Scheduling Independent and Identically Distributed Tasks with In-Tree Constraints on Three Machines in Parallel

Moritz G. Maaß

München, 2001

Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen

Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Diplomarbeit

Thema:	Scheduling Independent and Identically Distributed Tasks with In-Tree Constraints on Three Machines in Parallel
Bearbeiter:	Moritz G. Maaß
Aufgabensteller:	Prof. Dr. Ernst W. Mayr
Betreuer:	Prof. Dr. Ernst W. Mayr
Abgabedatum:	15. Oktober 2001

Ich versichere, dass ich diese Diplomarbeit
selbständig verfasst und nur die angegebenen
Quellen und Hilfsmittel verwendet habe.

27. August 2001

Contents

List of Figures

Chapter 1

Introduction

Scheduling – the problem of allocating resources to different tasks – is a classical problem of computer science. As this is one of the fundamental problems of organizing work in a world based on the division of labor, scheduling problems have their roots in the domain of operations research, an economical, mathematical and military discipline. With the availability of computers, a lot more problems have become tractable in reasonable time (compared to the time and resources saved by an optimized work schedule).

A scheduling problem generally consists of a set of resources (such as processors, workers, machines), a set of tasks, usually some constraints (due dates, machine constraints, precedence relationships, etc.), and an optimization objective (usually a time based term). The information is mostly given as the number and the size of the tasks (the amount of processing needed per task), the number of processors (machines) and their capabilities (the processing speed), some external constraints in the form of arrival and due dates or precedence constraints (an order on the processing sequence), and the objective to optimize. The first scheduling problems studied by computer scientists were problems where most information was deterministic and known a priori. Although it turned out soon that a lot of these problems were computationally intractable (NP-hard problems), optimal algorithms are known for quite a lot of relevant problems. Usually, even if a problem is NP-hard, some versions of it can be solved. A lot of approximation algorithms with good time bounds are known (see [Pin95, Bru95] for general overviews).

Precedence constraints in their most general form can be modeled by a directed acyclic graph. Unfortunately, even the simple problem of scheduling identical tasks on multiple processors in parallel with general precedence constraints to minimize the makespan (the total processing time) is NP-hard [Ull75, Ull76]. When the problem is limited to precedence constraints that form an in-tree or an out-tree, then the problem becomes solvable by the highest level first (HLF, sometimes also CP – critical path) rule [Hu61]. If the limited problem is generalized by allowing tasks of lengths 1 and 2, then the problem is again NP-hard [Ull75].

The assumption that all information for a scheduling problem is known a priori is often unrealistic. Things may happen coincidentally or there may be no chance to predict the behavior of a problem component. To take this into account, a scheduling problem is modeled with random variables. An early approach was made by Chandy and Reynolds [CR75], who modeled the task processing time as independent identically, exponentially distributed random variables and proved that the HLF strategy minimizes the expected makespan for two machines and precedence constraints that form an in-tree. They also gave a counter example for the three machines case. Under the same problem with two machines Bruno [Bru85] showed that the HLF strategy maximizes the probability that all tasks are finished by a given time (stochastically minimizes the total expected processing time) and that the HLF strategy also maximizes the probability that the sum of the finishing times is below a given value (only the preemptive case is considered). Pinedo and Weiss [PW85] were able to show that the HLF strategy minimizes the expected makespan if the processing times of all tasks at level l are independent identically, exponentially distributed (tasks at different levels may have different distributions). They consider the preemptive and the non-preemptive case. A modified HLF strategy even minimizes the expected makespan for more general (but discrete) distributions. Frostig

[Fro88] considers increasing hazard rate (IHR) distributions and proves that a modified HLF strategy (ties are broken by selecting tasks with lowest hazard rate, the strategy hence called HLF:LHR) stochastically minimizes the total expected processing time. Going back to the original problem definition by Chandy and Reynolds with a fixed number of machines, in-tree constraints and independent identically, exponentially distributed task running times, an optimal strategy for three or more machines is unknown. Papadimitriou and Tsitsiklis [PT87] have shown that the HLF strategy is asymptotically optimal as the number of tasks tends to infinity.

We take Chandy and Reynolds definition as a basis here and scrutinize the three machines version. Therefore, we have developed an algorithm with exponential running time to optimally solve the three machines version of the problem. With this algorithm we were able to generate a large number of small in-trees (with less than 19 nodes) and compare the optimal strategy with other strategies. We prove that not only the HLF strategy, but all static list scheduling strategies (a static list scheduling orders all tasks at the beginning and always schedules the first available task from the list) cannot be optimal. Furthermore, if one considers intermediate states in the scheduling processes, even all semi-static list scheduling strategies (the task list is generated per tree) cannot be optimal. As a result, an optimal schedule must take already scheduled leaves from a prior step into account.

We will show that the structure of the problem depends on subtrees of the precedence constraints in-tree with only two leaves. Their influence on the problem lies in the fact that in the end-phase of the scheduling process such a subtree with only two leaves is reached and one machine must be left idle. The expected makespan depends on how early and which such subtree is reached. Based on this structure we have developed some algorithms and compared their performance with each other and with the HLF strategy. With only two machines, the problem structure can be broken down analogously resulting in subtrees with one leaf only, which gives us another hint on the optimality of the HLF strategy for two machines. An optimal algorithm for the three machines version of the problem could not be found.

This thesis is structured as follows. Chapter 2 establishes the notation used throughout the work. The experienced reader might as well skim through the chapter and jump back later if needed. Chapter 3 gives some background on scheduling and scheduling strategies. The widely used three parts classification scheme is introduced and the problem at hand is classified. In chapter 4 the use of the exponential distribution in this particular scheduling problem is introduced and the main properties are extracted. A sidelong glance at the geometric distribution is taken. The in-tree precedence constraints are looked at more closely in chapter 5. We give an introductory example, establish a frame for scheduling tasks with independent identically, exponentially distributed running times and in-tree constraints and formalize the optimization objective. In chapter 6 we develop a dynamic programming algorithm with exponential running time to calculate the optimal solutions. We define some evaluation criteria for different strategies and apply these to the HLF and other strategies in chapter 7. The structure of the problem is broken down into two-leaves-subtrees in chapter 8. The results are applied to develop algorithms based on combinatorial approximations in chapter 9 and based on a node-wise reflection in chapter 10. We try to draw a conclusion and give an outlook in chapter 11.

I want to thank Prof. Dr. Ernst W. Mayr for suggesting the problem to me and for helpful remarks, hints, corrections, and encouragement. I also want to express gratitude to Alexander Offtermatt-Souza for some fruitful discussions. I am in debt to Hanjo Täubig and my mother Maren Stiller-Maaß for reading and correcting my final work. Last but not least I want to thank Ina Jahnel for infinite patience and support.

Chapter 2

Basic Definitions

This work will deal a lot with graphs, in particular with trees. We therefore start with some general notation for later use. The experienced reader can just skim through this section and jump back when needed.

Let $G = (V, E)$ be a graph, where $V \neq \emptyset$ is a non-empty set of vertices and $E \subseteq \{\{u, v\} | u \in V \wedge v \in V\}$ is a set of unordered pairs of nodes, the edges. If $E \subseteq V \times V$, the graph is said to be directed. Each edge in a directed graph is an ordered pair of vertices. If $v \in V$ is a vertex of G we will also use the shorthand $v \in G$. In the following we only look at directed graphs.

Let $e = (u, v) \in E$ be an edge. Then $start(e) = u$ and $end(e) = v$. For a node $v \in V$, we will use $in(v) = \{(u, v)|(u, v) \in E\}$ and $out(v) = \{(v, u)|(v, u) \in E\}$. The in-degree of node v is $d_{in}(v) = |in(v)|$, the out-degree is $d_{out}(v) = |out(v)|$.

A path p from node u to node v is an n-tuple of edges $p = path(u, v) = (e_1, e_2, \ldots, e_n)$ where $e_i \in E, i = 1, \ldots, n$ and $start(e_1) = u \wedge end(e_n) = v \wedge \forall i = 2..n : end(e_{i-1}) = start(e_i)$. A path can also be denoted by its nodes. In this case a path is an n-tuple of nodes $path(u, v) = (u, w_1, w_2, \ldots, w_{n-2}, v)$. The length of a path is the number of nodes it contains.

Let $paths_G(u, v)$ be the set of all paths from u to v in G.

A cycle is a path $(u, w_1, w_2, \ldots, w_{n-2}, u)$, where $\forall i = 1..n - 2 : w_i \neq u$ and the length of the path is at least 2.

A graph that contains no cycles is called a **directed acyclic graph (DAG)**.

A **tree** T is a graph with n nodes, $n - 1$ edges and with no cycles.

A **rooted tree** B is a pair (r, T) where T is a tree and $r \in T$ is a node of T. A rooted tree $B = (r, T)$ is called an out-tree, if and only if $\forall v \in T : v = r \vee \exists path(r, v)$. A rooted tree $B = (r, T)$ is called an in-tree, if and only if $\forall v \in T : v = r \vee \exists path(v, r)$.

Rooted trees with all maximal paths either starting or ending at the root are also called **distinct oriented trees with labeled vertices** (see [Knu97], section 2.3.4.4) or **rooted and labeled trees**.

In the following all trees will be in-trees and we will therefore not need to define the root of the tree explicitly. For an in-tree B the root is the only node r with out-degree $d_{out}(r) = 0$. Let $root(B) = r$.

A node $p \in B$ is an ancestor of $v \in B$ in in-tree B, if there exists a path $path(v, p)$ (p is "closer to root" than v).

In in-trees there is no need to specify a path by a tuple since there can only be one path between two nodes. We will therefore let $path(u, v)$ be the set of nodes rather than a tuple. If there exists a path from a to b, then let $dist(a, b) = |path(a, b)| - 1$ (which is the number of edges between a and b).

For the following let B be an in-tree and r be its root.

If a is a node in B, then let $height(a) = dist(a, r)$ (the number of node levels).

Let $height(B) = \max_{a \in B}\{height(a)\} + 1$.

Let a and b be nodes in B. Let p be a node such that p is a common ancestor of a and b ($\exists path(a,p) \wedge \exists path(b,p)$). p is said to be the lowest common ancestor (lca), if for all v that are common ancestors of a and b, $height(p) = max_v\{height(v)\}$. (Note: The max is introduced here because we let trees grow in an unusual direction - upwards. The lowest common ancestor is hence really a highest common ancestor, but lca is the name commonly used.). We define $lca(a,b) := p$. There is always a lowest common ancestor for any nodes a, b, because there is a path to the root from any node in the tree.

Let $leaves(B)$ be the set of leaves of tree B.

Let a be a node in B, then let $B|_a$ be the subtree of B, whose root is a.

Two trees $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ are isomorphic, if there exists a bijective function $f : V_1 \to V_2$, s.t. $\forall (v,u) \in E_1 : (f(v), f(u)) \in E_2$ and $\forall (v,u) \in E_2 : (f^{-1}(v), f^{-1}(u)) \in E_1$.

If we build equivalence classes of trees with respect to isomorphism, we can represent each class by a rooted, unordered tree that only describes the tree structure.

The number of different trees with n nodes differs immensely based on what trees are considered the same:

Lemma 2.1 (Number of Rooted and Labeled Trees). *The number of rooted and labeled trees is n^{n-1}.*

Proof. From Cayley's Formula we know that the number of labeled trees is n^{n-2}. There are n nodes, choosing a distinct root yields n^{n-1}. More detailed proof can be found in [Knu97], section 2.3.4.4. □

The number of rooted, unordered trees is smaller (we drop the labeling and a lot of trees are now isomorphic). The sequence a_n of the number of rooted, unordered trees with n nodes starts with $1, 1, 2, 4, 9, 40, 48, \ldots$ and it can also be found as sequence number $A000081$ in [Onl]. There is no closed formula for this number, but a_n grows asymptotically:

Lemma 2.2 (Asymptotic Growth of the Number of Rooted, Unordered Trees with n Nodes). *The number of rooted, unordered trees with n nodes a_n is asymptotically*

$$a_n = \frac{1}{\alpha^{n-1} n}\sqrt{\beta/2\pi n} + \mathcal{O}\left(\frac{1}{\sqrt{n^5}\alpha^n}\right)$$

Where $1/\alpha \approx 2.955765285652$ *and* $\sqrt{\beta/2\pi} \approx 0.439924012571$

Proof. See [Knu97], section 2.3.4.4. □

As a rule of thumb, the above formula tells us that a_n grows nearly as fast as 3^n.

Chapter 3

Scheduling

Michael Pinedo defines scheduling in [Pin95] as

> Scheduling concerns the allocation of limited resources to tasks over time. It is a decision-making process that has as a goal the optimization of one or more objectives.

Following [Pin95] we will give a brief introduction into the scheduling field.

3.1 A Classification Scheme for Scheduling Problems

A scheduling problem can be described by a triple $(\alpha|\beta|\gamma)$. The first part (α) describes the machines (also resources), the second part (β) describes the tasks and the third part (γ) describes the optimization objective. This notation was introduced by Lawler et al. [LLR82], the presented version is a slightly adapted one from Pinedo [Pin95].

The following notations for the description of the resources are known:

1	A single machine is available for processing the given task(s).
Pm	There are m identical machines available for processing the given tasks in parallel.
Qm	There are m machines with different processing speeds v_j available for working on the given tasks in parallel. (Machine j can process task i in p_i/v_j time).
Rm	There are m machines with different processing capabilities available for working on the given tasks in parallel. The machine speed depends also on the task, so that machine j can process task i at speed v_{ij}.
Fm	In the Flow Shop environment with m machines in series each task has to be processed on each machine, starting with the first and ending with the last machine.
FFs	In the Flexible Flow Shop environment there are s stages and a number of machines in parallel at each stage. Each task has to be processed in every stage, starting with the first and ending with the last one. No task may overtake another one.
Om	The Open Shop environment relaxes constraints from Fm further such that the task may be routed differently (determined by the scheduler). Some tasks may have processing time zero on some machines.
Jm	In the Job Shop Environment each task has to follow a specific route through the m machines.

The description of the tasks usually includes further constraints, some of which may be:

r_j Tasks have release times r_j.

s_{jk} Tasks have setup times depending on the sequence they are assigned to a machine (s_{jk} is the time needed to set up the machines between jobs j and k).

$prmp$ A processor processing a task may be preempted and be reassigned to another task by the scheduler. Usually one assumes that a processor once assigned to a task will only be available after that task has finished.

$prec$ Precedence constraints define precedence between different tasks, requiring that some tasks must be finished before others can be started. The precedence constraints can be described by a DAG $G = (V, E)$, where V are the tasks and $(u, v) \in E$ is an edge from u to v, if u must be finished before v can be started.

$chains$ This is a special case of precedence constraints ($prec$), where the maximal in-degree and the maximal out-degree of any node in the precedence graph is smaller or equals to one ($\forall v \in V : d_{in}(v) \leq 1 \wedge d_{out}(v) \leq 1$).

