Scheduling Tasks with Exponential Duration on Unrelated Parallel Machines

Mostafa Nouri*, Mohammad Ghodsi†‡

Computer Engineering Department
Sharif University of Technology,
Tehran, Iran

Abstract

This paper introduces a stochastic scheduling problem. In this problem a directed acyclic graphs (DAG) represents the precedence relations among m tasks that n workers are scheduled to execute. The question is to find a schedule Σ such that if tasks are assigned to workers according to Σ, the expected time needed to execute all the tasks is minimized. The time needed to execute task t by worker w is a random variable expressed by a negative exponential distribution with parameter λ_{wt} and each task can be executed by more than one worker at a time. In this paper, we will prove that the problem in its general form is NP-hard, but when the DAG width is constant, we will show that the optimum schedules can be found in polynomial time.

Keywords: Stochastic parallel scheduling, multiple choice hyperbolic 0-1 programming

1 Introduction

Scheduling is the problem of allocating resources to a set of tasks over time, with the objective of optimizing some performance measures. By changing the types of resources and tasks, the objective measures and constraints, hundreds of useful problems can be formulated. Tasks compete for resources, such as CPU, memory, I/O devices, agents in a bank, etc. Tasks can also have different characteristics, such as ready times, due dates, relative urgency weights, etc. The relations between tasks can be defined in different ways. In addition, different criteria can be taken into account for measuring the quality of performance of any schedule.

In this paper, we have defined a new problem. In this problem, called *Memoryless Workers* (or MW), we have a set of resources $W = \{w_1, \ldots, w_n\}$, called workers, and a set of tasks $\mathcal{T} = \{t_1, \ldots, t_m\}$. The dependency relations between tasks are modeled using a directed acyclic graph (DAG) $\mathcal{G}$. The execution time

*nourybay@ce.sharif.edu
†ghodsi@sharif.edu
‡This authors research was partially supported by the IPM under grant No: CS1389-2-01.

needed for each task t executed on worker w is a random variable with exponential distribution with parameter λ_{wt}. The problem is to find an optimal schedule, such that the expected completion time for all tasks is minimum.

Our problem is much related to the problem introduced by Malewicz [14], called ROPAS[1], in which there is uncertainty in the successful completion of a task assigned to a worker. In ROPAS, there are a set of n workers $\{w_1, \ldots, w_n\}$, and a set of m unit-time tasks $\{t_1, \ldots, t_m\}$, whose interdependencies are modeled by a DAG $\mathcal{G}$. For each worker w and each task t, we are given a probability p_{wt}, which is the probability of successful completion of t when executed by w at any given step. The goal of ROPAS is to find a schedule to minimize the expected time to complete all the tasks.

The author of [14] proves that his problem is NP-hard and provides a polynomial-time solution to the problem when the precedence graph width and the number of workers are bounded.

1.1 Our results

- We will prove that MW is an NP-hard problem in the general case.
- We will propose a simple and optimal solution for multiple-choice hyperbolic 0-1 programming problems, which are a class of optimization problems in which objective function is a quotient of two linear functions and the variables are partitioned into some disjoint subsets and only one variable in each partition can be 1 and the others should be 0.
- We will propose a polynomial time algorithm for MW when the width of $\mathcal{G}$ is constant (independent of n, m and other parameters of the model). The solution is based on a reduction to multiple-choice hyperbolic 0-1 programming problem which is also solved in this paper. It is an interesting result that the scheduling complexity is polynomially dependent on the number of workers in contrast to ROPAS.

1.2 Related works

Scheduling has a rich background. Relatively simple scheduling algorithms are studied by researchers in operations research, industrial engineering and management to manage activities in a workshop, to lower the production cost in a manufacturing process, etc. in the 1950s. Later in the 1960s, computer scientists also encountered scheduling problems in the development of operating systems. Introducing complexity theorey, they realized that many scheduling problems may be inherently difficult to solve. In the 1970s many scheduling problems were shown to be NP-hard [1, 11, 12, 17].

Scheduling problems classified based on several factors. In a scheduling problem, the workers may be *parallel* or *dedicated*. Parallel workers do the same functions while dedicated workers are specialized for specific tasks. When the workers are parallel, they may be *identical*, i.e. they have equal speeds, or *uniform*, i.e. each worker has a constant speed, independent of the tasks, or they may be *unrelated* if the speed of each worker depends on the task it processes. If the workers are dedicated, there may be three different cases: *flow*

[1]Recomputation and Overbooking allowed Probabilistic dAg Scheduling

shop, *open shop* or *job shop*. In these cases, each job needs to be executed on several workers.

There may be some precedence constraints among tasks of $\mathcal{T}$. The tasks in $\mathcal{T}$ are called *dependent* if there are two tasks in $\mathcal{T}$, with restriction in their order of execution. If there are no such pairs of tasks, the tasks are called *independent*. Usually a task set with precedence relation is represented as a DAG, in which nodes correspond to tasks and arcs correspond to precedence constraints.

The problem of optimal scheduling of m tasks on n identical workers to minimize the completion time for all the tasks, when the preprocessing time of each task is 1, given any arbitrary dependency relations was proved to be NP-hard [21, 11]. However, the problem has polynomial time algorithm when the precedence relations form a tree [10] or the number of workers is 2 [6]. For this problem, the NP-hardness proof of Lenstra and Rinnooy Kan [11] implies that the best possible worst-case bound for a polynomial time approximated algorithm would be 4/3, unless P=NP. Most of other related problems are also proved to be NP-hard.

For scheduling problems with unit time tasks, that are NP-hard with regard to only arbitrary dependency graphs, we can find a polynomial time algorithm, if we assume the dependency graph has a bounded width w. This is because, in the process of scheduling tasks, at any time there will be at most 2^w possible combinations of tasks that can be executed. Obviously, if there are more parameters which make the problem NP-hard, assuming those parameters as constants, too, makes the problem tractable. Möhring [15] and Veltman [23] assumed DAG width is constant and using dynamic programming, obtained polynomial time algorithms for multiprocessor scheduling with communication delays.

Scheduling problems with uncertainty are categorized into *reactive scheduling*, *stochastic scheduling*, *scheduling under fuzziness*, *proactive (robust) fuzziness*, and *sensitivity analysis*. Among these categories, stochastic scheduling is the problem we tackle in this paper. In stochastic scheduling, we need to schedule tasks with uncertain durations in order to minimize the expected completion time of the tasks. For more information about the classification of scheduling problems under uncertainty refer to the survey by Herroelen and Leus [9].

Pinedo and Weiss [16] studied the problem of scheduling of m tasks on two identical parallel workers, when the preprocessing time of each task is an exponential random variable, and showed that *longest expected processing time (LEPT) first* policy minimizes the expected completion time of all the tasks. They also considered the problem of preemptive scheduling of m tasks on n uniform workers in [25], and studied the effect of assigning at every moment the task with shortest or longest expected processing time to the fastest available worker for various cost functions. There are many works on stochastic scheduling, with different criteria [2, 22, 24]. To see a full list of works on stochastic scheduling see [17].

