.5.1.1: Example Precedence Graph

Often, in addition to precedence constraints, there is some notion of task priority. A system scheduling a collection of tasks might aim to run the most important ones first, while still obeying the precedence constraints between them.

This problem can be cast as finding a schedule that minimizes the sum of weighted completion times across tasks, where each weight represents the priority of its respective task.

### 17.5.2 Formalization

We are given a collection of $n$ tasks, $V = \{t_1, t_2, ..., t_n\}$, each with its own processing time $p_i$ and weight $w_i$. Let $c_i$ be the time $t_i$ finished (note that $p_i \leq c_i$).

Our objective is to find a *feasible schedule* that minimizes $\gamma(V) = \sum_{i=1}^{n} w_i c_i$, under precedence constraints. A feasible schedule is a list of tasks $V = \langle t_1, t_2, ..., t_n \rangle$, ordered from the first scheduled to the last scheduled, that does not violate precedence constraints.

We represent precedence constraints using a graph $G = (V, E)$, where $E$ contains the dependencies between tasks in $V$. We denote the dependency of task $j$ on the output of task $i$ as $i \rightarrow j$, read *$i$ preceeds $j$*. Note that the addition of precedence constraints to the weighted completion time problem have potential to increase $\gamma$ for a given $V$.

### 17.5.3 Single-Machine Scheduling With Precedence

This problem is NP-Hard when arbitrary precedence constraints are present. Here we present a 2-approximation discovered by Chekuri that runs in polynomial time [1].

#### 17.5.3.1 Rank of a Sub-DAG

As in many other approximation algorithms, here it is useful to order the inputs using a heuristic. We define the rank of a job $r_i$ as:

$$r_i = \frac{p_i}{w_i}$$

We may also wish to reason about the rank the of collection of tasks $T = \{t_1, t_2, ..., t_k\}$, defined as $R(T)$.

$$R(T) = \frac{\sum_{i=1}^{k} p_i}{\sum_{i=1}^{k} w_i}$$

Rank gives us some notion of which task or group of tasks we should schedule first. We also need to consider precedence constraints in this problem, and will need an additional construct to take them into account.

### 17.5.3.2 Precedence Closed Sub-DAG

A sub-DAG is precedence closed if all its tasks only depend on other tasks within itself. For any vertex $i \in V$, let $G_i$ denote the sub-DAG of $G$ induced by the set of vertices preceding $i$. More formally, we define a sub-DAG $G'$ of $G$ to be precedence closed when $\forall t_i \in G', G_i \in G'$.

Precedence closed sub-DAGs are important as we can schedule them by only considering the tasks inside them. Scheduling a sub-DAG with precedence constraints can be done in polynomial time using a topological sort (linear time in terms of the sum of edges and nodes).

### 17.5.3.3 G*, The Minimum Rank Precedence Closed Sub-DAG

To minimize $\gamma(V)$, we need to schedule tasks or groups of tasks in order of their rank while still satisfying precedence constraints. To do this, we need a fast way to identify the precedence closed group with lowest rank in the DAG, G*.

You might be able to guess what comes next: scheduling this group, removing it from the graph, and then finding the group with next lowest rank.

Chekuri tells us that there exists an optimal schedule for G in which an optimal schedul for $G^*$ occurs as a segment that starts at time 0.

For an arbitrary graph there are many possible groupings of nodes ($2^{|V|}$), many of which are not precedence closed. Thankfully, Chekuri has also devised a clever min-cut problem formulation that finds G* in polynomial time.

### 17.5.3.4 2-approximation Proof

In this algorithm, each precedence closed subgraph that does not contain a subgraph of lesser rank is subproblem. We can schedule the nodes in this subgraph using any arbitrary method (that obeys precedence constraints) and still acheive a 2-approximation.

$$\begin{aligned}
\text{OPT} &= \sum_j w_j C_j \\
&\geq \sum_j w_j \sum_{i \leq j} \alpha w_i = \alpha \left( \sum_j (w_j)^2 + \sum_{i \leq j} w_i w_j \right) \\
&= \alpha \left( (W(G))^2 + \frac{(W(G))^2}{2} \right) = \frac{\alpha (W(G))^2}{2} = \frac{P(G) W(G)}{2}
\end{aligned}$$

The last equality is true because $r(G^*) = r(G) = \frac{p(G)}{w(G)} = \alpha$

### 17.5.3.5 Finding G*

Here we construct a graph $G_\lambda$ with vertex set $V = T \cup \{(s,t)\}$. We add an edge from source $s$ to every job with cost on the edges equal to processing time of the job. Also, we add an edge from every job to the sink $t$ with cost on it equal to $\lambda w_i$.