$outtree$ This is a special case of precedence constraints ($prec$), where the maximal in-degree of any node in the precedence graph is smaller or equals to one ($\forall v \in V : d_{in}(v) \leq 1$). The task-nodes form an out-tree or an out-forest.

$intree$ This is a special case of precedence constraints ($prec$), where the maximal out-degree of any node in the precedence graph is smaller or equals to one ($\forall v \in V : d_{out}(v) \leq 1$). The task-nodes form an in-tree or an in-forest.

$brkdwn$ Machines can break down and not be available during some time.

M_j In the environment Pm this means that not all machines can process all tasks, only machines in set M_j can process task j

no-$delay$ No machine may be idle if there is a free task that can be processed. Usually all problems are considered no-delay (that is only greedy strategies are considered).

We will usually assume that there are n tasks to be processed. The optimization criterion makes use of the following variables:

p_{ij} is the time that task j has been processed on machine i. The index i can be omitted if the machines do not differ or if only one machine is available. p_j then denotes the total time that task j was processed on any machine.

d_j is the due date. That is the time by which a task should be finished.

w_j is a weight that defines the priority given to a task in relation to other tasks.

C_j is the completion time of task j (the time that task j has received all the processing that it needed).

U_j denotes whether a task finished on time. It is 1 if $d_j >= C_j$ and 0 otherwise.

L_j is the lateness of a task. It is defined as $L_j = C_j - d_j$.

X_{ij} is the random processing time of task j on machine i. This is a pendant to p_{ij} for a stochastic scheduling environment.

$\frac{1}{\lambda_{ij}}$ is the expected value of X_{ij} ($\mathbb{E}(X_{ij})$ – often the exponential distribution is used, hence the fraction here).

R_j is the random release time of task j (similar to r_j).

D_j is the random due date of task j (similar to d_j).

The following are common optimization criteria (most of which are only included for completeness):

C_{max}	The **makespan** is the time that the last job has finished: $$C_{max} = max\{C_1, \dots, C_n\}$$
$\mathbb{E}(C_{max})$	The **expected makespan** is the expected time that the last job has finished: $$\mathbb{E}(C_{max}) = \mathbb{E}(max\{C_1, \dots, C_n\})$$
L_{max}	The **maximum lateness** is defined as $$L_{max} = max\{L_1, \dots, L_n\}$$
$\sum w_j C_j$	The **total weighted completion time** is defined as $$\sum_{j=0}^{n} w_j C_j$$
$\sum w_j(1 - e^{-rC_j})$	The **discounted total weighted completion time** is defined as $$\sum_{j=0}^{n} w_j C(1 - e^{-rC_j})$$
$\sum w_j T_j$	The **total weighted tardiness** is defined as $$\sum_{j=0}^{n} w_j T_j$$
$\sum w_j U_j$	The **weighted number of tardy jobs** is defined as $$\sum_{j=0}^{n} w_j U_j$$

With this notation it is possible to classify the problem we are investigating in this work. We have three identical machines available and n tasks that are **independent identically, exponentially distributed** and have **in-tree constraints**:

$$(P3|intree|\mathbb{E}(C_{max}))$$

We will also take a short look at the preemptive version:

$$(P3|intree, prmp|\mathbb{E}(C_{max}))$$

3.2 Scheduling Strategies

The key to most problems is to find a scheduling strategy that optimizes the chosen criterion. A scheduling strategy (or policy) determines which tasks are processed at what times. The strategy is feasible, if no constraints are violated. In the following we will only deal with feasible strategies.

Let T be the set of tasks (in our case an in-tree), let $n = |T|$ be the number of task and let m be the number of machines. At time t let α_S^t be the tasks that are scheduled by the strategy S. Obviously $\forall t : \alpha_S^t \subseteq T \wedge |\alpha_S^t| \leq m$. The value of the chosen optimization criterion should depend on nothing else but S.

A classification that is not shown in the above scheme is the difference between stochastic and deterministic scheduling. The classical problems in scheduling were all deterministic. In this work we are dealing with a stochastic scheduling problem. Different classes of scheduling strategies are known for the case of stochastic scheduling problems. The concrete parameters (such as processing time) of a stochastic scheduling problem are random variables. Hence there exist multiple concrete instances of the same stochastic scheduling problem that can differ and reach states that are not reached by another instance (e.g. with multiple machines and random processing times jobs may finish in different order). Therefore certain information (e.g. task i finishes before task j) will only become available after some time. Scheduling strategies can thus be classified by whether they (can) take this information into account or not.

Definition 3.1 (Static List Policy). *A **static list scheduling policy** is a policy where all tasks are ordered at the beginning. At any point in time the highest available tasks are scheduled. The decisions made at the beginning are binding throughout the total execution.*

Definition 3.2 (Dynamic Policy). *A **dynamic policy** allows the scheduler to take all prior information available into account when a new task is to be scheduled. The decisions made are only binding for a single step.*

Chapter 4

Exponentially Distributed Task Processing Times

4.1 Motivation

In deterministic scheduling models all variables needed for optimization are known beforehand. For a real world problem this is usually not the case. Stochastic scheduling models are closer to reality because they do not assume the input values to be known. A common issue in a lot of real-world planning problems is that the durations of tasks are not known a priori.

Take the development of a software as an example: A software consists of a number of modules that can be developed concurrently. The development tasks and other tasks, such as testing, depend on one another. To generate an optimal project plan one would need to know the time that the development of the components needs. Although a time can be guessed from prior experience the real time is unknown.

To be able to achieve any results on such problems, we usually assume that the expected duration is known (e.g. it can be guessed from prior projects). The expected values are then used to generate an optimal plan with respect to the expected total completion time.

Often it does not suffice to know the expected values, but one also needs to know something about the character of the underlying distribution. For continuous random distributions the character can be described by a distribution and a density function. The knowledge of the density functions allows to deduce more complex results.

One well-liked distribution is the exponential distribution because of its special properties. The exponential distribution can be easily combined with the like and – even more important – the exponential distribution is memory-less (see Lemma 4.3).

4.2 Continuous Distributions

A random variable X is continuously distributed, if it can take any value in $\mathbb{R}$ (or a continuous interval of $\mathbb{R}$). A continuous distribution can best be described by a density function f. Integrating over f yields the probability that the value of X falls in a certain range (note, that the probability of a continuous function taking a single value is zero). The probability of X taking a value below t is denoted by the distribution function F:

$$P[X \leq t] = F(t) = \int_{-\infty}^{t} f(x)dx$$

There are many well-known distributions, having different beneficial properties (such as well describing natural processes or being easy to handle).

For distributions describing random time periods the probability that the time interval is negative is zero ($\forall t \leq 0 : f(t) = 0, F(t) = \int_0^t f(x)dx$). A time distribution can be used to describe the time until an event occurs.

A time distribution is **memory-less**, if it behaves the same after some time has passed, i.e. under such distribution the probability that an event occurs in the next two minutes is the same as the probability that an event occurs in the next two minutes after two minutes have elapsed. Formally this is true for continues distributions, if

$$f(t) = \frac{f(t+t_0)}{1-F(t_0)} = \frac{f(t+t_0)}{\int_{t_0}^{\infty} f(x)dx}.$$

Another definition of the memory-less property often used is

$$P[X > t] = P[X > t + t_0 | X > t_0].$$

Lemma 4.1 (Expected Value of a Memory-Less Distribution). *If X is a randomly time distributed variable with density function f and X is memory-less, then the expected value $\mathbb{E}(X|_{t_0})$ of X after time zero is $\mathbb{E}(X) + t_0$.*

Proof. The expected value of X is

$$\mathbb{E}(X) = \int_0^{\infty} t \cdot f(t)dt$$

After some time t_0 has elapsed the expected value of $X|_{t_0}$ is

$$\begin{aligned}
\mathbb{E}(X|_{t_0}) = & \\
& \frac{\int_{t_0}^{\infty} t \cdot f(t)dt}{\int_{t_0}^{\infty} f(t)dt} = \\
& \int_0^{\infty} (t+t_0) \cdot \frac{f(t+t_0)}{\int_{t_0}^{\infty} f(t)dt} dt = \\
& \int_0^{\infty} (t+t_0) \cdot f(t)dt = \\
& \int_0^{\infty} t \cdot f(t)dt + t_0 \int_0^{\infty} f(t)dt = \\
& \mathbb{E}(X) + t_0
\end{aligned}$$

□

Memory-Less time distributions are very helpful in scheduling problems because they reduce the search space to discrete time points:

Lemma 4.2 (Discrete Preemption Points). *If decisions are based only on random variables with memory-less time distributions (and possible non-random variables), then there are only a discrete number of points where a preemption may occur.*

Proof. Let t_i and t_j be two times with $t_i < t_j$ and let no event occur between t_i and t_j. All information available at point t_i is the history of events and the behavior of the distributions described by their density functions. Because no events occur, the history stays the same for point t_j. Since all distributions are memory-less, the behavior of the density functions does not change. Therefore the identical information is available at time t_i and t_j and thus the scheduler would schedule the same task. A reevaluation at time t_j would not lead to a change. As a result no preemption will occur. □

Another nice property of memory-less time distributions is that a lot of times a problem reduction can be made. It makes no difference what happened before a certain set of tasks with some scheduled tasks is reached. The work and time already invested in a task does not change the expected remaining time needed.

While the above properties often allow the achievement of results, they also hint at the problem of applying the results in the real world. There surely exist a lot of tasks that are not memory-less (e.g., the building of a wall or the digging of a hole) for which the results cannot be applied. On the other hand one can argue that there exist tasks for which the memory-less property makes sense. Examples for such tasks are (under certain assumptions) the catching of a fly, reaching a certain person over the phone, and others. Some tasks also look memory-less to an extend that might make it reasonable to model them as such. The solving of a riddle, or other tasks that include searching for a solution to a problem without knowing an algorithm are examples of such tasks.

The main reason for postulating the memory-less property is certainly the nice mathematical properties of such distributions.

4.3 The Exponential Distribution

The exponential distribution is one of the most commonly used distributions. The exponential distribution is the only continues memory-less time distribution (see Theorem 2.25 in [SS01]).

Let X be an exponentially distributed random variable with parameter λ.

If the finishing time of a task is exponentially distributed with variable X, then $\frac{1}{\lambda}$ is the expected finishing time.

Let f be the density of the exponential distribution:

$$f = x \rightarrow \begin{cases} \lambda e^{-\lambda x} & \text{if } x >= 0, \\ 0 & \text{otherwise.} \end{cases}$$

Let F be the exponential distribution function:

$$F = x \rightarrow \begin{cases} 1 - e^{-\lambda x} & \text{if } x >= 0, \\ 0 & \text{otherwise.} \end{cases}$$

The relation between F and f is

$$\begin{aligned} F(x) &= \int_{-\infty}^{x} f(t)dt \\ &= \begin{cases} \int_{-\infty}^{0} 0 \cdot dt + \int_{0}^{x} \lambda \cdot e^{-\lambda t} dt = & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \\ &= \begin{cases} -e^{-\lambda t}\big|_0^x & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \\ &= \begin{cases} -e^{-\lambda x} - (-1) & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \\ &= \begin{cases} 1 - e^{-\lambda x} & \text{if } x \geq 0, \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

The expected processing time is $\frac{1}{\lambda}$:

$$\begin{array}{rcl}
\mathbb{E}(X) & = & \int_{-\infty}^{\infty} t \cdot f(t)dt \\
& = & \int_{0}^{\infty} t \cdot \lambda \cdot e^{-\lambda t}dt \\
& = & (-1)\int_{0}^{\infty} t \cdot (-\lambda) \cdot e^{-\lambda t}dt \\
& = & -t \cdot e^{-\lambda t}\big|_0^\infty - \int_{0}^{\infty} (-1) \cdot e^{-\lambda t}dt \\
& = & -t \cdot e^{-\lambda t}\big|_0^\infty - \frac{1}{\lambda} \cdot e^{-\lambda t}\big|_0^\infty \\
& = & 0 - 0 - \frac{1}{\lambda} \cdot (0-1) \\
& = & \frac{1}{\lambda}
\end{array}$$

Among others the memory-less property of the exponential distribution makes the distribution well-liked by scheduling researchers.

Lemma 4.3 (Memory-less Property of the Exponential Distribution). *The exponential distribution is memory-less.*

Proof. We will show

$$f(t) = \frac{f(t+t_0)}{\int_{t_0}^{\infty} f(x)dx}$$

for the exponential distribution.

We first calculate the probability of an exponentially distributed X taking a value larger than t_0:

$$\int_{t_0}^{\infty} f(x)dx = \int_{t_0}^{\infty} \lambda e^{-\lambda x}dx = (-e^{-\lambda x})\big|_{t_0}^{\infty} = (-e^{-\lambda\infty}) - (-e^{-\lambda t_0}) = e^{-\lambda t_0}$$

With that we can proceed to show that

$$\frac{f(t+t_0)}{\int_{t_0}^{\infty} f(x)dx} = \frac{\lambda e^{-\lambda(t+t_0)}}{e^{-\lambda t_0}} = \lambda e^{-\lambda(t+t_0-t_0)} = \lambda e^{-\lambda t} = f(t)$$

□

A proof that every memory-less time distribution must be exponentially distributed is given in [SS01] (Theorem 2.25).

When a number of tasks with independent identically, exponentially distributed processing times is scheduled on a number of machines the outcome is random.

Lemma 4.4 (Minimum of two exponentially distributed Variables). *Let X_1 be an exponentially distributed random variable with expected value $\frac{1}{\lambda_1}$ and let X_2 be an exponentially distributed random variable with expected value $\frac{1}{\lambda_2}$. Let $Y = min(X_1, X_2)$. Y is exponentially distributed with expected value $\frac{1}{\lambda_1+\lambda_2}$.*

Proof. Let $f_1 = \lambda_1 \cdot e^{-\lambda_1 \cdot t}$ be the density function for X_1. Let $f_2 = \lambda_2 \cdot e^{-\lambda_2 \cdot t}$ be the density function for X_2.