In the work related to this paper, Malewicz [14] showed that the difficulty of ROPAS problem totally depends on two parameters of the instance, namely the precedence graph width and the number of workers. He proposed a polynomial time algorithm when both parameters are bounded, and proved that the problem is NP-hard in cases that either parameter is unbounded. He also showed that if both parameters are unbounded, the problem cannot be approximated within a factor less than 5/4. More recently, Lin and Rajaraman [13] found

an approximation algorithm for ROPAS for some special cases of dependency graphs with logarithmic approximation ratio.

1.3 Paper structure

The structure of the remaining sections is as follows. We first introduce the required concepts and define the new problem formally in Section 2. In Section 3 the algorithm for the problem with a restriction is presented. This restriction is on the value of DAG width, which we assume is constant. To solve the actual problem we first give a solution for Multiple-Choice Hyperbolic 0-1 programming. We also show two examples to demonstrate our method of scheduling. In Section 4, the complexity of the problem without restriction on the value of DAG width is discussed. Finally, Section 5 summarizes the obtained results.

2 Definitions

In this section we briefly review the required concepts and notations and then formally define the problems.

2.1 Required Concepts

Let $\mathcal{G} = (N_\mathcal{G}, E_\mathcal{G})$ represent a DAG where $N_\mathcal{G}$ is the set of nodes and $E_\mathcal{G}$ is the set of edges. An *antichain* in $\mathcal{G}$ is a subset of nodes $N_\mathcal{G}$ none of the nodes of which is an ancestor of another, i.e. no two are comparable with regard to the parent-child relation. A famous theorem of Dilworth [5], states that the size of the largest antichain is equal to the minimum number of paths covering all the nodes; i.e. having each node at least on one path. The size of the largest antichain is defined as the *width* of the graph.

In the parallel scheduling problem of our interest, tasks and their relations are modeled as a DAG. Nodes denote the tasks and edges represent the precedence relations between the tasks. If v is a child of u, then v can be started only if u has been finished. It means that a task can only be started if all its parents are already finished. Assuming the subset X of tasks has been completed, the subset $E(X)$ of tasks, called eligible tasks, is defined as all $u \in N_\mathcal{G} - X$, such that all parents of u are contained in X. In the next round, we can start any task in $E(X)$. A subset Y of tasks is said to satisfy precedence constraints if there is no task in Y whose parent is not.

2.2 Memoryless Workers

We now define *Memoryless Workers* problem or simply *MW*. As in ROPAS problem [14], there are n workers to perform m tasks. In contrast to ROPAS, in MW, the duration of executing task t by worker w is a random variable with *negative exponential* (simply exponential) distribution. Therefore the time needed for executing a task by a worker is not fixed (a unit time), instead worker w can execute task t within a time determined by an exponential distribution with parameter λ_{wt}.The goal is to find a schedule that minimizes the expected *completion time*. The completion time of a set of tasks is the total time needed to execute all the tasks.

Formally, given a subset X of accomplished tasks (initially empty), in each round the schedule Σ assigns some tasks in $E(X)$ to the workers, possibly assigning a single task to more than one worker, or assigning nothing to a worker. Σ is a function that maps $X \subseteq N_{\mathcal{G}}$ satisfying precedence constraints, to a function $f_X : W \to (N_{\mathcal{G}} \cup \{\perp\})$. By assigning tasks to the workers, the schedule waits until (at least) one of the workers executes its assigned task. After completion of one or more tasks, the accomplished tasks are added to X and Σ reassigns the new eligible tasks, $E(X)$, to the workers.

3 Scheduling Algorithms for Restricted Versions

As we will see in the next section, the general case of MW is NP-hard, so we must solve the problem with some restrictions that make it tractable in polynomial time. This restriction, as stated previously, is to limit the DAG width. In this section we focus on a polynomial time algorithm for restricted version of the problem, but before continuing, we first consider a programming problem which is useful for solving the actual problem.

3.1 Multiple-choice Hyperbolic 0-1 Programming

In this section we give a solution for multiple-choice hyperbolic 0-1 programming (MCHP). Based on this solution, in the next section we will solve MW easily. The multiple-choice hyperbolic 0-1 programming can be defined as following:

$$\text{(MCHP)} \qquad \text{Minimize } z(x) = \frac{a_0 + \sum_{j=1}^{m} \sum_{i=1}^{n} a_{ij} x_{ij}}{b_0 + \sum_{j=1}^{m} \sum_{i=1}^{n} b_{ij} x_{ij}}$$

$$\sum_{j=1}^{m} x_{ij} \leq 1 \qquad i = 1, \ldots, n$$

$$x_{ij} \geq 0, \text{ integer} \qquad i = 1, \ldots, n, \ j = 1, \ldots, m.$$

MCHP is a generalization of unconstrained hyperbolic 0-1 programming. The unconstrained hyperbolic 0-1 program consists of optimizing the sum of ratios of linear functions of binary variables. Hammer and Rudeanu [7] studied the single ratio version of the unconstrained hyperbolic 0-1 program. They showed that when all coefficients are non-negative, any local optimum solution of the problem is also a global optimum solution. They also gave two polynomial algorithms for the problem in that case. Robillard [20] improved the running time of the algorithm to solve the problem.

Later, Hansen *et al.* [8] proposed a linear time algorithm that can solve the problem in efficient time assuming that the denominator of the objective function is always positive. They showed that if the sign of the denominator is unrestricted, the problem is NP-complete.

Prokopyev *et al.* [18, 19] considered both single-ratio and multiple-ratio unconstrained hyperbolic 0-1 programmings. They proved that checking whether these problems have a unique solution is NP-hard. They also showed that finding the global maximizer of problems even with unique solution is NP-hard.

In order to solve MCHP, we use a linear 0-1 program in the same way Hansel *et al.* used for their problem. Since unconstrained hyperbolic 0-1 programming

is a special case of MCHP, the claim of Hansel *et al.* [8] about the sign of the denominator still holds for MCHP. This linear 0-1 program is shown in the following Theorem.