Also if there is a precedence constraint between two vertices $t_1, t_2$ in G, then we add an edge from $t_2$ to $t_1$ having infinite cost in $G_\lambda$. By adding this edge we will be sure that we are not going to violate any precedence constraints during construction of a mincut.

To construct a G*, we solve a more general problem that if there exists any cut $(A, B)$ in $G_\lambda$ whose value is bounded $\lambda w(G)$ then subgraph $A - \{s\}$ is precedence closed and that the rank of $A - \{s\}$ is at most $\lambda$.

The main challenge here is to calculate the value of $\lambda$ which satisfies above property.

Here $\lambda$ ranges from $\lambda_{MIN}$ = lowest rank of the vertex and $\lambda_{MAX}$ = rank of the graph. Using binary search with in this range, we can find a minimal value of $\lambda$ that satisfies the above property. This process takes polynomial time.

Once we find an appropriate value of the $\lambda$, we have a precedence closed subgraph $G* = A - \{s\}$.

### 17.5.3.6 Algorithm Overview

At the highest level, we are repeatedly taking a new $G^*$ out of $G$ and scheduling it. We produce a schedule consisting of precedence closed graphs from lowest to highest rank.

Given a graph $G$, calculate $G^*$ (which takes polynomial time). If $G^*$ is a proper subset of $G$, then we recursively schedule both $G^*$ and $G - G^*$ separately.

If $G^* = G$, we can schedule the jobs in $G^*$ in any order (only taking into consideration the precedence constraints), getting a 2 approximation.

Note that if $(G^*)* = G^*$, it is enough to recurse only on $G - G^*$. Now total weighted completion time of G is $\gamma(G^*) + p(G^*)w(G - G^*) + \gamma(G - G^*)$.

Since we have a 2 approximation weighted completion time schedule to $G^*$, the total weighted completion time is at most 2 times the optimal weighted completion time.

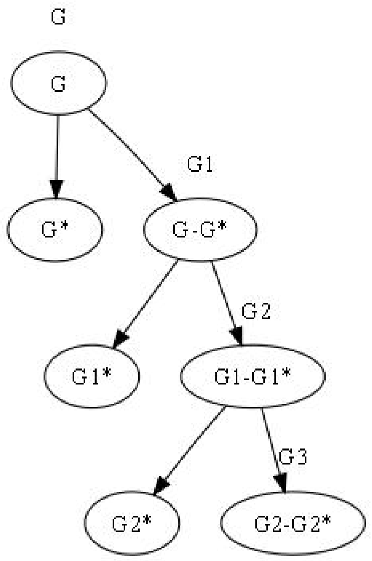
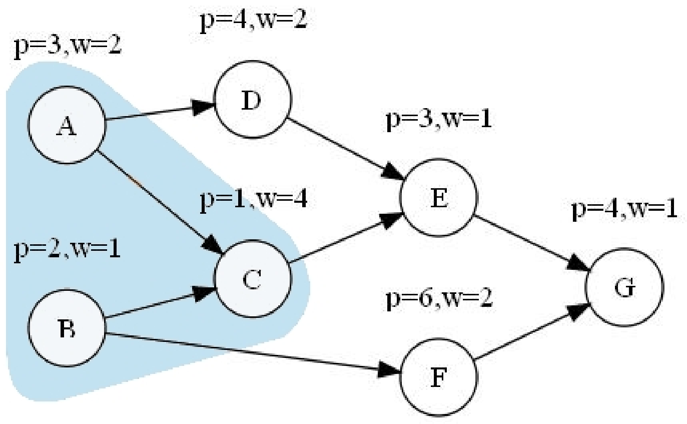
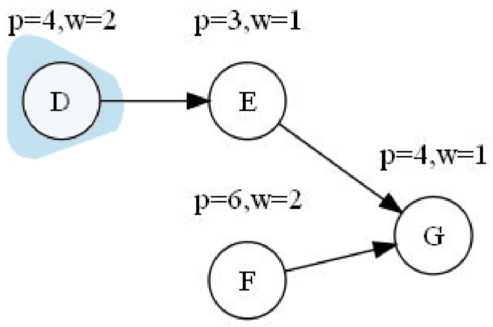


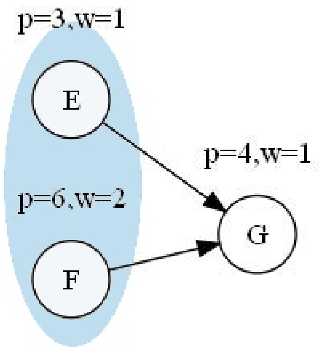
Figure 17.5.3.2: Breaking the Scheduling Problem in Subproblems