The density function f_{min} of Y is either the density of X_1, if X_1 is smaller than or equals X_2 or the density of X_2 otherwise:

$$f_{min}(x) = f_1(x) \cdot P[X_1 \leq X_2] + f_2(x) \cdot P[X_2 \leq X_1] = f_1(x) \cdot P[x \leq X_2] + f_2(x) \cdot P[x \leq X_1] \quad (4.1)$$

The value of $P[x \leq X_2]$ (and $P[x \leq X_1]$ analogous) can be calculated using f_2:

$$P[x \leq X_2] = \int_x^\infty f_2(t)dt = \int_x^\infty \lambda_2 \cdot e^{-\lambda_2 \cdot t} dt = (-1) \cdot e^{-\lambda_2 t}\Big|_x^\infty = 0 - (-1) \cdot e^{-\lambda_2 x} = e^{-\lambda_2 x} \quad (4.2)$$

Using 4.2 in 4.1 yields:

$$f_{min}(x) = \lambda_1 \cdot e^{-\lambda_1 \cdot x} \cdot e^{-\lambda_2 x} + \lambda_2 \cdot e^{-\lambda_2 \cdot x} \cdot e^{-\lambda_1 x} = (\lambda_1 + \lambda_2) \cdot e^{-(\lambda_1 + \lambda_2) \cdot x} \quad (4.3)$$

Hence Y is exponentially distributed with parameter $\lambda_1 + \lambda_2$. □

Lemma 4.5 (Minimum of n independent identically, exponentially distributed Variables). *When n tasks with independent identically, exponentially distributed processing times are scheduled on n machines, where the expected time of a task to finish is $\frac{1}{\lambda}$, then the expected time of the first task to finish is $\frac{1}{n \cdot \lambda}$.*

Proof. The proof follows directly from Lemma 4.4. Combining the first two of the n tasks yields an exponential distributed task with expected finishing time $\frac{1}{\lambda + \lambda}$. Continuing iteratively leads to the stated result. □

In the following we will look at scheduling with three machines. If three machines are working concurrently at a given in-tree after expected time $\frac{1}{3} \cdot \frac{1}{\lambda}$ the first machine should finish. If the constraints are such that only two of the machines can work at tasks, the first machine is expected to finish after time $\frac{1}{2} \cdot \frac{1}{\lambda}$. If only a single machine can work at tasks, the machine is expected to finish after time $\frac{1}{\lambda}$. Since all these times share a common factor $\frac{1}{\lambda}$, we will assume that $\frac{1}{\lambda} = 1$ for subsequent work. All results can be easily extended to an arbitrary value for λ by multiplying with $\frac{1}{\lambda}$.

The expected value of the maximum of a number of independent identically, exponentially distributed variables follows directly from Lemma 4.5:

Lemma 4.6 (Maximum of n independent identically, exponentially distributed Variables). *When n tasks with independent identically, exponentially distributed processing times are scheduled on n machines, where the expected time of a task to finish is $\frac{1}{\lambda}$, then the expected time of the last task to finish is $\frac{1}{\lambda} \cdot H_n$, where H_n denotes the n-th harmonic number.*

Proof. The proof follows directly from Lemma 4.5 and the memory-less property of the exponential distribution. The expected value of the finishing time of the first tasks is (by Lemma 4.5): $\frac{1}{n \cdot \lambda}$. After time $\frac{1}{n \cdot \lambda}$ the expected value of the finishing time of the first remaining task is $\frac{1}{n \cdot \lambda} + \frac{1}{(n-1) \cdot \lambda}$. Continuing these steps leads to the expected finishing time of the last process to finish, which is

$$\frac{1}{n \cdot \lambda} + \frac{1}{(n-1) \cdot \lambda} + \ldots + \frac{1}{2 \cdot \lambda} + \frac{1}{\lambda} = \frac{1}{\lambda} \sum_{i=1}^{n} \frac{1}{i} = \frac{1}{\lambda} H_n.$$

□

4.4 Other Distributions Besides the Exponential

Before we restrict ourselves for the rest of this work to the exponential distribution, we will take a short look at another distribution which shares an important property of the exponential distribution. The main properties of the exponential distribution used throughout this work are the memory-less property and the expected minimum of n variables. Hence any distribution that has these properties and that is a time distribution can be used. There is no other continuous memory-less distribution, but there is a discrete distribution that is memory-less.

The **geometric distribution** is the only discrete distribution that has the memory-less property. For discrete random variable X with range $\mathbb{N}^+$, the geometric distribution with parameter $0 \leq p \leq 1$ is defined through

$$P[X = t] = p \cdot (1-p)^{t-1},$$

which is the equivalent to the density function. The distribution function's equivalent is

$$P[X \leq t] = \sum_{i=1}^{t} P[X = i] = \sum_{i=0}^{t-1} p \cdot (1-p)^i = p\frac{1-(1-p)^t}{1-(1-p)} = 1-(1-p)^t.$$

The following lemma helps us to determine the expected value of the geometric distribution.

Lemma 4.7.

$$\sum_{i=0}^{\infty} i \cdot p^i = \frac{p}{(1-p)^2}$$

Proof. We know the geometric sum $\sum_{i=0}^{n} p^i = \frac{1-p^{n+1}}{1-p}$, hence $\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}$.
We go the reverse way for the proof and start with the closed formula:

$$\frac{p}{(1-p)^2} = p \cdot \frac{1}{1-p} \cdot \frac{1}{1-p} = p \cdot \Big(\sum_{i=0}^{\infty} p^i\Big) \cdot \Big(\sum_{i=0}^{\infty} p^i\Big) =$$
$$p \cdot \Big(\sum_{i=0}^{\infty}\sum_{j=0}^{i} p^j \cdot p^{i-j}\Big) = p \cdot \Big(\sum_{i=0}^{\infty}(i+1)p^i\Big) = \sum_{i=0}^{\infty}(i+1)p^{i+1} = \sum_{i=0}^{\infty} i \cdot p^i$$

□

The expected value of the geometric distribution is therefore

$$\sum_{t=1}^{\infty} t \cdot p \cdot (1-p)^{t-1} = \frac{p}{1-p}\sum_{t=0}^{\infty} t \cdot (1-p)^t = \frac{p}{1-p} \cdot \frac{1-p}{(1-(1-p))^2} = \frac{1}{p}.$$

The geometric distribution is memory-less because

$$\begin{aligned} P[X = t + t_0 | X > t_0] &= \frac{P[X = t+t_0]}{P[X > t_0]} = \frac{p \cdot (1-p)^{t+t_0-1}}{1 - P[X \leq t_0]} \\ &= \frac{p \cdot (1-p)^{t+t_0-1}}{1-(1-(1-p)^{t_0})} = \frac{p \cdot (1-p)^{t+t_0-1}}{(1-p)^{t_0}} = p \cdot (1-p)^{t-1} = P[X = t]. \end{aligned}$$

Any memory-less discrete distribution with range $\mathbb{N}^+$ must be a geometric distribution:

Lemma 4.8 (The Memory-Less Property of Discrete Distributions). *A discrete distribution X with range $\mathbb{N}^+$ and the memory-less property $P[X > n+m | X > m] = P[X > n]$ is geometrically distributed.*

Proof. Let X be any memory-less discrete distribution with range $\mathbb{N}^+$. It follows that

$$P[X > n+m] = P[X > n+m | X > m] \cdot P[X > m] = P[X > n] \cdot P[X > m].$$

Induction yields

$$\begin{aligned} \forall n \in \mathbb{N}^+ : P[X > n] &= P[X > 1 + (n-1)] = P[X > 1] \cdot P[X > n-1] \\ &= P[X > 1] \cdot (P[X > 1])^{n-1} = (P[X > 1])^n. \end{aligned}$$

Let $P[X > 1] = 1 - P[X \leq 1] = 1 - P[X = 1] = 1 - p$ with $0 \leq p \leq 1$. It follows that $P[X > n] = (1-p)^n$. Therefore,

$$\begin{aligned} P[X = n] = 1 - P[X > n] - P[X \leq n-1] &= 1 - P[X > n] - (1 - P[X > n-1]) \\ &= (1-p)^{n-1} - (1-p)^n = (1-p)^{n-1} \cdot (1 - (1-p)) = p \cdot (1-p)^{n-1}. \end{aligned}$$

As a result, X must be geometrically distributed with parameter p. $\square$

Lemma 4.1 can be easily extended to discrete memory-less distributions.

The minimum $Y = min(X_1, X_2)$ of two independent identically, geometrically distributed random variables X_1 and X_2 with parameter p_1 and p_2 is a random variable that is geometrically distributed with parameter $p_1 + p_2 - p_1p_2$:

$$\begin{aligned} &P[Y = t] = \\ &P[X_1 = t] \cdot P[X_2 = t] + P[X_1 = t] \cdot P[X_2 > t] + P[X_2 = t] \cdot P[X_1 > t] = \\ &p_1(1-p_1)^{t-1} \cdot p_2(1-p_2)^{t-1} + p_1(1-p_1)^{t-1} \cdot (1-p_2)^t + p_2(1-p_2)^{t-1} \cdot (1-p_1)^t = \\ &\big((1-p_1)(1-p_2)\big)^{t-1} \cdot \big(p_1p_2 + p_1(1-p_2) + p_2(1-p_1)\big) = \\ &\big(1 - (p_1 + p_2 - p_1p_2)\big)^{t-1} \cdot \big(p_1p_2 + p_1 - p_1p_2 + p_2 - p_2p_1)\big) = \\ &\big(1 - (p_1 + p_2 - p_1p_2)\big)^{t-1} \cdot \big(p_1 + p_2 - p_1p_2)\big) \end{aligned}$$

The expected value of the minimum of two geometrically distributed variables is hence $\mathbb{E}(Y) = \frac{1}{p_1+p_2-p_1p_2}$. If $p = p_1 = p_2$, then $\mathbb{E}(Y) = \frac{1}{2p-p^2} = \frac{1}{p} \cdot \frac{1}{2-p}$. Therefore the new expected value is not linearly dependent on the original one. As a result, although the geometric distribution is the discrete counterpart to the exponential distribution, it cannot be used in its place for the results of this thesis.

Applying a distribution that does not have the memory-less property seems to result in an even larger search-space. Whether the geometric distribution allows different results might be an interesting problem.

For the rest of this work we only consider independent identically, exponentially distributed task processing times.

Chapter 5

In-Tree Constraints

When scheduling with in-tree constraints the dependencies between the tasks form an in-tree. This is the case, if each task constraints at most one other task and each task can be constrained by an arbitrary number of tasks. The problem is hierarchically structured with a single ending task that is the last one to be processed.

For simplicity we assume that the constraints form a single tree. If not, we simply add a new root that has all trees of the forest as children (the expected time of the forest is the expected time of the constructed tree minus the expected time of processing the new root task, since the new root task does not need to be finished).

5.1 An Introductory Example

Amount	Measure	Ingredient – Preparation Method
1	cup	Sugar
2/3	cup	All-purpose flour (divided 1/3 and 1/3 cups)
2	large	Eggs – lightly beaten
1 1/3	cups	Sour cream
1	teaspoon	Vanilla extract
3	cups	Fresh or frozen blackberries – thaw if frozen
1		Unbaked 9-inch pastry shell
1/3	cup	Firmly packed brown sugar
1/4	cup	Chopped toasted pecans
3	tablespoons	Softened butter
		Whipped cream, Fresh whole berries (opt.)

1. Preheat oven to 400 degrees.
2. Mix together sugar, 1/3 cup flour, eggs, sour cream and vanilla. Blend until smooth.
3. Gently fold in blackberries.
4. Spoon mixture into pastry shell.
5. Bake 30-35 minutes or until center is set.
6. Combine remaining 1/3 cup flour, brown sugar, pecans and softened butter. Mix together well.
7. Sprinkle over hot pie.
8. Return pie to oven for 10 minutes or until golden brown.
9. Remove from oven and cool on wire rack.
10. Garnish with whipped cream and whole berries if desired.

Figure 5.1: Blackberry Cream Pie Recipe (see http://members.aol.com/Jimg005/pie6.html)

Take a recipe for a Blackberry Cream Pie (see Figure 5.1) for an example of a hierarchical problem. Assume that every step listed is expected to take the same time (“long” steps can be divided, s.t. all steps take about the same expected time). If the pie is prepared by more than one person, tasks can be executed in parallel. Figure 5.2 shows two trees, the first being the sole structure displaying what needs to be done in sequence and what can be done in parallel (the node ’P’ represents the preparing of the whole berries), the second tree has been rearranged, such that each task takes about the same amount of time (the whole thing is abused a bit as to result in a ’nicer’ tree – baking for 30–35 minutes can hardly be modeled with an exponential distribution).

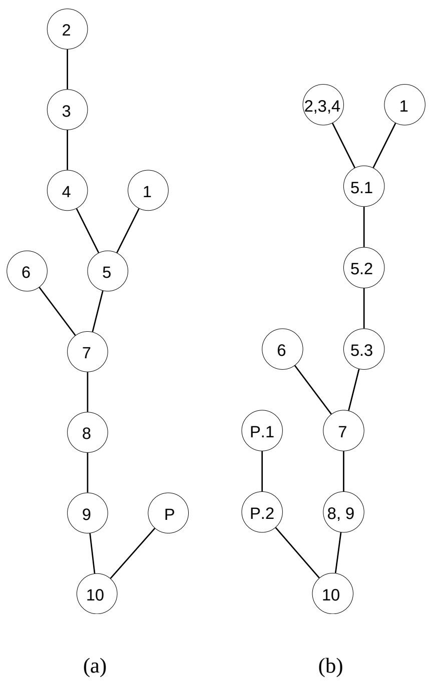


Figure 5.2: Hierarchical Structure of Preparing a Blackberry Cream Pie

Under the assumption that the duration of each step of the tree depicted in Figure 5.2 (b) is an independent identically, exponentially distributed random value, the tree is an instance of $(P3|intree|\mathbb{E}(C_{max}))$. Assume that the expected value for a single task is $\frac{1}{\lambda} = 10\,min$, then there are four possibilities of an initial schedule (where two are equivalent):

Task of Person A	Task of Person B	Task of Person C	Expected Time
2,3,4	6	P.1	1h, 17min 53s
1	6	P.1	1h, 17min 53s
1	2,3,4	6	1h, 16min, 5s
1	2,3,4	P.1	1h, 16min, 4s

As the calculated results show, it is advantageous to start preparing the whole berries before starting the second layer by about one second (although this does not seem obvious because task 6 is represented by a higher node).

5.2 Calculating an Optimal Solution for a Smaller Example

To show how the expected times for a schedule are calculated we will take a smaller example tree that is shown in Figure 5.3.

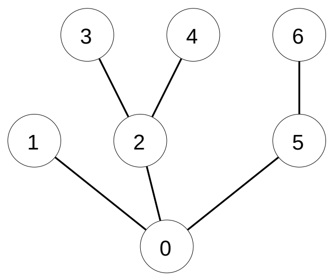


Figure 5.3: Small Tree Example

The expected time can be calculated by a recursive algorithm (see section 5.3). The scheduling process is divided into time intervals, at the end of each a machine finishes a task and the scheduler assigns the machine to a new task (if available). This points are often called decision points. For a tree B there are $|B|$ decision points. From each decision point there are one or more possibilities to choose a new schedule. Once the schedule is chosen, the machines start (or continue) working on their tasks until one machine finishes. The decisions are assumed to take no time (there are no setup times). Which machine finishes first is a random event. Figure 5.4 shows a DAG of subtrees with scheduled nodes that describes this process. The dark-framed DAG-nodes are those where the scheduler decides on the next step, from the light-framed DAG-nodes a random decision is taken. The expected length of the interval between two decision points depends on the number of machines working (see Lemma 4.5 in the previous chapter). The goal is to choose the schedules in such a way that the expected sum of the interval lengths is minimized.