Theorem 3.1 *Assume the denominator of the objective function of MCHP is always positive. Let z^* denote the optimal value of MCHP. x^* is the optimal solution of MCHP iff it is an optimal solution of the following multiple-choice linear 0-1 programming problem (MCLP):*

$$\begin{aligned} \textit{(MCLP)} \qquad \textit{Minimize } y(x) &= a_0 + \sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}x_{ij} \\ &\quad - z^*\big(b_0 + \sum_{i=1}^{n}\sum_{j=1}^{m} b_{ij}x_{ij}\big) \qquad (1) \\ \sum_{j=1}^{m} x_{ij} &\le 1 \qquad\qquad i = 1,\ldots,n \\ x_{ij} &\ge 0,\ \textit{integer} \qquad i = 1,\ldots,n,\ j = 1,\ldots,m. \end{aligned}$$

Proof. Let x^* be a 0-1 n-vector. Since $b_0 + \sum_{i=1}^{n}\sum_{j=1}^{m} b_{ij}x_{ij} > 0$, dividing (1) by $b_0 + \sum_{i=1}^{n}\sum_{j=1}^{m} b_{ij}x_{ij}$ shows that $y(x^*) < 0 \iff z(x^*) < z^*$, which contradicts the definition of z^*; $y(x^*) = 0 \iff z(x^*) = z^*$; and $y(x^*) > 0 \iff z(x^*) > z^*$. Therefore x^* is an optimal solution of MCHP if and only if it is an optimal solution of MCLP. □

Theorem 3.2 *The optimal value of MCLP can be found by a piecewise linear function of z^*.*

Proof. Let z^* be a fixed value. In order to minimize (1), for each $i = 1,\ldots,n$, we should select the best element (a_{ij}, b_{ij}) from the set $S_i = \{(a_{ij}, b_{ij}) | j = 1,\ldots,m\}$ such that the value of $a_{ij} - b_{ij}z^*$ is the minimum value among all possible values, while it is also negative. If for all elements of S_i, this value is non-negative, we will not select anything from S_i, corresponding to the case that all $x_{ij} = 0$ for $j = 1,\ldots,m$.

Now consider the following function:

$$F(z) = a_0 - b_0 z + \sum_{i=1}^{n} f_i(z),$$

where $f_i(z) = \min\big\{\min_{j=1}^{m}\{a_{ij} - b_{ij}z\}, 0\big\}$.

Obviously for each possible value of z, which is the set of real numbers, $\mathbb{R}$, $F(z)$ determines the optimal value of MCLP.

$f_i(z)$ is the minimum of m linear functions, so $f_i(z)$ is a continuous piecewise linear function with at most $m-1$ breaking points. $f_i(z)$ is also a concave function, i.e. any segment connecting two points of $f_i(z)$ completely lies below $f_i(z)$. This is because each linear function is concave and the minimum of a set of concave function is also concave. Therefore $F(z)$, which is the sum of these continuous, piecewise linear, concave functions is also a continuous, piecewise linear, concave function with at most $n(m-1)$ breaking points.

To specify the function $F(z)$ explicitly, we find all the distinct maximal intervals in which $F(z)$ is linear. In each of these intervals $F(z)$ is defined as a linear function $c - dz$. The value of c, d is determined by a particular selection of (a_{ij}, b_{ij})'s. In addition, each two adjacent intervals have different expressions, because they differ in selection of at least one (a_{ij}, b_{ij}). For each interval we find the best selection of (a_{ij}, b_{ij})'s and assign $x_{ij} = 1$ for all the selected (a_{ij}, b_{ij})'s and $x_{ij} = 0$ for all the others, and compute $F(z)$ for this assignment.

The function $F(z)$ gives us the optimal value of $y(x)$ in MCLP for each given z. As we saw in the proof of Theorem 3.1, the optimal solution x^* of MCHP is the optimal solution of MCLP when the optimal value $y(x^*)$ of MCLP is 0. Therefore we need to search for the smallest value of z such that $F(z) = 0$. This can be easily done by checking each linear part of $F(z)$ and see if it intersects the x-axis. If this is the case, the value of z for which $F(z) = 0$, is a candidate for the optimal value of MCHP. The exact solution is the minimum value among all such candidates.

Because $F(z)$ is a concave function, there are at most two values z such that $F(z) = 0$. We can use concavity property of $F(z)$ to speed up searching for the optimal value of z. The process of finding the intersections of $F(z)$ with line $y = 0$ can be accomplished in $O(\log n)$ [4]. □

The above theorem leads us to algorithm SolveMCHP for MCHP problem which can be seen in Algorithm 1.

Algorithm 1 The algorithm for solving multiple choice hyperbolic 0-1 programs.

1: **function** SolveMCHP(a_0, b_0, A, B)
   ▷ A is a $n \times m$ matrix stores a_{ij} for each $i = 1, \ldots, n$, $j = 1, \ldots, m$
   ▷ B is a $n \times m$ matrix stores b_{ij} for each $i = 1, \ldots, n$, $j = 1, \ldots, m$
2: **for** $i \leftarrow 1$ to n **do**
3: $LE_i \leftarrow$ ComputeLowerEnvelope$(A[i], B[i])$.
   ▷ $A[i]$ is the i-th row of A
   ▷ $B[i]$ is the i-th row of B
4: **end for**
5: $F \leftarrow a_0 - b_0 z + \sum_{i=1}^{n} LE_i$
6: $(x, 0) \leftarrow$ The leftmost intersection of F with line $y = 0$.
7: if $(x, 0)$ exists, **then return** x.
8: **end function**

To complete the algorithm SolveMCHP, we should describe the method for computing the linear pieces of $f_i(z)$ efficiently. It is easy to see that they can be computed in $O(n^2)$ by comparing each line $a_{ij_1} - b_{ij_1} z$ with all other lines $a_{ij_2} - b_{ij_2} z$ and finding the interval in which $a_{ij_1} - b_{ij_1} z$ is the minimum compared to other lines. But we can do much better by the following geometric approach.

Draw all lines $a_{ij} - b_{ij} z$ for all $j = 1, \ldots, m$. We want to compute the piecewise linear lower envelope of these m lines. Consider dualizing these lines. Each line $l_{ij} = a_{ij} - b_{ij} z$ will be mapped to point $l'_{ij} = (-b, -a)$. The lower envelope of those lines will be mapped to the upper hull of the convex hull of these points [4]. It is enough to compute the upper hull of the convex hull of

points l'_{ij} for $j = 1, \ldots, m$. The order of l'_{ij} in the convex hull determines the order of lines l_{ij} in the lower envelope. ComputeLowerEnvelope which is presented in Algorithm 2 can be used for computing the lower envelope.

Algorithm 2 The algorithm for finding lower envelope of a piecewise linear function.

```
1: function ComputeLowerEnvelope(A, B)
                                  ▷ A is a vector stores a_ij for each j = 1, ..., m
                                  ▷ B is a vector stores b_ij for each j = 1, ..., m
2:     LE ← ∅.  ▷ Sorted list of lower envelope lines and break points of f_i(z)
3:     CH ← ConvexHull((−b_{i,0}, −a_{i,0}), ..., (−b_{i,m}, −a_{i,m}), (0, 0)).
4:     for each vertex (p, q) in the upper hull of CH do
5:         insert line y = px − q into LE.
6:     end for
7:     for each two adjacent lines k : y = px − q and l : y = rx − s in LE do
8:         insert point ((q−s)/(p−r), (qr−ps)/(p−r)) between k, l.
9:     end for
10:    return LE.
11: end function
```

Theorem 3.3 *Algorithm* SolveMCHP *determines an optimal solution of problem MCHP in* $O(nm \log(nm))$.