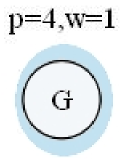
## 17.5.4 Example Run

To illustrate the entire process, we run the 2-approximation on the example DAG introduced earlier. In each of the 4 phases, $G^*$ is represented by the blue region.




p=4,w=2
D
p=3,w=1
E
p=4,w=1
G
p=6,w=2
F


p=3,w=1
E
p=4,w=1
G
p=6,w=2
F


p=4,w=1
G

### 17.5.5 Multi-Machine Scheduling With Precedence

Chekuri has developed an algorithm that converts any feasible single-machine schedule within a constant factor of OPT into a multi-machine schedule within a constant factor of OPT in polynomial time [2].

#### 17.5.5.1 New Considerations

Recall that a *feasible schedule* is a list of tasks $V = \langle t_1, t_2, ..., t_n \rangle$, ordered from the first scheduled to the last scheduled, that does not violate precedence constraints.

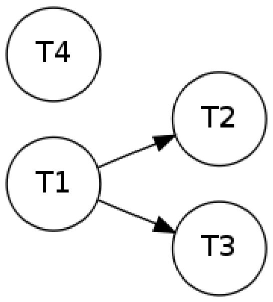
In the multi-machine case, we have a list of identical machines $M = \langle m_1, m_2, ..., m_k \rangle$. To minimize weighted completion time, we want to keep as many tasks running on these machines as possible. Unfortunately, precedence constraints may limit the degree of parallelism we can achieve.

#### 17.5.5.2 Delay List Algorithm Overview

```
$t = 0$
if a machine $m$ in $M$ is idle, then:
    if the first task $i$ in $V$ is ready:
        schedule $i$ on $m$
        mark all idle time up to start time of task $i$
    otherwise:
        scan through $V$, pick the first task $i$ that is ready
        if $\beta * P_i \leq$ sum of all unmarked idle time
            schedule task $i$ on $m$
    $t = t + 1$
```

#### 17.5.5.3 Example Conversion

Let's see Delay List in action. Suppose we have the following graph:



Suppose we have two processors, $M = \langle m_1, m_2 \rangle$. Consider the case where $w_{1 \ldots k} = 1$, $p_1 = 2$, $p_2 = 2$, $p_3 = 2$, and $p_4 = 3$, and we are given the feasible schedule $V = \langle t_1, t_2, t_3, t_4 \rangle$. Two obvious feasible parallel schedules are:

```
(A) [1][1][2][2][ ][ ][ ] and (B) [1][1][2][2][ ][ ][ ]
    [4][4][4][3][3][ ][ ]         [ ][ ][3][3][4][4][4]
```

Schedule B is true to the order of our single-machine schedule, while schedule A has scheduled task 4 out-of-order. Despite this, of the two schedules, A has a total cost of $2 + 4 + 3 + 5 = 14$, while B costs $2 + 4 + 4 + 7 = 17$.

How does delay list decide which schedule is better? The answer lies in the $\beta$ parameter. With a low $\beta$, delay list is biased against processor downtime, and is more willing to schedule a task out-of-order. When $\beta$ is high, delay list is more willing to accept downtime in an effort to adhere to the original single-machine schedule.

The above seems reasonable, as A has a shorter makespan than schedule B. When the task weights are the same, the cost of downtime is intuitive. If we change $w_3$ to 5 things get more interesting. Schedule A now costs $2 + 4 + 3 + 25 = 34$, while B costs $2 + 4 + 20 + 7 = 33$. What happened? We had to push task 3 one time unit later in schedule A. Our weighting now values it more than exposing the maximum amount of task parallelism. It's important to keep in mind that the original single-machine schedule may be cognisant of the weights of each job.

#### 17.5.5.4 Picking a Good $\beta$

We have touched on a difficult balacing act: how much do we wish to deviate from the order encoded in the single-machine schedule in order to maximize parallelism?

It depends. Given an optimal input schedule where task weights vary dramatically, it is better for Delay List to sacrifice some parallelism in order to prioritize certain jobs. On the other hand, with a poor feasible input schedule and low variance in the task weights, we might wish to parallelize as much as possible.

## References

[1] Chekuri, C., and Motwani, R. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics 98*, 1-2 (1999), 29 – 38.

[2] Chekuri, C., Motwani, R., Natarajan, B., and Stein, C. Approximation techniques for average completion time scheduling. *SIAM Journal on Computing 31*, 1 (2001), 146–166.