5.3 Calculating the Solution Value of a Strategy

In the following we will be concerned with the scheduling problem $(P3|intree|\mathbb{E}(C_{max}))$. The number of machines is fixed to 3. The optimization criterion will always be the expected total processing time. The individual task processing times will all be independent identically, exponentially distributed with parameter λ. From chapter 4 we know that we can assume $\lambda = 1$ and multiply the result with $\frac{1}{\lambda}$. Therefore a description of a concrete instance of the problem will only need to include the precedence tree with all the tasks. The expected total processing depends only upon the strategy S chosen and the in-tree B of the constraints. From 4.2 we know that there are only $n = |B|$ points in time where scheduling decisions can be made by the strategy. In the following let $\alpha_S^B(t)$ be the set of leaves chosen for processing at time t. If the problem requires non-preemptive scheduling, if there is more than one machine available, and if the problem is a stochastic one, then α does not only depend on the time t, but rather on the history of the schedule (which tasks have finished, or have become available) and on which machines are still processing other tasks. For $(P3|intree|\mathbb{E}(C_{max}))$ the history can simply be described by the remaining tree B' and by the set of currently scheduled tasks β (e.g. as marked leaves). α_S can then be seen as a function of B' and β, where at the beginning $B' = B$ and $\beta = \emptyset$: $\alpha_S(B', \beta)$ are the tasks which are to be scheduled. The time does not play a role any more because of the memory-less property of the exponential distribution.

We will also use the function α to identify a strategy (leaving out the subscript S if it is clear from context). From the previous chapter we know the expected time of the first of m machines to finish. Let there be n tasks. An instance of a schedule will follow this scheme:

1. Schedule at least three tasks that are leaves of the tree.

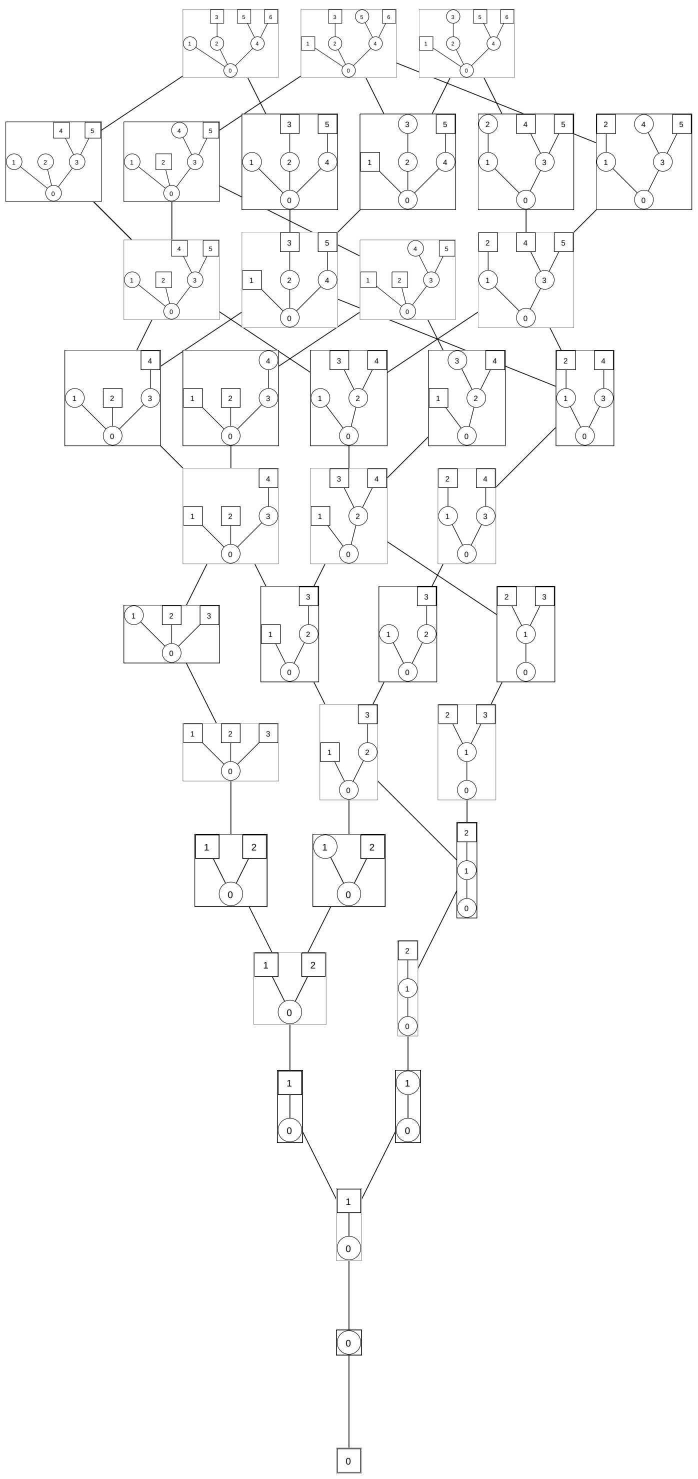


Figure 5.4: The Calculation DAG for the Example Tree in Figure 5.3. (The subtrees are ordered left to right in ascending expected total processing time. Each DAG node is represented with the subtree it stands for. In these subtrees, square nodes denote scheduled leaves.)

2. After some time t_i one task (the i-th) is finished and the machine can be assigned to another leaf, if there is still one left (some task that has been an inner node might have become available now because all its children have been finished).

3. The previous step is iterated until the last task finishes.

The total time is

$$T = \sum_{1}^{n} t_i$$

Let the n tasks be assigned numbers 1 through n. In a concrete instance of a schedule task i will be finished as $p(i)$-th where p is a permutation over $\{1, \ldots, n\}$. For a given in-tree B let $CP(B)$ (Concrete instance Permutations) be the set of all permutations that correspond to a concrete instance of a schedule (they are feasible, if for any nodes i and j: if j is an ancestor of i, then j will never occur before i in any permutation in $CP(B)$: $\forall i, j \in B : \exists path(i, j) \Rightarrow \forall p \in CP(B) : p(i) < p(j)$). We will call a feasible permutation an **execution path**.

The expected time of the above process is determined by the expected finishing times in step 2. Since all tasks are independent identically distributed, the expected time $\mathbb{E}(t_i)$ depends only on the number of machines that were working at the time task i was finished.

A recursive algorithm for calculation of the total expected processing time, given a strategy function $\alpha_S(B, \beta)$ is given as Algorithm 1.

Algorithm 1 $\mathbf{Frame(B}, \beta, \alpha_\mathbf{S})$

```
1:  proc  $Frame(B, \beta, \alpha_S)$ :
2:  if $|B| = 1$ then
3:     return(1)
4:  else
5:     $\sigma \longleftarrow \alpha_S(B, \beta)$
6:     $s \longleftarrow 0$
7:     for all $i \in \sigma$ do
8:        $\beta' \longleftarrow \sigma \setminus \{i\}$
9:        $B' \longleftarrow B \setminus \{i\}$
10:       $s \longleftarrow s + Frame(B', \beta', \alpha_S)$
11:    end for
12:    $s \longleftarrow (s + 1)/(|\sigma|)$
13:    return($s$)
14: end if
```

Given an execution path $p \in CP(B)$, we can determine the tasks i_1 and i_2 (permuted numbers) after which there are only one or two leaves left. Let $s_2(p) = i_2$ and $s_1(p) = i_1$. s_1 and s_2 are dependent on the tree B and the execution path p.

Let $P_\alpha^B[p]$ be the probability that the execution path described by p occurs under the strategy α and tree B.

Theorem 5.1 (Expected Processing Time for Strategy α). *The expected processing time $\mathbb{E}(T_\alpha)$ for a scheduling problem with n independent identically, exponentially distributed tasks with individual expected processing time $\frac{1}{\lambda}$ and constraints defined by the in-tree B is*

$$\mathbb{E}(T_\alpha) = \frac{1}{\lambda} \cdot \sum_{p \in CP} P_\alpha^B[p] \cdot \left(s_2(p) \cdot \frac{1}{3} + (s_1(p) - s_2(p)) \cdot \frac{1}{2} + (n - s_1(p)) \right) \tag{5.1}$$

Proof. The proof follows from the above definitions and by summing over all possible concrete schedule instances:

$$
\begin{aligned}
\mathbb{E}(T_\alpha) &= \sum_{p\in CP} P_\alpha^B[p] \cdot \Big(\text{Expected time of concrete schedule instance corresponding to } p\Big) \\
&= \sum_{p\in CP} P_\alpha^B[p] \cdot \left(\sum_{i=1}^{s_2(p)} \frac{1}{3}\cdot\frac{1}{\lambda} + \sum_{i=s_2(p)+1}^{s_1(p)} \frac{1}{2}\cdot\frac{1}{\lambda} + \sum_{i=s_1(p)+1}^{n} \frac{1}{1}\cdot\frac{1}{\lambda} \right) \\
&= \frac{1}{\lambda}\cdot \sum_{p\in CP} P_\alpha^B[p] \cdot \left(\sum_{i=1}^{s_2(p)} \frac{1}{3} + \sum_{i=s_2(p)+1}^{s_1(p)} \frac{1}{2} + \sum_{i=s_1(p)+1}^{n} 1 \right) \\
&= \frac{1}{\lambda}\cdot \sum_{p\in CP} P_\alpha^B[p] \cdot \Big((s_2(p)-1+1)\cdot\frac{1}{3} + (s_1(p)-s_2(p)-1+1)\cdot\frac{1}{2} + (n-s_1(p)-1+1) \Big) \\
&= \frac{1}{\lambda}\cdot \sum_{p\in CP} P_\alpha^B[p] \cdot \Big(s_2(p)\cdot\frac{1}{3} + (s_1(p)-s_2(p))\cdot\frac{1}{2} + (n-s_1(p)) \Big) \\
&= \frac{1}{\lambda}\cdot \sum_{p\in CP} P_\alpha^B[p] \cdot \Big(n - \frac{1}{6}\cdot s_2(p) - \frac{1}{2}\cdot s_1(p) \Big)
\end{aligned}
$$

□

Equation 5.1 can also be rewritten such that the emphasis lies more on the number of times three, two, or only one machine can work at the tree.

Let TWO be the number of the task after which there are only two leaves left (the concrete value of $s_2(p)$ for a concrete execution path p), and let ONE be the number of the task after which there is only one leaf left (the concrete value of $s_1(p)$ for a concrete execution path p). Clearly for a given tree B and a strategy α TWO and ONE are random variables.

Corollary 5.1. *Let $\mathbb{E}(TWO)$ ($\mathbb{E}(ONE)$) be the expected value of TWO (ONE) under strategy α and in-tree B. The expected processing time $\mathbb{E}(T_\alpha)$ for a scheduling problem with n exponentially distributed tasks with individual expected finishing time $\frac{1}{\lambda}$ and constraints defined by the in-tree B is*

$$\mathbb{E}(T_\alpha) = \frac{1}{\lambda}\cdot\Big(n - \frac{1}{6}\cdot\mathbb{E}(TWO) - \frac{1}{2}\cdot\mathbb{E}(ONE)\Big) \tag{5.2}$$

or

$$\mathbb{E}(T_\alpha) = \frac{1}{\lambda}\cdot\Big(\frac{1}{3}\cdot\mathbb{E}(TWO) + \frac{1}{2}\cdot(\mathbb{E}(ONE)-\mathbb{E}(TWO)) + 1\cdot(n-\mathbb{E}(ONE))\Big) \tag{5.3}$$

where $\mathbb{E}(TWO)$ is the expected number of steps with three machines, $(\mathbb{E}(ONE) - \mathbb{E}(TWO))$ is the expected number of steps with two machines, and $(n - \mathbb{E}(ONE))$ is the expected number of steps with one machine.

Proof. The expected values of ONE and TWO are:

$$\mathbb{E}(TWO) = \sum_{p\in CP} P_\alpha^B[p]\cdot s_2(p)$$

$$\mathbb{E}(ONE) = \sum_{p\in CP} P_\alpha^B[p]\cdot s_1(p)$$

Therefore from Theorem 5.1:

$$\begin{aligned}\mathbb{E}(T_\alpha) = \frac{1}{\lambda} \cdot \left(\sum_{p \in CP} P_\alpha^B[p] \cdot \left(\frac{1}{3} \cdot s_2(p) + \frac{1}{2} \cdot (s_1(p) - s_2(p)) + (n - s_1(p)) \right) \right) = \\ \frac{1}{\lambda} \cdot \left(\sum_{p \in CP} P_\alpha^B[p] \cdot \left(-\frac{1}{6} \cdot s_2(p) - \frac{1}{2} \cdot s_1(p) + n \right) \right) = \\ \frac{1}{\lambda} \cdot \left(n \cdot \sum_{p \in CP} P_\alpha^B[p] - \frac{1}{6} \cdot \sum_{p \in CP} P_\alpha^B[p] \cdot s_2(p) - \frac{1}{2} \cdot \sum_{p \in CP} P_\alpha^B[p] \cdot s_1(p) \right) = \\ \frac{1}{\lambda} \cdot \left(n - \frac{1}{6} \cdot \mathbb{E}(TWO) - \frac{1}{2} \cdot \mathbb{E}(ONE) \right)\end{aligned}$$

The second equation is a simple transformation of the first. □

As can be seen from Theorem 5.1 and Corollary 5.1 the value $\frac{1}{\lambda}$ is always an outer constant factor. We will therefore often use 1 as the expected value of the exponential distributions of the task processing times. The corresponding results only need to be multiplied by the real expected value (e.g. $\frac{1}{\lambda}$ =“one hour”, $\frac{1}{\lambda}$ =“3 days”, . . .).

Chapter 6

Calculating the Optimal Schedule for $(P3|intree|\mathbb{E}(C_{max}))$

6.1 A Simple Recursive Algorithm

A trivial algorithm to calculate the total expected processing time would calculate the probabilities $P_\alpha^B[p]$ of every execution path $p \in CP(B)$ and multiply by the expected processing time $s_2(p) \cdot \frac{1}{3} + (s_1(p) - s_2(p)) \cdot \frac{1}{2} + (n - s_1(p))$.

Because all processing times are independently distributed, given a set of scheduled tasks every task has the same probability of finishing first. An algorithm can simply calculate the optimal expected processing time in the following steps:

1. For each possible set of leaves to schedule,

   for each leaf in that set,

   remove the leaf from the tree and recursively calculate the optimal value for the case that this leaf is finished first.

2. Add all cases, divide by the number of cases and add the minimal expected finishing time of the first node (1, $\frac{1}{2}$, or $\frac{1}{3}$, see chapter 4).