Proof. Algorithm ComputeLowerEnvelope runs in $O(m \log m)$ since it will compute the convex hull of m points in $O(m \log m)$ and then for each edge ($O(m)$) will produce a break point.

Algorithm SolveMCHP uses ComputeLowerEnvelope for each i so this will be done in $O(nm \log m)$. Then it merges B_i's into B which are n sorted lists of size $O(m)$ in $O(nm \log n)$ [3] and finally for each interval computes the ratio for the selection of x_{ij}'s. If we compute the ratio by simply summing the required terms, the total running time will be $O(n^2 m)$, but since the differences between each two adjacent intervals are at most in the value of two variables x_{ij_1} and x_{ij_2}[2], we can compute the new ratio by simply a subtraction and an addition in the numerator and the denominator of the previous ratio. Thus the total running time will be $O(nm \log m + n + nm \log n) = O(nm \log(nm))$. □

3.2 Scheduling Algorithm for MW Problem

In this section, we give a solution for MW problem, based on MCHP. The following lemma, proves that the minimum expected time to completion of tasks across all schedules, denoted by B_X is finite when the schedule starts with tasks X already executed. Then, we give a schedule for executing the tasks and prove that it has the minimum expected time to completion of tasks.

Lemma 3.4 *For any set X of tasks that satisfies precedence constraints, B_X is finite.*

[2] if two or more $f_i(z)$ have a common break point, the two adjacent intervals differ in more than two values of x_{ij}, but in this case the total number of intervals is substantially reduced.

Proof. The proof is very simple, and similar to the proof of Lemma 2.1 in [14]. The general idea is to assign all the workers to each task one after another and show the expected execution time of this schedule is finite, and so is B_X. $\square$

For a set of tasks, X, satisfying precedence constraints, in order to compute B_X, we must derive the relationship between B_X and $B_{X'}$ for any possible executed tasks. This relation helps us to decide in each iteration how to assign the tasks to the workers. The next theorem states this relation.

Theorem 3.5 *Consider a schedule Σ, a set X of tasks that satisfies precedence constraints and does not contain all sinks of $\mathcal{G}$. Let the expected time to completion for the schedule starting with tasks X already executed be finite. Let $\tau(w)$ be the task that is assigned to w by $\Sigma(X)$. Then*

$$T_X = \frac{1 + \sum_{w \in W} (T_{X \cup \tau(w)} . \lambda_{w\tau(w)})}{\sum_{w \in W} \lambda_{w\tau(w)}}, \tag{2}$$

In the above equation, when nothing is assigned to w, we set $\lambda_{w\tau(w)} = 0$

Proof. Let us first describe the meaning of Equation 2. Σ assigns a task from $E(X)$ to each worker. Because the distribution of executing a task by each worker is exponential, and thus memoryless, the expected execution time of each task from now, given it has not been finished yet, is equal to the expected execution time of that task from the beginning. Therefore whenever a worker finishes a task successfully, Σ tries to reassign the new set of eligible tasks to the workers.

Let Y_w denote the required time to execute $\tau(w)$ by w. Y_w is a random variable with exponential distribution with parameter $\lambda_{w\tau(w)}$. The required time to finish the first task among all assigned tasks is $Y = \min_{w \in W}\{Y_w\}$. Y is a random variable with exponential distribution with parameter $\sum_{w \in W} \lambda_{w\tau(w)}$. The expected value of Y is $1/\sum_{w \in W} \lambda_{w\tau(w)}$, which is the first term in Equation 2. T_X is the sum of the expected value of Y and the expected time to finish the remaining tasks. The probability that worker w finishes its task before all other workers, is

$$\frac{\lambda_{w\tau(w)}}{\sum_{w' \in W} \lambda_{w'\tau(w')}}.$$

If we multiply this probability to the expected execution time for the remaining tasks, assuming $\tau(w)$ is finished, and sum over all workers, we will have the expected execution time for the remaining tasks. Therefore the expected time to execute the remaining tasks will be

$$\frac{\sum_{w \in W} (T_{X \cup \tau(w)} . \lambda_{w\tau(w)})}{\sum_{w \in W} \lambda_{w\tau(w)}},$$

which is the second term in Equation 2.

We now give a formal proof for the theorem. Let R_X denote the time needed to execute all tasks, starting with tasks X already executed. We can write the

probability density function (pdf) of R_X as

$$f_{R_X}(t) = \sum_{w \in W} f_{Y=Y_w}(t - T_{X \cup \tau(w)})$$
$$= \sum_{w \in W} \left(f_{Y_w}(t - T_{X \cup \tau(w)}). \prod_{u \neq w} P(Y_u > t - T_{X \cup \tau(w)}) \right)$$

In the above equation, the term $f_{Y=Y_w}(t - T_{X \cup \tau(w)})$ is the probability density of success of w before all other workers exactly at time $t - T_{X \cup \tau(w)}$, $f_{Y_w}(t - T_{X \cup \tau(w)})$ is the probability density that worker w finishes its assigned task at exact time $t - T_{X \cup \tau(w)}$ and $P(Y_u > t - T_{X \cup \tau(w)})$ is the probability that worker u finishes its task after time $t - T_{X \cup \tau(w)}$. Therefore we consider the possibility of success of each worker and sum up the probabilities.

Based on f_{R_X}, we can easily compute the expected value of R_X, which is T_X.

$$
\begin{aligned}
E[R_X] &= \int_0^\infty t.f_{R_X}(t)dt \\
&= \int_0^\infty t \sum_{w\in W} \left(f_{Y_w}(t - T_{X\cup\tau(w)}). \prod_{u\neq w} P(Y_u > t - T_{X\cup\tau(w)}) \right) dt \\
&= \sum_{w\in W} \int_0^\infty t \left(f_{Y_w}(t - T_{X\cup\tau(w)}). \prod_{u\neq w} P(Y_u > t - T_{X\cup\tau(w)}) \right) dt \\
&= \sum_{w\in W} \int_0^\infty (t + T_{X\cup\tau(w)}) \left(f_{Y_w}(t). \prod_{u\neq w} P(Y_u > t) \right) dt \\
&= \sum_{w\in W} \int_0^\infty ((t + T_{X\cup\tau(w)}).\lambda_{w\tau(w)} e^{-t\lambda_{w\tau(w)}} \prod_{u\neq w} e^{-t\lambda_{u\tau(u)}})dt \\
&= \sum_{w\in W} \int_0^\infty ((t + T_{X\cup\tau(w)}).\lambda_{w\tau(w)} e^{-t\sum_{u\in W}\lambda_{u\tau(u)}})dt \\
&= \sum_{w\in W} \int_0^\infty t.\lambda_{w\tau(w)} e^{-t\sum_{u\in W}\lambda_{u\tau(u)}} dt \\
&\qquad + \sum_{w\in W} \int_0^\infty T_{X\cup\tau(w)}.\lambda_{w\tau(w)} e^{-t\sum_{u\in W}\lambda_{u\tau(u)}} dt \\
&= \sum_{w\in W} \frac{\lambda_{w\tau(w)}}{(\sum_{u\in W}\lambda_{u\tau(u)})^2} + \sum_{w\in W} (T_{X\cup\tau(w)} \frac{\lambda_{w\tau(w)}}{\sum_{u\in W}\lambda_{u\tau(u)}}) \\
&= \frac{\sum_{w\in W}\lambda_{w\tau(w)}}{(\sum_{u\in W}\lambda_{u\tau(u)})^2} + \frac{\sum_{w\in W} T_{X\cup\tau(w)}.\lambda_{w\tau(w)}}{\sum_{u\in W}\lambda_{u\tau(u)}} \\
&= \frac{1 + \sum_{w\in W}(T_{X\cup\tau(w)}.\lambda_{w\tau(w)})}{\sum_{w\in W}\lambda_{w\tau(w)}}
\end{aligned}
$$