3. Select the best of all schedules.

The central procedure is given as Algorithm 2.

The algorithm that starts the calculation and receives the result in s_{min} looks like

```
$s_{min} \longleftarrow \infty$
for all $\beta \subseteq leaves(B)$ with $|\beta| = 2 \vee (|\beta| < 2 \wedge \beta = leaves(B))$ do
  $s \longleftarrow Opt_{recursive}(B, \beta)$
  if $s < s_{min}$ then
    $s_{min} \longleftarrow s$
  end if
end for
```

We will not give the time complexity of this algorithm here, but first proceed with a small modification. An upper bound on the space complexity is easy to prove.

Lemma 6.1. *$(P3|intree|\mathbb{E}(C_{max}))$ belongs to $PSPACE$.*

Proof. The above algorithm solves $(P3|intree|\mathbb{E}(C_{max}))$ optimally. In each call to $Opt_{recursive}$ the size of the tree is reduced by one node. Hence the call stack is at most $|B|$ deep. For each call the parameters

Algorithm 2 $\mathbf{Opt_{recursive}}(\mathbf{B}, \beta)$

Require: $|\beta| = 2 \vee (|leaves(B)| < 3 \wedge |leaves(B)| - 1 \leq |\beta| \leq |leaves(B)|)$

```
 1: proc  $Opt_{recursive}(B, \beta)$ :
 2: if $|leaves(B)| = 1$ then
 3:   return$(|B|)$
 4: else
 5:   if $|leaves(B)| = 2$ then
 6:     $\sigma \longleftarrow leaves(B)$
 7:     $s \longleftarrow 0$
 8:     for all $i \in \sigma$ do
 9:       $\beta' \longleftarrow \sigma \setminus \{i\}$
10:       $B' \longleftarrow B \setminus \{i\}$
11:       $s \longleftarrow s + Opt_{recursive}(B', \beta')$
12:     end for
13:     $s \longleftarrow (s + 1)/(|\sigma|)$
14:     return$(s)$
15:   else        $/* \ |leaves(B)| \geq 3 \ */$
16:     $s_{min} \longleftarrow \infty$
17:     for all $l \in leaves(B) \setminus \beta$ do
18:       $\sigma \longleftarrow \beta \cup \{l\}$
19:       $s \longleftarrow 0$
20:       for all $i \in \sigma$ do
21:         $\beta' \longleftarrow \sigma \setminus \{i\}$
22:         $B' \longleftarrow B \setminus \{i\}$
23:         $s \longleftarrow s + Opt_{recursive}(B', \beta')$
24:       end for
25:       $s \longleftarrow (s + 1)/(|\sigma|)$
26:       if $s < s_{min}$ then
27:         $s_{min} \longleftarrow s$
28:       end if
29:     end for
30:     return$(s_{min})$
31:   end if
32: end if
```

and some sets of at most constant size must be remembered. As a result, the maximal space needed is $\mathcal{O}(|B|^2)$. □

6.2 A Dynamic Programming Algorithm

It is easy to see that Algorithm 2 visits a lot of trees multiple times. We can use a hash table to remember optimal schedules for the pair $\langle B, \beta\rangle$. The algorithm could thus be altered to look in a central hash table H. The procedure $Opt_{DP}(B, \beta)$ for this algorithm is given as Algorithm 3.

Algorithm 3 $\mathbf{Opt_{DP}(B}, \beta)$

```
Require: $|\beta| = 2 \vee (|leaves(B)| < 3 \wedge |leaves(B)| - 1 \leq |\beta| \leq |leaves(B)|)$
proc  $Opt_{DP}(B, \beta)$ :
if $H$ contains $\langle B, \beta\rangle$ then
  return$(H(\langle B, \beta\rangle))$
end if
if $|leaves(B)| = 1$ then
  return$(|B|)$
else
  if $|leaves(B)| = 2$ then
    $\sigma \longleftarrow \beta$
    $s \longleftarrow 0$
    for all $i \in \sigma$ do
      $\beta' \longleftarrow \sigma \setminus \{i\}$
      $B' \longleftarrow B \setminus \{i\}$
      $s \longleftarrow s + Opt_{DP}(B', \beta')$
    end for
    $s \longleftarrow (s + 1)/(|\sigma|)$
    $H \longleftarrow H \cup (\langle B, \beta\rangle, s)$
    return$(s)$
  else        $/* |leaves(B)| \geq 3 */$
    $s_{min} \longleftarrow \infty$
    for all $l \in leaves(B) \setminus \beta$ do
      $\sigma \longleftarrow \beta \cup \{l\}$
      $s \longleftarrow 0$
      for all $i \in \sigma$ do
        $\beta' \longleftarrow \sigma \setminus \{i\}$
        $B' \longleftarrow B \setminus \{i\}$
        $s \longleftarrow s + Opt_{DP}(B', \beta')$
      end for
      $s \longleftarrow (s + 1)/(|\sigma|)$
      if $s < s_{min}$ then
        $s_{min} \longleftarrow s$
      end if
    end for
    $H \longleftarrow H \cup (\langle B, \beta\rangle, s_{min})$
    return$(s_{min})$
  end if
end if
```

The number of steps performed depends on the number of subtrees visited. With the usage of the hash table no subtree is visited twice with the same combination of leaves (we could equally use a two step hashing, first from the tree to a hash table that hashes from a set of leaves to the minimal expected processing time).

Each subtree B' of B is visited at most $\mathcal{O}(leaves(B')^2) = \mathcal{O}(|B'|^2) = \mathcal{O}(|B|^2)$ times. The number of subtrees of a tree B depends on the tree structure, because the structure determines what subsets of the set of all permutation over $\{1, \dots, |B|\}$ are feasible (no non-leaves are removed at any point). An upper bound for the number of different subtrees is the number of permutations over $\{1, \dots, |B|\}$, which is $|B|!$. These considerations lead to the following lemma:

Lemma 6.2 (Complexity of Algorithm 3). *The running time of Algorithm 3 is in $\mathcal{O}(|B'|^2 \cdot |B|!)$.*

Proof. See considerations above. □

6.3 Excluding Isomorphic Subtrees

Algorithm 3 can be further optimized by removing isomorphic subtrees from the list of considered subtrees and by further ignoring sets of scheduled leaves of any subtree that are isomorphic to already seen sets of scheduled leaves. To do this we need a unique representation of a subtree with some scheduled leaves.

Observation 6.1 (Unique Representation of Rooted, Unordered Trees). *Rooted, unordered trees can be represented uniquely by recursively sorting the children by their in-degrees. If two children have the same in-degree, the degrees of the children's children are compared recursively. Scheduled leaves can be included by defining their in-degree as -1 (while other leaves' in-degree is 0).*

The algorithm for sorting a tree is given as Algorithm 5. We assume in-trees and that the children of a node n are ordered and can be retrieved by $child_i^n$.

Algorithm 4 $\mathbf{sortTree_{compare}(a, b)}$

```
1: if $d_{in}(a) < d_{in}(b)$ then
2:   $\mathbf{return}(-1)$
3: else if $d_{in}(a) > d_{in}(b)$ then
4:   $\mathbf{return}(1)$
5: else
6:   for $i$ from 1 to $d_{in}(a)$ do
7:     if $sortTree_{compare}(child_i^a, child_i^b) < 0$ then
8:       $\mathbf{return}(-1)$
9:     else
10:      if $sortTree_{compare}(child_i^a, child_i^b) > 0$ then
11:        $\mathbf{return}(1)$
12:      end if
13:    end if
14:  end for
15:  $\mathbf{return}(0)$
16: end if
```

Algorithm 5 $\mathbf{sortTree(B)}$

```
1: $r \longleftarrow root(B)$
2: for all $c \in in(r)$ do        /* all children of $r$ */
3:   $sortTree(B|_c)$        /* sort subtree with root $c$ */
4: end for
5: Sort the children of $r$ with $sortTree_{compare}()$.
```

Figure 6.1 shows an example of a tree and the same tree recursively sorted. The trees unique representation is $(2, 1, 3, 0, -1, -1, 2, 0, 2, 0, -1)$.

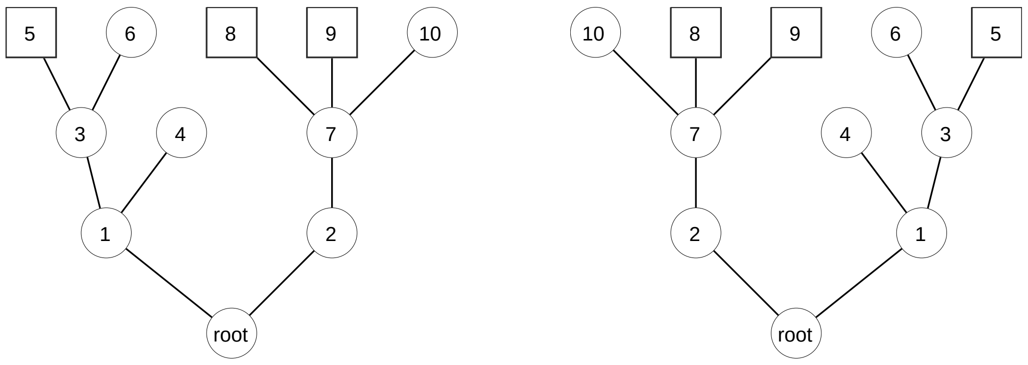


Figure 6.1: Example of a Tree and Same Tree Reordered

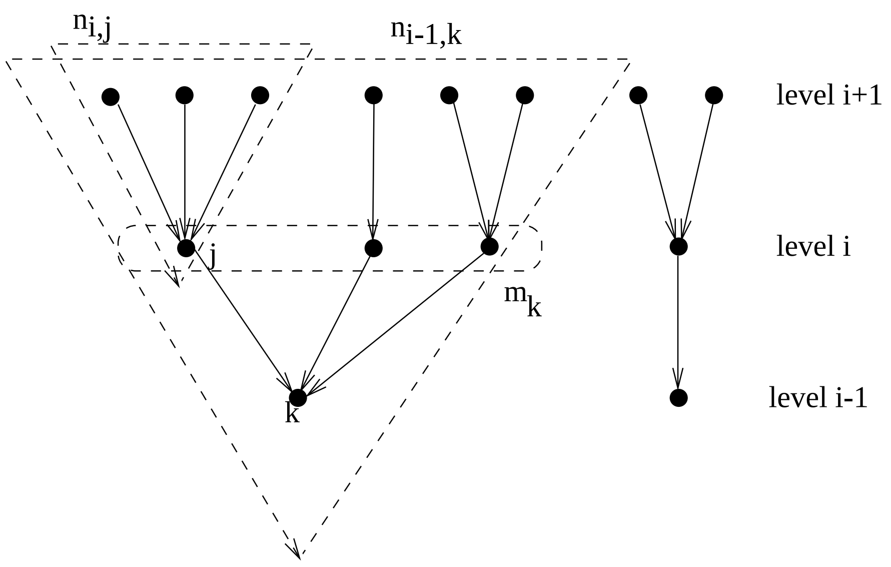


Figure 6.2: Sorting a Tree for a Unique Representation

Lemma 6.3 (Sorting a Tree for a Unique Representation). *Sorting a tree to extract the unique representation for comparison with other trees takes time $\mathcal{O}(n^2 \cdot \log(n))$.*

Proof. Let the tree have n nodes and height h. Let l_i be the number of nodes at level i (where $l_0 = 1$ is the root level and l_h is the highest level).

Let there be $n_{i,j}$ nodes in the subtree of node j at level i, then $\sum_{j=1}^{l_i} n_{i,j} = \sum_{j=i}^{h} l_j$.

Comparing a child j with another child j' of a node k at level $i-1$ takes at most $\min\{n_{i,j}, n_{i,j'}\} \leq n_{i,j}$. Suppose k has m_k children. With a sorting algorithm (such as mergesort) that makes $\mathcal{O}(nlog(n))$ comparisons ($log(n)$ per item), the sorting of the children of node k takes (see Figure 6.2)

$$\sum_{j=1}^{m_k} \log(m_k) \cdot n_{i,j} = \log(m_k) \cdot \sum_{j=1}^{m_k} n_{i,j} = \log(m_k) n_{i-1,k}$$

The number of operations t_{i-1} needed for sorting the children of the nodes at level $i-1$ is:

$$t_{i-1} = \sum_{k=1}^{l_{i-1}} \log(m_k) n_{i-1,k} \leq \log(l_{i-1}) \sum_{k=1}^{l_{i-1}} n_{i-1,k} \leq \log(l_{i-1}) \sum_{k=i-1}^{k} l_k \leq n \log(n)$$

Since there are at most n levels the sum over all levels is $\mathcal{O}(n^2 \log(n))$. □

This bound is not very tight and we conjecture that the real bound for sorting a tree is $\mathcal{O}(n \log(n))$.

We can prove a slightly better bound of $\mathcal{O}(n^2)$:

Lemma 6.4 (Sorting a Tree for a Unique Representation 2). *Sorting a tree to extract the unique representation for comparison with other trees takes time $\mathcal{O}(n^2)$.*

Proof. Let B be a tree that is sorted to the unique representation, let $n = |B|$. Let $sortOps(v)$ be the number of times the in-degree of node v was compared to the in-degree of another node. Let v_{max} be a node with $sortOps(v_{max}) = \max_v\{sortOps(v)\}$ and let $s_{max} = sortOps(v_{max})$.

Let v_{max} have k ancestors, ancestor 1 being the direct parent and ancestor k being the root. Further, let m_i be the number of comparisons of v_{max} that occurred during the sorting of the children of the i-th ancestor, and let n_i be the number of children of ancestor i. Since we use mergesort, the number of comparisons of the subtree including v_{max} can be at most $\log_2(n_i)$, therefore $m_i \leq \log_2(n_i)$. The maximal number of comparisons per node is then:

$$s_{max} = \sum_{i=1}^{k} m_i \leq \sum_{i=1}^{k} \log_2(n_i) \leq \sum_{i=1}^{k} n_i \leq n$$

As a result the total number of comparisons per node is $\mathcal{O}(n)$, and the total running time of the sorting algorithm is $\mathcal{O}(n^2)$ □

We will represent all isomorphic trees with a tree that has been sorted with Algorithm 5 and then labeled with a simple DFS. In the DAG of all (non-isomorphic) trees an edge will lead from a tree B to a tree B', if $B' = B \setminus \{a\}$ for some node a. For each edge from B to a smaller tree we need to construct $B' = B \setminus \{a\}$ and sort it. Running a simple DFS we can determine the mapping of node numbers from B to B'.

An outline of the algorithm that excludes isomorphic subtrees is given as Algorithm 6, the details are given as Algorithms 7, 8, and 9.