and hence Equation 2 holds. □

Using Equation 2, we can discuss about the optimum scheduling. We try to express this optimization with a multiple-choice hyperbolic 0-1 programming. In the previous section, we showed how to solve this problem efficiently.

Let $x_{wt} = 1$ if the worker w is assigned to task t and $x_{wt} = 0$ otherwise. In other words, for $t = \tau(w)$, $x_{wt} = 1$, and for $t \neq \tau(w)$, $x_{wt} = 0$ otherwise. Obviously $\tau(w) \in E(X)$, so we can rewrite Equation 2 using these new variables as

$$T_X = \frac{1 + \sum_{w \in W} \sum_{t \in E(X)} T_{X \cup \{t\}}.\lambda_{wt}.x_{wt}}{\sum_{w \in W} \sum_{t \in E(X)} \lambda_{wt}.x_{wt}}. \quad (3)$$

We have to add some additional constraints such that for each worker w, there exists at most one t with $x_{wt} = 1$ and the others must be 0. The following constraints fulfill what we need:

$$\sum_{t \in E(X)} x_{wt} \leq 1 \qquad \forall w \in W, \quad (4)$$

$$x_{wt} \geq 0, \text{ integer} \qquad \forall w \in W, \forall t \in E(X). \quad (5)$$

Equations 3-5, demonstrates the characteristic of every schedule, when the set of finished tasks is X. Each different combination of 0 and 1 for variables x_{wt}, gives a schedule and each schedule can be characterize with a unique sequence of 0 and 1. So the problem of finding the optimum schedule, when we have finished tasks in X, is equivalent to finding the minimum value for T_X, given Equations 3-5.

This will lead us to a multiple-choice hyperbolic 0-1 programming, in which we try to minimize T_X, given values $T_{X \cup \{t\}}$, for each $t \in E(X)$. In the previous section we gave an algorithm to solve MCHP problems. We can use that algorithm to solve the problem of scheduling MW. For $W = \{w_1, \ldots, w_n\}$ and $E(X) = \{t_1, \ldots, t_k\}$, the input parameters of MCHP are as follows: $a_{ij} = \lambda_{w_i t_j}.T_{X \cup \{t_j\}}$, $b_{ij} = \lambda_{w_i t_j}$, $a_0 = 1$ and $b_0 = 0$. Since $\lambda_{w_i t_j} > 0$, the denominator of the objective function is always positive, unless nothing is assigned to any worker, which is a useless schedule. The details of the algorithm for finding the optimum value of T_X, given values of $T_{X \cup \{t\}}$ can be seen in Algorithm 3.

As it can be seen, the optimum values of $T_{X \cup \{t\}}$ for all tasks $t \in E(X)$ are required to find the optimum value of T_X. We compute these values using dynamic programming, similar to the method used for ROPAS by Malewicz [14]. For this, we create a graph $\mathcal{A}$, called *Admissible evolution graph*, and for each different set X of executed tasks we add a node to $\mathcal{A}$, and for each set of executed tasks X and each task $t \in E(X)$, we add an edge from node X to node $X \cup t$. We start from node $\mathcal{T}$ with $T_{\mathcal{T}} = 0$, which is the node with all tasks executed before, we compute the expected time to execute unfinished tasks for each node. The procedure continues until we compute $T_\emptyset$ which we are looking for.

The difference between Malewicz's method and ours is in the depedency of T_X to different assignments of tasks to workers. In other words Malewicz needs to consider different assignments of tasks to workers and then selects the best one, which has exponential dependency to the number of workers, but in our method, in each state, based on the result of MCHP, we select the best assignments of tasks to workers.

The running time of our algorithm, as in ROPAS, grows exponentially when the width of $\mathcal{G}$ increases. This is unavoidable, as we will see in the next section, because if the DAG width is unbounded, WM will be NP-hard. Therefore we assume the DAG width is limited by a constant and cannot increase arbitrarily.

We now give a bound on the running time of the algorithm. Let d be the width of $\mathcal{G}$. By Dilworth's Theorem [5], there are d chains that cover $\mathcal{G}$

Algorithm 3 The algorithm for finding optimum schedule for Memoryless Workers problem.

```
1: function ComputeMinimumDuration(E(X), Λ, T_{X+})
                                              ▷ E(X) = {t_1, ..., t_k}
                  ▷ Λ is a n × k matrix stores λ_{wt} for each w ∈ W, t ∈ E(X)
                                  ▷ T_{X+} is a vector stores T_{X∪{t}} for each t ∈ E(X)
2:     A ← [Λ[i, j]T_{X+}[j]]
                                                  ▷ A is a matrix, with item in i-th row
                                  ▷ and j-th column equals to λ_{w_i t_j}.T_{X∪t_j}

3:     for i ← 1 to n do
4:         LE_i ← ComputeLowerEnvelope(A[i], Λ[i]).
                                                  ▷ A[i] is the i-th row of A
                                                  ▷ Λ[i] is the i-th row of Λ
5:     end for
6:     F ← 1 + Σ_{i=1}^{n} LE_i
7:     (x, 0) ← The leftmost intersection of F with line y = 0.
8:     if (x, 0) exists, then return a.
9: end function
```

completely. Let X be a subset of tasks that satisfies precedence constraints. If a task t in a chain is in X, then all preceding tasks in that chain are also in X. The maximum size of a chain is m, so there are at most $(m+1)^w$ distinct subsets X that satisfy precedence constraints, which is the number of states for dynamic programming. For each X, the number of different values $T_{X\cup\{t\}}$ that must be computed before, so that T_X is computable is $|E(X)|$ which is at most w. Since the time needed to compute T_X while we know all required values $T_{X\cup\{t\}}$, is $O(n|E(x)|\log(n|E(x)|))$, which is the running time of MCHP, the total time needed for finding the optimum schedule is at most $O((m+1)^d nd\log(nd)) = O(nd\log(nd)m^d)$, which is polynomial when d is bounded. This yields the following theorem.

Theorem 3.6 *There is a polynomial time algorithm for solving MW Problem when the width of $\mathcal{G}$ is bounded by a constant. The dependency of running time of the algorithm to the number of workers is $O(n\log n)$.*

3.3 Example 1: An Interesting Special Case

Consider the case in which each worker executes all tasks similarly; i.e. for $w \in W$, $t \in T$, $\lambda_{wt} = \lambda_w$. This example is more general than the example used by Malewicz [14], because in his example, he chose a constant p, such that the probability of executing each task successfully by each worker is p.

Let X be the set of tasks finished, and $E(X) = \{t_1, \ldots, t_k\}$ be the set of eligible tasks. By definition,

$$\begin{aligned} F(z) &= a_0 - b_0 z + \sum_{i=1}^{n} \min\left\{ \min_{j=1}^{k}\{a_{ij} - b_{ij}z\}, 0 \right\} \\ &= 1 + \sum_{w\in W} \min\left\{ \min_{t\in E(X)}\{\lambda_w T_{X\cup\{t\}} - \lambda_w z\}, 0 \right\}. \end{aligned}$$

In the above equation, the inner min function, selects the item with the lowest value for $T_{X\cup\{t\}}$, which means that worker w is assigned to task t, such that $T_{X\cup\{t\}}$ is the smallest. This is true for all workers, so in this iteration, all workers are assigned to a specific task, such that, when the task is finished, the minimum completion time is the smallest.

The interesting conclusion here is that, the order of the assignment in this example is not important, as long as the dependency relations between tasks are satisfied; i.e. we can assign all workers to t_1, then to t_2 and so on. This is because in each step all workers are assigned to the same task, and this makes the order of assignment unimportant. Therefore we can compute the optimum execution time, by simply computing the expected time to execute each individual task by all workers and summing up these values. This can be done in $O(mn)$.

3.4 Example 2

Consider a set of 5 tasks with dependency relations $\mathcal{G}$ in Figure 1.a and two workers that can execute the tasks with parameters listed in Figure 1.b. Figure 1.c shows the admissible evolution of executions of the tasks, $\mathcal{A}$. Each node in $\mathcal{A}$, represents a set X that satisfies precedence constraints. From each node, there are outgoing edges to other nodes associated to sets of finished tasks after one step. In each node, the optimum time for executing all tasks, starting from black tasks in that node executed, is written inside it. The optimum schedule for each node is also shown by assigning each worker to a task among eligible tasks in that node.

Let us describe computing the optimum schedule for a node. Consider the second node in the fourth level of $\mathcal{A}$, with expected completion time equal to 0.489. In this node, each worker has three option to work on: t_3, t_4, t_5. In this node, we have $f_1(z) = \min\{1\times 0.278 - 1\times z, 1\times 0.358 - 1\times z, 4\times 0.403 - 4\times z, 0\}$, which is

$$f_1(z) = \begin{cases} 0 & \text{if } z \leq 0.278, \\ 0.278 - z & \text{if } 0.228 < z \leq 0.445, \\ 1.612 - 4z & \text{if } 0.445 < z. \end{cases}$$

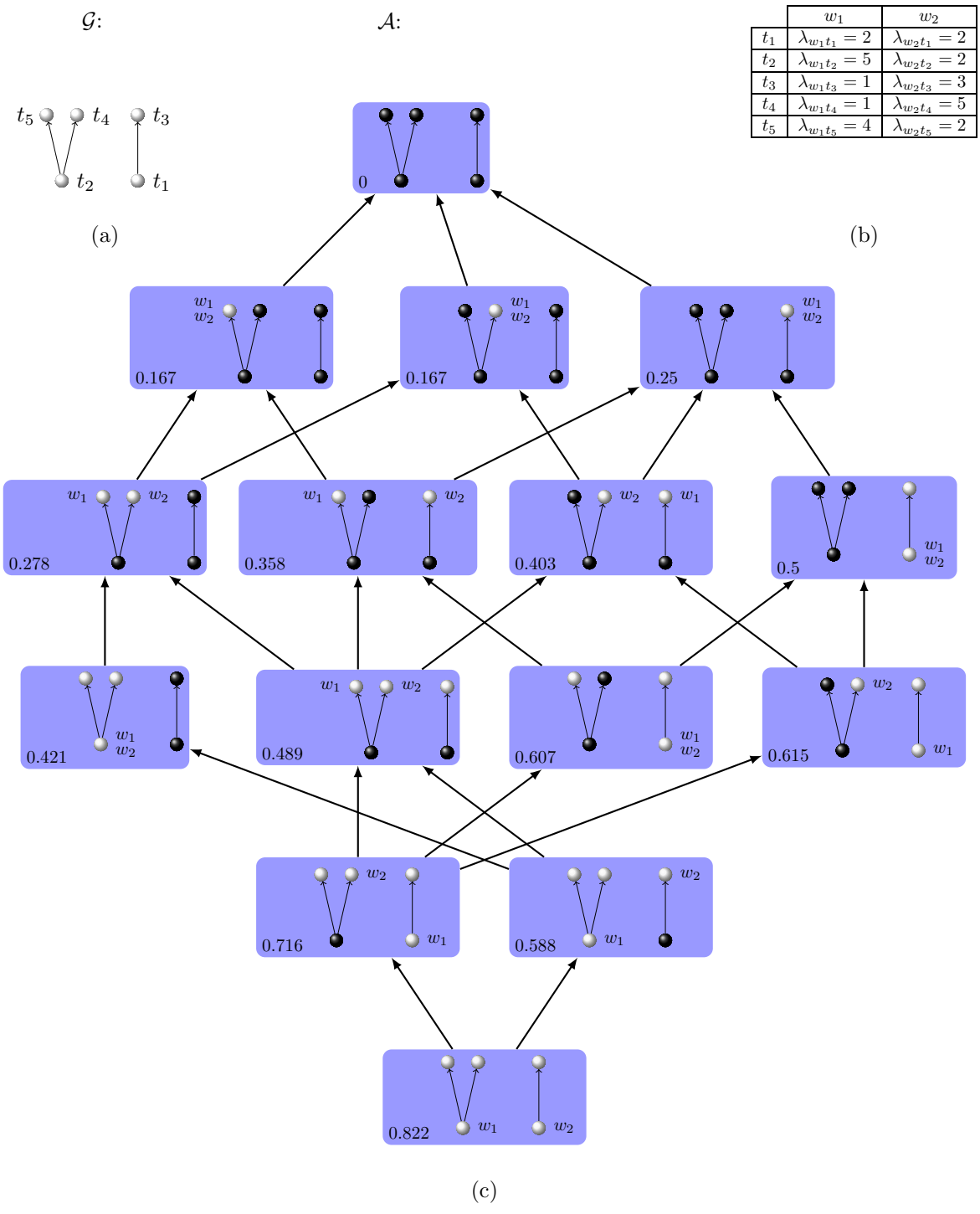
For the second worker we have $f_2(z) = \min\{3 \times 0.278 - 3 \times z, 5 \times 0.358 - 5 \times z, 2 \times 0.403 - 2 \times z, 0\}$, or

$$f_2(z) = \begin{cases} 0 & \text{if } z \leq 0.278, \\ 0.834 - 3z & \text{if } 0.278 < z \leq 0.478, \\ 1.79 - 5z & \text{if } 0.478 < z. \end{cases}$$