For simplicity we have left out the special cases, where there are only two or less leaves left in a subtree. Also note that there might be more than one optimal set of leaves to schedule, hence we might want to return a set of optimal leaf-triples in Algorithm 9.

Obviously the third part of the algorithm takes time $\mathcal{O}(leaves(B)^2) = \mathcal{O}(n^2)$.

Algorithm 6 $\mathbf{Opt_{DP,ISO}(B)}$

```
1: Build the DAG $D$ of all isomorphic subtrees. Label the edges between each subtree with the leaf that
   was removed and a mapping of the node numbers.
2: Calculate schedule for minimal expected processing time for all nodes in $D$
3: Calculate minimal expected processing time for $B$ and corresponding schedule and return
```

Algorithm 7 Part 1 of $\mathbf{Opt_{DP,ISO}(B}, \beta)$

```
1: $\sigma \longleftarrow \{B\}$
2: $D \longleftarrow$ empty DAG
3: $p \longleftarrow$ new node for $B$
4: insert $p$ in $D$
5: while $|\sigma| > 0$ do
6:    $\sigma' \longleftarrow \emptyset$
7:    for all $B'$ in $\sigma$ do
8:       $p \longleftarrow$ node in $D$ for $B'$
9:       for all $a \in leaves(B')$ do
10:         $B'_a \longleftarrow B' \setminus a$
11:         $S_a \longleftarrow sortTree(B'_a)$
12:         $m \longleftarrow$ mapping from $DFS(B'_a)$ to $DFS(S_a)$
13:         if $S_a \in \sigma'$ then
14:            $n \longleftarrow$ node for $S_a$ in $D$
15:         else
16:            $\sigma' \longleftarrow \sigma' \cup \{S_a\}$
17:            $n \longleftarrow$ new node for $S_a$
18:            insert $n$ in $D$
19:            $tree(n) \longleftarrow S_a$
20:         end if
21:         add edge from $p$ to $n$ in $D$ labeled with $a, m$
22:      end for
23:   end for
24:   $\sigma \longleftarrow \sigma'$
25: end while
```

Algorithm 8 Part 2 of $\mathbf{Opt}_{\mathbf{DP},\mathbf{ISO}}(\mathbf{B}, \beta)$

```
1:  for $i$ from 1 to $|B|$ do
2:      for all $n$ with $n \in D$ and $|tree(n)| = i$ do
3:          for all $a, b \in leaves(tree(n))$ do
4:              $v_{min} \longleftarrow \infty$
5:              $l_{min} \longleftarrow \perp$
6:              $(n_a, m_a) \longleftarrow$ node pointed to by edge labeled with $a$, mapping stored as edge label
7:              $(n_b, m_b) \longleftarrow$ node pointed to by edge labeled with $b$, mapping stored as edge label
8:              for all $c \in leaves(tree(n)) \setminus \{a, b\}$ do
9:                  $(n_c, m_c) \longleftarrow$ node pointed to by edge labeled with $c$, mapping stored as edge label
10:                 $v \longleftarrow \Big(opt_2(n_a, m_a(DFS(b)), m_a(DFS(c))) + opt_2(n_b, m_b(DFS(a)), m_b(DFS(c))) + opt_2(n_c, m_c(DFS(a)), m_c(DFS(b))) + 1\Big)/3$
11:                 if $v < v_{min}$ then
12:                     $v_{min} \longleftarrow v$
13:                     $l_{min} \longleftarrow c$
14:                 end if
15:             end for
16:             $opt(n, DFS(a), DFS(b)) \longleftarrow (l_{min}, v_{min})$
17:         end for
18:     end for
19: end for
```

Algorithm 9 Part 3 of $\mathbf{Opt}_{\mathbf{DP},\mathbf{ISO}}(\mathbf{B}, \beta)$

```
1:  $v_{min} \longleftarrow \infty$
2:  $s_{min} \longleftarrow \perp$
3:  $n \longleftarrow$ node for $B$ in $D$
4:  $S \longleftarrow tree(n)$
5:  for all $a, b \in leaves(S)$ do
6:      if $opt_2(n, DFS(a), DFS(b)) < v_{min}$ then
7:          $v_{min} \longleftarrow opt_2(n, DFS(a), DFS(b))$
8:          $s_{min} \longleftarrow \langle a, b, opt_1(n, DFS(a), DFS(b)) \rangle$
9:      end if
10: end for
11: $\mathbf{return}(v_{min}, s_{min})$
```

The second part basically iterates over all nodes in the DAG of subtrees once in lines 1 and 2. The inner part of the loop in line 8 is iterated $\mathcal{O}(leaves(B')^3) = \mathcal{O}(leaves(B)^3) = \mathcal{O}(n^3)$ times. All operations are otherwise constant. Hence the second part takes time $\mathcal{O}(n^3 \cdot |D|)$, where D is the DAG of subtrees of B.

The first part of the algorithm constructs D, the DAG of subtrees of B. The loops in lines 5 and 7 are executed $|D|$ times. The inner part of the loop at line 9 is therefore executed $|D| \cdot leaves(B') = \mathcal{O}(n|D|)$ times. The set look-ups can be implemented in average time $\mathcal{O}(1)$ (one comparison) and the sorting of the tree takes times $\mathcal{O}(n^2)$ (see Lemma 6.4). Hence the overall running time of part three is $\mathcal{O}(n^3|D|)$.

Summing up we get:

Lemma 6.5 (Running Time of Algorithm 6). *Algorithm 6 takes time $\mathcal{O}(|B|^3|D|)$, where D is the DAG of subtrees of B.*

Proof. See considerations above. □

The question that is left is the size of the DAG of subtrees of B. This DAG excludes isomorphic subtrees, otherwise the worst case size would be $n!$ (following the same consideration as for Algorithm 3 in section 6.2).

6.4 A Bound for the Worst Case Size of the DAG of Subtrees

We will only give a lower bound for the worst case size of D. This lower bound is reached by analyzing the number of non-isomorphic subtrees of binary trees. The n-th binary tree B_n is constructed by adding two complete binary trees B_{n-1} to a new root.

Lemma 6.6 (Asymptotic Size of Subtree DAG of Complete Binary Trees). *The asymptotic size of the subtree DAG of a complete binary tree B_n with height n is*

$$2^{c \cdot 2^n}$$

where $0.6346563536974 \leq c \leq 0.63477505712875$.

Proof. Because of the recursive composition of binary trees we will use induction to calculate the number of distinct subtrees.

The size of the DAG of B_0 is $b_0 = 1$ and of B_1 is $b_1 = 3$.

Let b_n be the number of distinct subtrees of B_n. When combining the binary tree B_n of the next order, we can estimate the number of non-isomorphic subtrees as the combination of two non-isomorphic subtrees of B_{n-1}. Combining the same kinds of subtree twice would introduce a pair of isomorphic subtrees, hence we can enumerate the subtrees of B_{n-1} and only combine subtrees where the first number is higher or equal to the second. Additionally we can combine all non-isomorphic subtrees of B_{n-1} with no tree (or the empty tree) on the other side and add the subtree for the single root node. This leads to the following recursive formula:

$$b_n = \sum_{i=0}^{b_{n-1}} i + \sum_{i=0}^{b_{n-1}} 1 + 1 = \sum_{i=0}^{b_{n-1}} (i+1) + 1 = \sum_{i=1}^{b_{n-1}+1} i + 1 = \sum_{i=0}^{b_{n-1}+1} i = \frac{1}{2}(b_{n-1}+1)(b_{n-1}+2)$$

To get an asymptotic bound for $b_n = (b_{n-1}+1)(b_{n-1}+2)/2$, we will take the (dual) logarithm:

$$\begin{array}{rrl}
 & b_n & = \frac{1}{2}(b_{n-1}+1)(b_{n-1}+2) \\
\Rightarrow & \log_2(b_n) & = \log_2(\frac{1}{2}(b_{n-1}+1)(b_{n-1}+2)) \\
 & & = \log_2(\frac{1}{2}b_{n-1}^2 + \frac{3}{2}b_{n-1} + 1) \\
 & & = \log_2(\frac{1}{2}b_{n-1}^2 \cdot (1 + \frac{3}{b_{n-1}} + \frac{2}{b_{n-1}^2})) \\
 & & = \log_2(\frac{1}{2}) + \log_2(b_{n-1}^2) + \log_2(1 + \frac{3}{b_{n-1}} + \frac{2}{b_{n-1}^2}) \\
 & & = -1 + 2\log_2(b_{n-1}) + \log_2(1 + \frac{3}{b_{n-1}} + \frac{2}{b_{n-1}^2}) \\
 & & = -1 + 2(-1 + 2\log_2(b_{n-2}) + \log_2(1 + \frac{3}{b_{n-2}} + \frac{2}{b_{n-2}^2})) + \log_2(1 + \frac{3}{b_{n-1}} + \frac{2}{b_{n-1}^2}) \\
 & & = -1 \cdot \sum_{i=0}^{n-2} 2^i + 2^{n-1}\log_2(b_1) + \sum_{i=0}^{n-2} 2^i \cdot \log_2(1 + \frac{3}{b_{n-1-i}} + \frac{2}{b_{n-1-i}^2}) \\
 & & = -2^{n-1} + 1 + 2^{n-1}\log_2(b_1) + \sum_{i=0}^{n-2} 2^i \cdot \log_2(1 + \frac{3}{b_{n-1-i}} + \frac{2}{b_{n-1-i}^2}) \\
\Rightarrow & \log_2(b_n)/2^n & = -\frac{1}{2} + \frac{1}{2^{n-1}} + \frac{\log_2(3)}{2} + \sum_{i=0}^{n-2} \frac{1}{2^{n-i}} \cdot \log_2(1 + \frac{3}{b_{n-1-i}} + \frac{2}{b_{n-1-i}^2}) \\
 & & = \frac{\log_2(3)}{2} - \frac{1}{2} + \frac{1}{2^{n-1}} + \sum_{i=2}^{n} \frac{1}{2^i} \cdot \log_2(1 + \frac{3}{b_{i-1}} + \frac{2}{b_{i-1}^2})
\end{array}$$

Since b_n is an increasing function, the term $\frac{1}{2^i} \cdot \log_2(1 + \frac{3}{b_{i-1}} + \frac{2}{b_{i-1}^2})$ decreases very fast as $i \to \infty$:

$$\begin{array}{rcl}
\log_2(b_n)/2^n & \underset{n\to\infty}{\longrightarrow} & c \\
\log_2(b_n) & \underset{n\to\infty}{\longrightarrow} & c \cdot 2^n \\
b_n & \underset{n\to\infty}{\longrightarrow} & 2^{c \cdot 2^n}
\end{array}$$

To calculate the bounds on c, we simply evaluate $\sum_{i=2}^{n} \frac{1}{2^i} \cdot \log_2(1 + \frac{3}{b_{i-1}} + \frac{2}{b_{i-1}^2})$ for the first four terms, resulting in the lower bound 0.6346563536974. Since the individual log-terms decrease, we add $\sum_{i>=6} \frac{1}{2^i} \cdot \log_2(1 + \frac{3}{b_4} + \frac{2}{b_4^2})$ to receive the upper bound of 0.63477505712875. □

Since the number of nodes of the binary trees is asymptotic to 2^n, the size of the subtree DAG of a binary tree with n nodes (we could always calculate the size of the next smallest binary tree) is $\Omega(2^{c \cdot n})$. This is also a lower bound for worst case size of the subtree DAG of a tree with n nodes.

The dynamic programming algorithm is therefore at least exponential.

From our empirical results and from their inner structure we conjecture that binomial trees are the trees with the largest number of non-isomorphic subtrees. The number of subtrees b_n (including isomorphic ones) of the n-th binomial tree is given by the recurrence

$$b_n = \begin{cases} b_{n-1} \cdot (b_{n-1}+1) & \text{if } n > 0, \\ 1 & \text{otherwise.} \end{cases}$$

The asymptotic behavior of b_n is given by $2^{c \cdot 2^n}$, with $0.67618 \le c \le 0.67819$ (see [Urq95], page 434). This is of course also an upper bound on the number of non-isomorphic subtrees.

6.5 Optimizations for $\mathbf{Opt_{DP,ISO}(B)}$

Although the exponential nature of the algorithm cannot be altered, we will give some further hints on optimizing the algorithm.

The following optimizations can additionally be applied:

- Iterate only over edges of DAG nodes (in line 3 and 8 of Algorithm 8.) This should exclude redundant work for leaves whose removal will lead to the same tree (thus avoid to meet these redundancies twice, because they were already observed in line 13 of Algorithm 8)

- In the preemptive case, there is no need to store a mapping from the nodes of one tree to another, the nodes can simply be reassigned. Therefore the inner loop of Algorithm 8 can easily be replaced by sorting the edges of the corresponding DAG node by the expected processing time of their destination nodes and take the nodes labeling the lowest three edges as schedule.

- For this work there was the need to quickly find counter examples to given scheduling strategies. If all trees with a given number of nodes are to be calculated, part 1 of Algorithm 6 can be replaced by an algorithm similar to Algorithm 7, that starts at the single node tree and level wise calculates the DAG by adding leaves (instead of removing). The complexity is the same as for Algorithm 6. The resulting DAG size is asymptotic to (see 2.2):

  $$a_n = \frac{1}{\alpha^{n-1} n}\sqrt{\beta/2\pi n} + \mathcal{O}\left(\frac{1}{\sqrt{n^5}\alpha^n}\right)$$

  where $1/\alpha \approx 2.955765285652$ and $\sqrt{\beta/2\pi} \approx 0.439924012571$.

- When generating the DAG of all rooted, unordered tree as described above, the computation of level $n+1$ only depends upon level n.

  The size of the levels grows with 1, 1, 2, 4, 9, 20, 48, 115, 286, 719, 1842, 4766, 12486, 32973, 87811, 235381, 634847, 1721159, ... To be able to reach to level 18 (all trees with 18 nodes) measures must be taken to keep the size of the tree small. Since today it seems virtually impossible to reach anything higher than 26 nodes with a normal Workstation (5759636510 trees, the number exceeding a 32bit integer constant), a tree representation can stick to numbers smaller than 32.

  Also it is nearly impossible to keep this amount of data in main memory. Therefore the trees need to be clustered. A trivial clustering could take the degree of the root node (although this only “gains” a level).

6.6 Using m Machines to Schedule an In-Tree

The only values of m for which an efficient algorithm for $(Pm|intree|\mathbb{E}(C_{max}))$ is known are 1 and 2. When $m > 2$ it seems very hard to find an efficient (polynomial time) algorithm (although HLF is near optimal). The following result shows that the usage of more machines results in a better total expected time and thus gives a reason, why it is of interest to be able to schedule more than two machines.

Lemma 6.7. *For any given in-tree B and m machines, let $\mathbb{E}(T_{\alpha_m}(B))$ be the optimal expected total processing time for $(Pm|intree|\mathbb{E}(C_{max}))$. Then*

$$\forall m_l, m_s : m_l > m_s \Rightarrow \forall B : \mathbb{E}(T_{\alpha_{m_l}}(B)) \leq \mathbb{E}(T_{\alpha_{m_s}}(B))$$

(Using more machines always results in a better optimal expected total processing time).

Proof. Given the optimal strategy α_{m_s} for m_s machines we will show that there is a strategy for m_l machines with an expected value that is no worse than $\mathbb{E}(T_{\alpha_{m_s}}(B))$. This strategy $\hat{\alpha}_{m_l}$ will be a strategy that always schedules the nodes that α_{m_s} would schedule and some additional nodes. More precise, at any point in time t the strategy $\hat{\alpha}_{m_l}$ will always schedule at least those nodes that α_{m_s} would schedule at time t. We will prove that such a strategy exists in the following by induction over the discrete decision points of $\hat{\alpha}_{m_l}$.

The induction hypothesis is that $\hat{\alpha}_{m_l}$ schedules all nodes that α_{m_s} would schedule no later than α_{m_s}. The beginning is easy, since at $t = 0$ we simply schedule the same nodes as α_{m_s} (and some more).

Lets look the point t, at which under schedule $\hat{\alpha}_{m_l}$ a machine has just finished its task. Because we have always scheduled all nodes scheduled by α_{m_s}, these nodes are either under processing or already finished. Since work has begun at any task no later than under α_{m_s}, no such task can finish later.

We can reconstruct the tree that α_{m_s} would see from consulting the history. (α_{m_s} is known, we know the processing time that each finished task had required, and all tasks that would be finished under α_{m_s} are also finished under $\hat{\alpha}_{m_l}$). From that tree and α_{m_s} we can derive what tasks would be scheduled at the time t. (There is no setup time between the finishing of one machine and the reassignment of that machine to a new task. Hence, there is always a complete set of scheduled tasks and a definite tree.)

If all tasks that α_{m_s} would schedule are scheduled (or can be scheduled at that instance if two decision points are exactly the same), there is no problem and we can just schedule any additional node.

If a task that would be scheduled under α_{m_s} at time t has already been finished, we must take care that by scheduling other nodes we do not "miss" a decision point of α_{m_s}. Suppose the next task that would finish under α_{m_s} is one that is not finished yet but scheduled. Then this task will finish under $\hat{\alpha}_{m_l}$ schedule at least as early as under α_{m_s}. Therefore, for this case the lemma holds. If the next task that would finish under α_{m_s} is a task that has already been finished, we can exactly determine which one of the already finished ones that would be, because we know all processing times of all finished tasks. We can simply assume that the task finishes first and look at the resulting tree and the next set of nodes to be scheduled. We can continue speculating on the upcoming decision points under α_{m_s} until we find a point where all tasks are scheduled but not finished or where we can schedule an additional task.

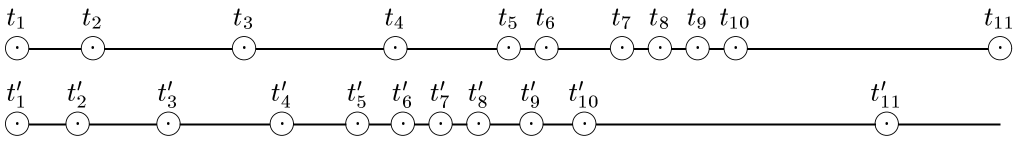


Figure 6.3: Anticipating Decision Points

This way we always stay ahead of α_{m_s}. Since the next decision point (the time until the next task is finished) is not known, we must anticipate all possible decision points that α_{m_s} might reach by finishing nodes that we have already finished (see Figure 6.3, at decision point t'_{10} we must anticipate α_{m_s}'s decisions points t_7, t_8, t_9, and t_{10} because we will not be able to schedule a new machine until t'_{11}).

Because no task is scheduled later with m_l machines than with m_s machines, $\hat{\alpha}_{m_l}$ finishes at least as early as α_{m_s} for any tree and for any instantiation of the random variable for the task processing times. As a result, the total expected processing time cannot be smaller with fewer machines.

$\square$

Hence, for this problem the saying "too many cooks spoil the broth" does not hold.

Chapter 7

Falsification and Evaluation of Scheduling Strategies

7.1 Falsification and Evaluation Criteria

In the following we will take a look at known scheduling strategies and classes of strategies. For a given tree B let $\dot{\alpha}_{OPT}(B)$ be the set of all optimal solutions to $(P3|intree|\mathbb{E}(C_{max}))$ with in-tree constraints B.

A strategy S is **optimal**, if and only if for all trees B $\alpha_S(B) \in \dot{\alpha}_{OPT}(B)$. If a strategy S results in a set $\dot{\alpha}_S(B)$ of solutions, then S is **optimal**, if and only if $\dot{\alpha}_S(B) \subseteq \dot{\alpha}_{OPT}(B)$.

Especially if a strategy will result in a set of solutions, the strategy might be **can-optimal**. This is the case, if $\dot{\alpha}_S(B) \cap \dot{\alpha}_{OPT}(B) \neq \emptyset$ and $\dot{\alpha}_S(B) \setminus \dot{\alpha}_{OPT}(B) \neq \emptyset$ because $\dot{\alpha}_S(B)$ is a set and we could assume that a solution is chosen at random. Hence there is a chance that an optimal solution is drawn (the strategy can be optimal).

If $\dot{\alpha}_S(B) \cap \dot{\alpha}_{OPT}(B) = \emptyset$, the solution is **non-optimal**.

To decide whether a strategy is not optimal, a single counter example suffices. If a counter-example with $\dot{\alpha}_S(B) \not\subseteq \dot{\alpha}_{OPT}(B)$ exists, the strategy S is at most can-optimal.

Figure 7.1 shows an example. For the optimal schedule, the node 3 must be scheduled and two nodes can be picked from the nodes 5, 6, and 7. A can-optimal scheduling strategy is an algorithm that results in multiple possible schedules of which some are optimal and some are not optimal. Figure 7.1 (b) shows such a case where an algorithm's solution can be any three member set from the nodes 3, 5, 6, and 7. If the picked set of scheduled nodes contains node 3, the result is optimal, otherwise it is not optimal. Finally, Figure 7.1 (c) shows the non-optimal case. Any schedule will include node 1, so that no schedule can be optimal. It may often occur that an algorithm only presents a single solution. In this case it is either optimal or non-optimal.

When providing counter examples we will restrict ourselves to the trees with the marked nodes (of the optimal and/or falsified strategy). Displaying the complete DAGs and the points where the falsified strategy loses time would enlarge this work with an unnecessary number of figures. However, we will make the deviations plausible wherever possible (trying to state an intuitive reason, why certain strategies are not optimal). We will also restrict ourselves to showing the smallest existing counter example tree (if possible).

There is no known easier way to choose the optimal strategy that is better than Algorithm 6 yet. When comparing two non-optimal strategies S_1 and S_2 we need to compare the expected processing times reached by applying S_1 and S_2 in comparison to the optimal time. This is not an easy task, because we need to take all trees into account, hence we usually restrict ourself to the falsification of strategies.

Another comparison method is the counting of the number of trees that the strategy fails at. This simply compares all solutions for all trees up to a given size and compares the number of trees that are scheduled

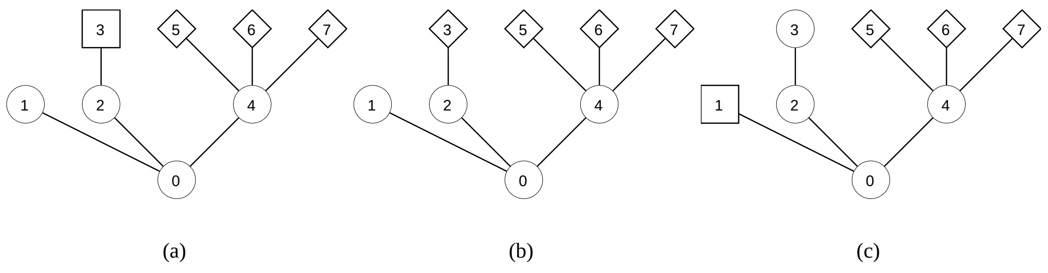


Figure 7.1: Difference Between Optimal (a), Can-Optimal (b), and Non-Optimal (c) (all square nodes must be scheduled, from the diamond nodes enough can be picked at random to get three nodes).

optimally. This quantitative measure also has an impact on the quality of solutions. Any non-optimal schedule for a small tree will be inherited by larger trees of which the smaller is a subtree of.

7.2 Preemptive versus Non-Preemptive Scheduling

We will mainly focus on non-preemptive scheduling because it seems that it is the harder of the two. Unfortunately an optimal preemptive strategy is not necessarily an optimal non-preemptive strategy and vice versa. Figure 7.2 shows the smallest trees, where the optimal preemptive strategy is only can-optimal (a) or non-optimal (b) to the non-preemptive schedule. The intuitive reason in both examples is that it is bad to be left with a high tree with only two nodes left. In (a) removing any leaf leads to the same subtree, but the non-preemptive strategy must already take into account, that the majority of scheduled leaves should then be on the smaller of both sides (that reduces the probability to end up with a tree, with two leaves in two 2-node branches). Basically the same holds for (b), but it is far less easy to see. If one of the leaves $\{3, 4, 5\}$ is finished, both strategies prefer scheduling the remaining two. On the other hand the tree, with one of the leaves $\{8, 10\}$ removed has a slightly lower expected value, so there is a trade-off between being ready for the first case or working towards the second case. For the preemptive schedule this decision needs not be made.

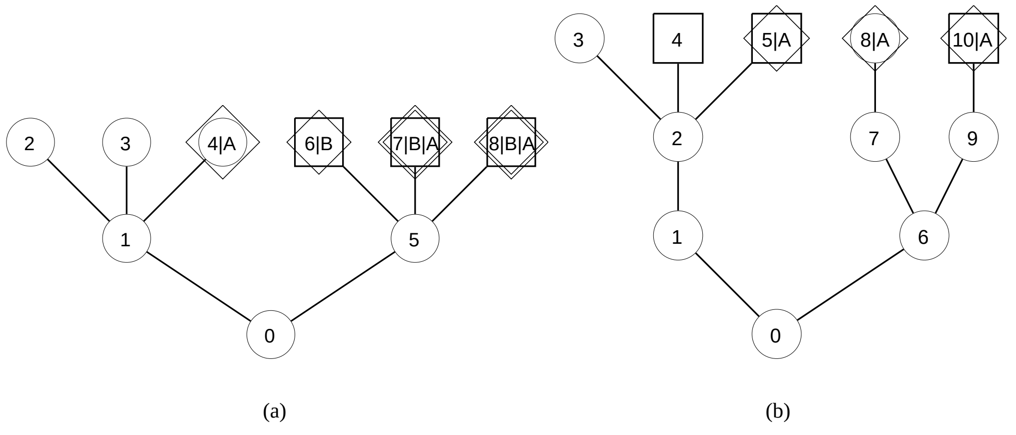


Figure 7.2: Preemptive Schedules in Relation to Non-Preemptive Schedules. (Squares show the non-preemptive optimal solutions. Diamonds and upper case letters show the different optimal preemptive solutions.)

Figure 7.2 (b) is also the smallest tree, where the optimal non-preemptive solution is non-optimal to the optimal preemptive schedule. For all smaller trees the optimal non-preemptive schedule is also optimal as preemptive schedule.

7.3 The Highest Level First Strategy (HLF)

The **highest level first** (HLF) strategy is optimal for the case with two machines in parallel ($P2|intree|\mathbb{E}(C_{max})$), for the preemptive as well as for the non-preemptive case. This result is due to Chandy and Reynolds [CR75] (the proof is also found in [Pin95]). In [Pin95] various scheduling problems with deterministic input variables are shown to be optimally solved by the HLF rule (which Pinedo calls critical path (CP) rule).

For our problem with three processors [CR75] already contains a counter example tree with 12 nodes where HLF fails. The smallest two counter examples (trees with 11 nodes) showing that HLF is non-optimal are given in Figure 7.3 (a), (b). Figure 7.3 (c) also gives the smallest tree that shows that HLF is at most can-optimal.

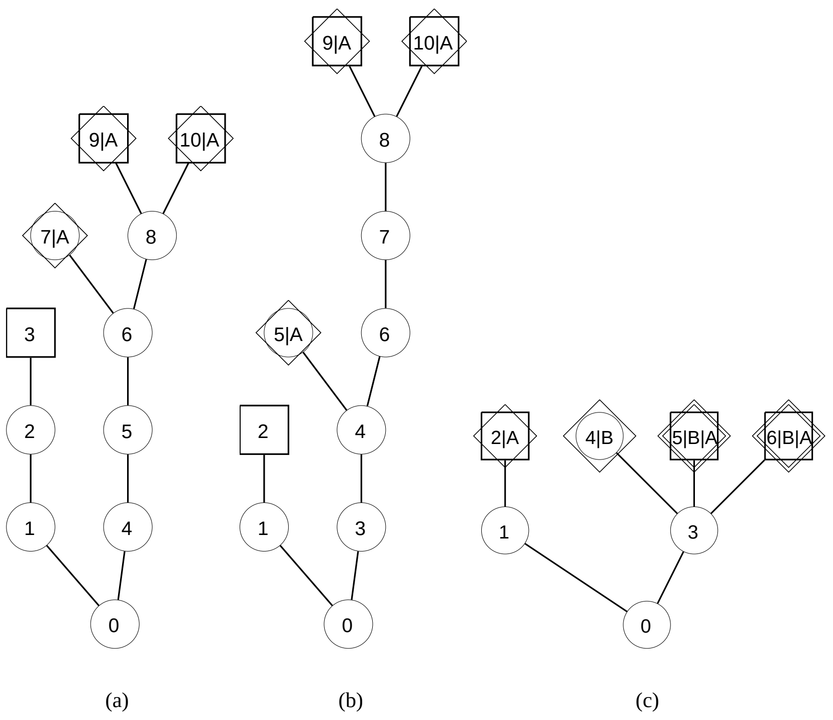


Figure 7.3: HLF is Non-Optimal (Squares show the non-preemptive optimal solutions. Diamonds and upper case letters show the HLF solutions.)

Papadimitriou and Tsitsiklis have shown that HLF is asymptotically near optimal in the sense that the expected processing time of an HLF strategy $\mathbb{E}(T_{HLF})$ divided by the expected processing time of an optimal schedule $\mathbb{E}(T_{OPT})$ approaches 1 very quickly as the tree size grows (see [PT87]), precisely:

Theorem 7.1 (Relative Optimality of HLF). *For any in-tree B of size n and an arbitrary number of processors m, let $\mathbb{E}(T_{HLF}(B))$ be the expected processing time under an HLF scheduling strategy and let $\mathbb{E}(T_\pi(B))$ be the expected processing time under any strategy π. There exists some function*

$\beta : \{1, 2, \ldots\} \to [0, \infty)$ *such that* $\lim_{n\to\infty} \beta(n) = 0$ *and*

$$\mathbb{E}(T_{HLF}(B)) \leq \inf_{\pi} \mathbb{E}(T_\pi(B))(1 + \beta(n))$$

Proof. See [PT87]. □

This is a rather strong result. On the one hand we know that HLF is non-optimal and very often only can-optimal, on the other we know that it is close to optimal.