We can rewrite function $F(z) = 1 + f_1(z) + f_2(z)$ as

$$F(z) = \begin{cases} 1 & \text{if } z \leq 0.278, \\ 2.112 - 4z & \text{if } 0.278 < z \leq 0.445, \\ 3.446 - 7z & \text{if } 0.445 < z \leq 0.478, \\ 4.402 - 9z & \text{if } 0.478 < z. \end{cases}$$

Solving equation $F(z) = 0$, gives $z = 0.489$, which shows the best task for w_1 and w_2 are t_3 and t_2, respectively. The optimal schedule and expected execution time for the node are shown in the graph.



	w_1	w_2
t_1	$\lambda_{w_1 t_1} = 2$	$\lambda_{w_2 t_1} = 2$
t_2	$\lambda_{w_1 t_2} = 5$	$\lambda_{w_2 t_2} = 2$
t_3	$\lambda_{w_1 t_3} = 1$	$\lambda_{w_2 t_3} = 3$
t_4	$\lambda_{w_1 t_4} = 1$	$\lambda_{w_2 t_4} = 5$
t_5	$\lambda_{w_1 t_5} = 4$	$\lambda_{w_2 t_5} = 2$

Figure 1: (a) Dependency graph between tasks, $\mathcal{G}$. (b) Parameters for execution of tasks by workers. (c) Admissible evolution of execution, $\mathcal{A}$, for $\mathcal{G}$.

Using graph $\mathcal{A}$, we can easily apply the optimum schedule for executing tasks. We start from the source of $\mathcal{A}$, in which none of tasks are executed, and assign workers based on its optimum schedule. After we saw the first success in executing a task by the workers, we switch to the associated node in $\mathcal{A}$ and choose the optimum schedule for that node, and so on.

It can be easily seen that $\mathcal{A}$ in Figure 1.c is similar to the admissible evolution graph of execution in Malewicz's example [14]. In spite of this resemblance, which is the result of similarity in dependency relations between tasks in these two examples, the required time for computing the optimum schedule for MW is much less than computing it for ROPAS. This is because in each node in ROPAS, we need to consider all possible assignments of workers to eligible tasks and select the best assignment, which grows exponentially as the number of workers increases, while in WM, it has complexity $O(n \log n)$, where n is the number of workers.

4 Complexity of Scheduling

In this section we discuss about the complexity of the introduced scheduling problem. As we described earlier, we must restrict the DAG width so that the problem can be tractable, otherwise it is NP-hard. We will show that if the number of workers is restricted to two while the DAG width can grow, MW is NP-hard. Because of the similarity of WM to ROPAS, the process is basically like the proof of NP-hardness of ROPAS when the number of workers is two, with some modifications. This can be proved using an auxiliary problem that has been defined and proved to be NP-complete by Malewicz [14]. For convenience we mention the problem here.

> **Fixed Ratio Many Subsets with Small Union (FIRMSSU)**
>
> *Instance*: Number k, nonempty subsets $S_1, \ldots, S_{3k}$ of the set $\{1, \ldots, 3k\}$ whose union is $\{1, \ldots, 3k\}$.
>
> *Question*: Are there $2k$ of these subsets whose union has cardinality at most $2k$?

We use this problem and prove the NP-hardness of scheduling Memoryless Workers problem.

Theorem 4.1 *Scheduling Memoryless Workers problem restricted to two workers while the DAG width can grow is NP-hard.*

Proof. We reduce an instance of FIRMSSU to an instance of MW by a polynomial reduction, and based on the minimum expected completion time of tasks of the new problem, we can determine exactly if the answer to the given instance of FIRMSSU is positive.

Take an instance of FIRMSSU with $n = 3k$ nonempty subsets $S_1, \ldots, S_{3k}$ on $\{1, \ldots, 3k\}$. We construct an instance of MW problem. The tasks are arranged in a DAG with three levels (see Figure 2 for an example) and in five different sets. For each $i \in \{1, \ldots, n\}$ we have a group of n nodes in A_1, labeled with i. For each set S_j, $j = 1, \ldots, n$, we have a node in B_2, labeled with S_j. If in the instance of FIRMSSU, $i \in S_j$, we connect all nodes in the i-th group of A_1 to

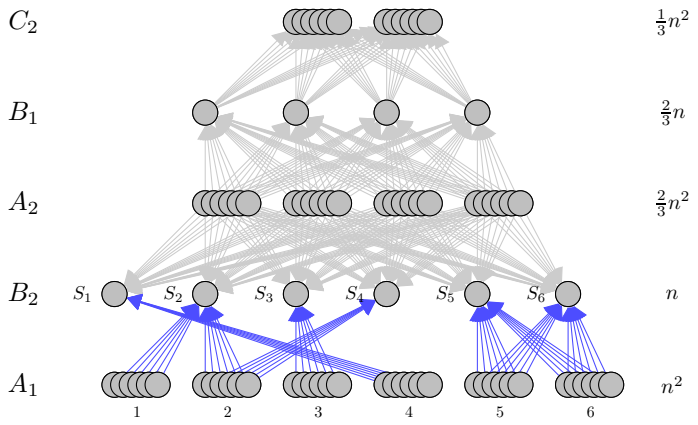


Figure 2: A reduction from FIRMSSU to MW problem with two workers and unconstrained DAG width. Each task in A_1 and B_1 can be executed quickly by the first worker and each task in A_2, B_2 and C_2 can be executed by the second worker quickly.

node S_j. We add three more sets of nodes to the DAG, a set of $2/3n^2$ nodes, A_2, a set of $2/3n$ nodes, B_1, and a set of $1/3n^2$ nodes, C_2. We add an edge from each node in A_2 to each node in B_1 and B_2. We also add an edge from each node in B_1 to each node in C_2.

We use the above described DAG to express the dependency relations between tasks in our new problem. We set the workers parameters such that the first worker can execute tasks in A_1 and B_1 with mean time equal to 1 and the second worker can execute tasks in A_2, B_2 and C_2 with mean time equal to 1. Each worker can execute tasks in the other sets with a very small parameter λ_ϵ (which we will later give a bound for), that is, the mean time for executing those tasks is very large.

Assume the answer to the instance of FIRMSSU is positive. Therefore there are $2k$ subsets from $S_1, \ldots, S_{3k}$ with union U such that $|U|$ is at most $2k$. If the cardinality of U is less than $2k$, add some additional elements from $\{1, \ldots, 3k\}$ so as to increase the cardinality of U to $2k$. Consider the following schedule, which consists of four rounds. In the first round, the second worker starts executing tasks in A_2 one after the other and the first worker selects those $2/3n^2$ tasks of A_1 from groups corresponding to U. After both of them finished their tasks, in the second round, the first worker executes tasks in B_1 which are eligible now, and the second worker executes those tasks in B_2 that are eligible. Obviously the number of eligible tasks in B_2 is at least $2/3n^2$. When both of the workers finished their tasks, in the third round, the first worker starts executing the remaining tasks from A_1 and the second worker starts executing the tasks in C_2. In the fourth round, after both finished their tasks, the remaining tasks in B_2 which are at most $1/3n$ are executed by both of them.

Because in the first three round, the expected time to execute a task by each worker is 1, and the number of tasks for each worker is $n^2 + 2/3n$, the expected time for the first three rounds is $n^2 + 2/3n$ in this schedule. The expected time

to execute tasks in the fourth round is

$$1/3n \times \frac{1}{1+\lambda_\epsilon} = 1/3n - \frac{n.\lambda_\epsilon}{3(1+\lambda_\epsilon)}.$$

Because the optimum expected time to execute all tasks is not greater than the expected required time for any given schedule, the optimum time will be at most $n^2 + n - \frac{n.\lambda_\epsilon}{3(1+\lambda_\epsilon)}$ which is less than $n^2 + n$

Now consider that the instance of FIRMSSU has a negative answer. Pick any schedule to execute the tasks. The union of any $2k$ subsets has cardinality at least $2k + 1$, Therefore we need to execute at least $2/3n^2 + n$ tasks from A_1, to make $2/3n$ tasks of B_2 eligible. We claim after $2/3n^2 + 2/3n$ tasks from A_1 and B_1 are finished, there are at least $1/3n^2 + 1/3n + 1$ tasks that have not been executed yet. This is because i) after we have executed $2/3n^2 + 2/3n$ tasks from A_1, there are at most $2/3n - 1$ eligible tasks in B_2 and ii) all tasks of C_2 depend on exactly $2/3n^2 + 2/3n$ tasks that must be executed before. So there are $1/3n + 1$ tasks in B_2 and $1/3n^2$ tasks in C_2 not executed yet. Based on this reasoning, the expected execution time for tasks will be at least

$$(2/3n^2 + 2/3n + 1/3n^2 + 1/3n + 1)\frac{1}{1+\lambda_\epsilon} = (n^2 + n + 1)\frac{1}{1+\lambda_\epsilon}$$

or

$$n^2 + n + 1 - \frac{(n^2 + n + 1)\lambda_\epsilon}{1+\lambda_\epsilon}.$$

If we choose λ_ϵ small enough such that $\epsilon = (n^2 + n + 1)\lambda_\epsilon < 1$, the optimum expected execution time for tasks will be at least $n^2 + n + 1 - \epsilon$ which is greater than $n^2 + n$.

Therefore by comparing the minimum expected completion time of the tasks by $n^2 + n$, we can determine whether the answer to the instance of FIRMSSU is positive or negative. Since FIRMSSU is NP-complete, computing the minimum expected completion time of the tasks in this case is NP-hard. □

5 Conclusion

In this paper we introduced a new scheduling problem based on ROPAS [14] problem. The goal of ROPAS is to schedule some related tasks by some workers so that the total expected time to execute the tasks is minimum. The relations between the tasks is modeled by a DAG.

This problem, called MW, differs from ROPAS in how workers execute tasks. In ROPAS each worker can execute a task successfully by a predetermined probability in a unit time, however in MW each worker can execute a task in a time determined by an exponential distribution.

We proved that the problem is NP-hard in the general form, unless the width of the dependency graph between tasks is limited by a constant. We also gave a polynomial time algorithm for the problem in the restricted form. The running time of the algorithm is $O(ndm^d \log(nd))$, where n is the number of workers, m is the number of tasks and d is the DAG width. If d is constant, obviously the algorithm has polynomial running time.

References

[1] P. Brucker. *Scheduling Algorithms*. Springer, 5th edition, March 2007.

[2] J. Bruno, P. Downey, and G. N. Frederickson. Sequencing tasks with exponential service times to minimize the expected flow time or makespan. *J. ACM*, 28:100–113, January 1981.

[3] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[4] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, April 2008.

[5] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–168, 1950.

[6] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 4:287–326, 1979.

[7] P. L. Hammer and S. Rudeanu. *Boolean Methods in Operations Research and Related Areas*. Springer-Verlag, Berlin, Heidelberg, New York, 1968.

[8] P. Hansen, M. V. P. de Aragão, and C. C. Ribeiro. Hyperbolic 0-1 programming and query optimization in information retrieval. *Math. Program.*, 52(2):255–263, 1991.

[9] W. Herroelen and R. Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research*, 165(2):289–306, 2005.

[10] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, November-December 1961.

[11] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of Scheduling under Precedence Constraints. *Operations Research*, 26(1):22–35, 1978.

[12] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problem. *Annals of Discrete Mathematics 1 (North-Holland, Amsterdam)*, pages 343–362, 1997.

[13] G. Lin and R. Rajaraman. Approximation algorithms for multiprocessor scheduling under uncertainty. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 25–34, New York, NY, USA, 2007.

[14] G. Malewicz. Parallel scheduling of complex dags under uncertainty. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 66–75, New York, NY, USA, 2005.

[15] R. H. Möhring. Computationally tractable classes of ordered sets. In *Algorithms and Order*, pages 105–193, 1989.

[16] M. Pinedo and G. Weiss. Scheduling of stochastic tasks on two parallel processors. *Naval Research Logistics Quarterly*, 26(3):527–535, 1979.

[17] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer Publishing Company, Incorporated, 3rd edition, 2008.

[18] O. A. Prokopyev, H.-X. Huang, and P. M. Pardalos. On complexity of unconstrained hyperbolic 0-1 programming problems. *Operations Research Letters*, 33(3):312–318, 2005.

[19] O. A. Prokopyev, C. N. Meneses, C. A. S. Oliveira, and P. M. Pardalos. On multiple-ratio hyperbolic 0-1 programming problems. *Pacific Journal of Optimization*, 1(2):327–345, 2005.

[20] P. Robillard. (0, 1) hyperbolic programming problems. *Naval Research Logistics Quarterly*, 18(1):47–57, 1971.

[21] J. D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10:384–393, June 1975.

[22] L. Van Der Heyden. Scheduling jobs with exponential processing and arrival times on identical processors so as to minimize the expected makespan. *Mathematics of Operations Research*, 6(2):305–312, 1981.

[23] B. Veltman. *Multiprocessor scheduling with communication delays*. PhD thesis, Eindhoven University of Technology, 1993.

[24] R. R. Weber. Scheduling jobs with stochastic processing requirements on parallel machines to minimize makespan or flowtime. *Journal of Applied Probability*, 19:167–182, 1982.

[25] G. Weiss and M. Pinedo. Scheduling tasks with exponential service times on nonidentical processors to minimize various cost functions. *Journal of Applied Probability*, 17:187–202, 1980.