Because often there may be multiple nodes at the highest level, one can distinguish between HLF schedules that provide a tie-breaking method and ones that break ties at random (Chandy and Reynolds call these A-Schedules if a labeling scheme is used and B-Schedules if random decisions are made).

Providing a tie-breaking method still keeps close to the proven optimal property, while being able to improve the algorithm in a lot of schedules. Figure 7.4 shows some variants where HLF was extended to a lexicographical order with HLF in the first component and various other weights in the second component. The next section will show that – while improving HLF – no such approach will ever be able to yield an optimal algorithm. (Note, that for the algorithm in the last line of Figure 7.4, the parent's or an ancestor's in-degree is second in order, if that node is the root of a pod. A pod is a subtree where all leaves are at the same height and the in-degree of the root is equal to the number of leaves. E.g., node 3 of Figure 7.2 is the root of a pod.)

Algorithm	Failures on Trees With k Nodes							
(or Question)	7	8	9	10	11	12	13	14
Q: Number of trees	48	115	286	719	1842	4766	12486	32973
Q: Trees with more than 4 leaves	20	67	207	595	1655	4494	n/a	n/a
HLF without tie-breaking	1	8	33	116	372	1130	3352	9613
Lowest in-degree of leaf-parent		1	6	25	90	288	913	2846
Parent's subtree weight			2	8	36	123	453	1577
Reverse DFS number of leaves			1	6	30	110	n/a	n/a
Lexicographical order of parent's or ancestor's in-degree (if pod), parent's in-degree, and parent's subtree weight					11	58	250	976

Figure 7.4: HLF-Based Algorithms with Different Lexicographical Orders

7.4 Static List Scheduling Strategies

A static list scheduling strategy for $(P3|intree|\mathbb{E}(C_{max}))$ would order all nodes of the tree in a static list. At the beginning the highest available nodes are scheduled (a node is available if it is a leaf or its ancestors have already been processed). Every time a machine is freed, the highest available unscheduled node is assigned to that machine.

Lemma 7.1 (Static List Policies Fail for $(P3|intree|\mathbb{E}(C_{max}))$**).** *No static list scheduling strategy can be optimal for* $(P3|intree|\mathbb{E}(C_{max}))$.

Proof. Have a look at Figure 7.5. Initially node 4 and two of the nodes $\sigma = \{6, 7, 8\}$ need to be scheduled. Depending on the first node that is finished, different nodes need to be scheduled:

Case 1 If a node from σ is finished first, then the remaining unscheduled node of σ is scheduled, resulting in a scheduling order of $4, \sigma_1, \sigma_2, \sigma_3, \ldots$.

Case 2 If 4 is finished first, one of the nodes $\gamma = \{2, 3\}$ needs to be scheduled. If the newly scheduled node is finished next, the other node from γ needs to be scheduled.

Disregarding the nodes 0, 1, and 5 the following static list schedules can represent the first case:

$$\begin{array}{c} 3, 4, \sigma_1, \sigma_2, \sigma_3, 2 \\ 4, 3, \sigma_1, \sigma_2, \sigma_3, 2 \\ 4, \sigma_1, 3, \sigma_2, \sigma_3, 2 \\ 4, \sigma_1, \sigma_2, 3, \sigma_3, 2 \\ 4, \sigma_1, \sigma_2, \sigma_3, 3, 2 \end{array}$$

The following static list schedules can represent the second case:

$$\begin{array}{c} 4, \sigma_1, \sigma_2, 2, 3, \sigma_3 \\ 4, \sigma_1, \sigma_2, 3, 2, \sigma_3 \\ 4, \sigma_1, 3, \sigma_2, 2, \sigma_3 \\ 4, 3, \sigma_1, \sigma_2, 2, \sigma_3 \\ 3, 4, \sigma_1, \sigma_2, 2, \sigma_3 \end{array}$$

The lists for these cases contradict each other because in the first case at most one node from γ (the one constraint by 4) can be in the list before the last node from σ, while the second case both nodes from γ must appear in the list before the last node from σ. Therefore, no optimal static list schedule for this instance of $(P3|intree|\mathbb{E}(C_{max}))$ exists. $\square$

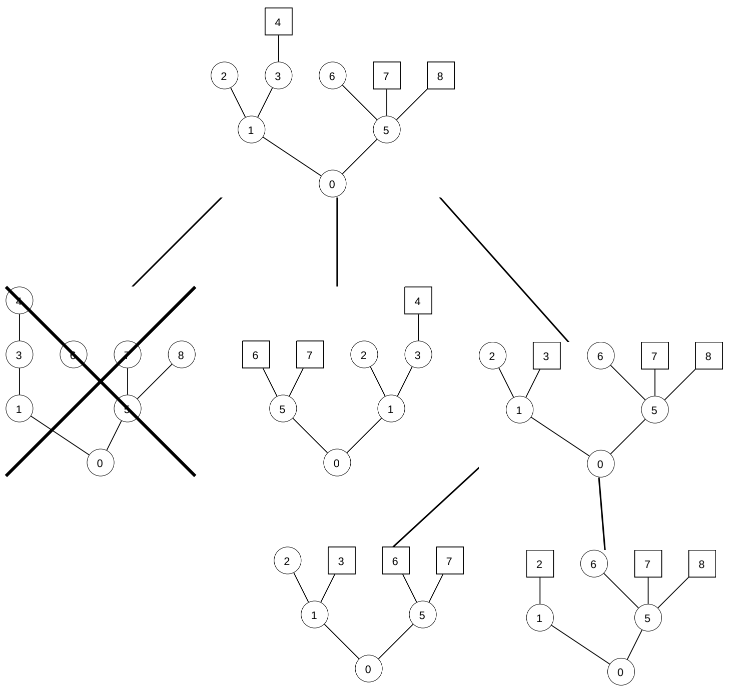


Figure 7.5: An Optimal Non-Preemptive Strategy Cannot be a Static List Schedule. (The complete highest two levels and part of third level of the subtree DAG are shown, squares are scheduled nodes for an optimal schedule and the crossed out tree is never reached under an optimal schedule.)

If we relax the definition of a static list schedule a bit and adapt to the problem at hand we can define **semi-static list schedules**:

Definition 7.1 (Semi-Static List Policy). *A **semi-static list scheduling strategy** is a policy where for any given tree or subtree all tasks are ordered into a list, depending only on the structure of the tree or subtree. Any free machine is assigned to the highest available, unscheduled task.*

(This corresponds to deciding solely by the tree structure).

Obviously there is a semi-static list policy that is optimal for the preemptive problem (just put the three optimal nodes at the top).

This is not the case for the non-preemptive problem:

Lemma 7.2 (Semi-Static List Policies Fail for $(P3|intree|\mathbb{E}(C_{max}))$). *No semi-static list scheduling strategy can be optimal for $(P3|intree|\mathbb{E}(C_{max}))$.*

Proof. Have a look at Figure 7.6. A semi-static list schedule assigns exactly one list to one tree. Let the tree shown as (b) and as the rightmost subtree of the second level of the subtree DAG shown as (a) be denoted as B (the unique representation is $2, 3, 0, 0, 1, 0, 4, 0, 0, 0, 0$).

From (b) the optimal static list for tree B starts with 4 and at least two nodes from $\{7, 8, 9, 10\}$. From (a) the static list for tree B starts with node 4 and nodes 2 and 3. This contradicts the existence of a single list for tree B.

Therefore no semi-static list scheduling strategy can be optimal for $(P3|intree|\mathbb{E}(C_{max}))$. □

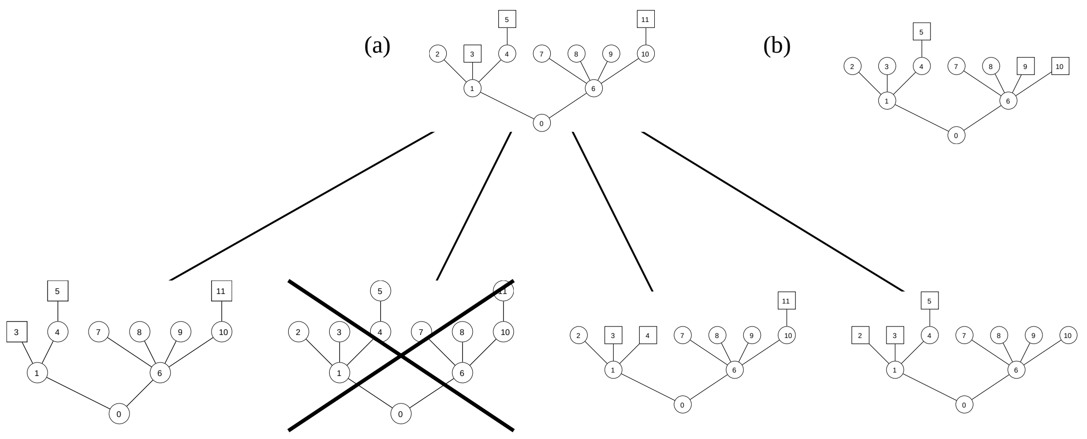


Figure 7.6: An Optimal Non-Preemptive Strategy Cannot be a Semi-Static List Schedule. (The highest two levels of the subtree DAG are shown in (a), squares are scheduled nodes for an optimal schedule, and the crossed out tree is never reached under an optimal schedule. The optimal schedule for the rightmost subtree of second level in (a) when scheduled by itself is shown in (b).)

As a result we can exclude a large number of strategies that depend solely on the tree structure (including HLF and variants). Any optimal strategy must take already scheduled tasks into account. This can be either accomplished by looking at sets of three nodes or by scheduling nodes on a given tree one by one and including information about the previous scheduled nodes. Surely there could be an algorithm that – given a tree – returns three leaves that correspond to the optimal initial assignment, but this algorithm would be of little help in subsequent steps.

7.5 Further Scheduling Strategies

A **Monte Carlo** algorithm is an algorithm that has a bounded probability of giving the correct answer, but it can also give an incorrect answer (Las Vegas algorithms never give a wrong answer, they will answer ”‘don’t know”’ instead). Iterating a Monte Carlo algorithm a number of times will result in reducing the probability of giving a wrong answer. A good Monte Carlo Algorithm makes a problem quite tractable (e.g. prime number testing – for which actually a Las Vegas algorithm exists). Since the problem at hand is stochastic in nature it seems interesting to try a Monte Carlo-like method. Such a method would basically work as follows. We choose a schedule and randomly execute it. The execution simulates the scheduling by randomly choosing a leaf which finishes first in each step. In this way, one execution takes time $\mathcal{O}(n)$. For each step either one third, one half, or one is added, corresponding to the expected length of the step with three, two, or one machine working at the same time. If this is iterated a large number of times the average of the sums should reflect the real expected value of the simulated schedule.

Unfortunately, this is not possible because we do not know how to continue after the first leaf of the initial schedule has ”‘finished”’. It is obviously not possible to check all possibilities again – this would result in the algorithm from section 6.1. Two rules seem plausible at that point: choose a random leaf to schedule or choose a random leaf at a highest level to schedule (because HLF is at least asymptotically optimal). Both variants have been tested a number of times and the results are presented in Figure 7.7. It may seem that 100 iterations per schedule is not a lot, but the number of schedules to test grows cubicly with the number of leaves.

Algorithm	Failures on Trees With k Nodes								
(or Question)	4	5	6	7	8	9	10	11	12
Q: Number of trees	4	9	20	48	115	286	719	1842	4766
Q: Trees with more than 4 leaves		1	5	20	67	207	595	1655	4494
HLF				1	8	33	116	372	1130
Monte Carlo 100 (HLF) 1. run				1	9	32	156	514	n/a
Monte Carlo 100 (HLF) 2. run				1	6	43	155	526	n/a
Monte Carlo 100 (HLF) 3. run					3	31	144	527	n/a
Monte Carlo 100 (random) 1. run				1	4	25	117	406	n/a
Monte Carlo 100 (random) 2. run				1	8	31	144	438	n/a
Monte Carlo 100 (random) 3. run					8	37	139	421	n/a

Figure 7.7: Comparison of Monte Carlo Algorithms to HLF

The method does not seem very stable either. Figure 7.8 shows a tree for which the above described Monte Carlo method results in very unpredictable results. Although the algorithm seldom produces a very bad result (e.g. schedules node 7 of the tree in Figure 7.8), it varies strongly between the other three possibilities (scheduling three leaves from $\{3, 4, 5\}$, two leaves from $\{3, 4, 5\}$ and one from $\{9, 10\}$, or one leaf from $\{3, 4, 5\}$ and two from $\{9, 10\}$ - see Figure 7.9).

The results shown in Figure 7.9 suggest that a larger number of iterations might result in a near optimal algorithm. Does a number exist for which the Monte Carlo algorithm hardly ever fails? We will see later in section 10.1 that the differences between two schedules may become very small, possibly as small as 3^{-n-2} where n is the size of the tree. If this is indeed the case, then the following reasoning suggests that no smallest sufficient number of iterations for the Monte Carlo method exists. For each compared schedule a number of l iterations is made. For each iteration a sum of the terms 1, $1/2$, and $1/3$ is calculated and from the total the arithmetic mean is calculated by summing all sub-sums and dividing by the number of iterations. The sum of all sub-sums is itself a sum of the terms 1, $1/2$, and $1/3$. When comparing two different schedules, the Monte Carlo algorithm compares these sums. The smallest possible difference between the two sums is $1/6$ (if same terms in both sums are coupled, the terms $1/2$ and $1/3$ remain). Both sums should reflect the expected value for their corresponding schedule. If the difference is 3^{-n-2}, then $1/6$ divided by l should be well below that difference. Otherwise the results do not seem to be exact enough

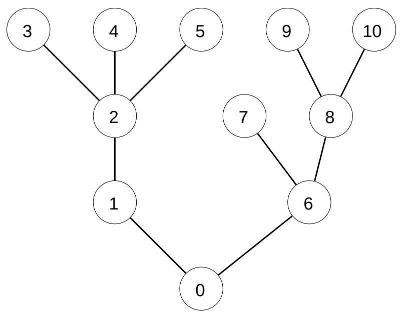


Figure 7.8: Example Tree for Instability of the Monte Carlo Method. (The optimal schedule is $\{3, 4, 5\}$ with an expected processing time of $35261/5832$.)

	Probability of random	Percentage with given number of iterations			
Schedule	HLF	100	1000	10000	100000
Three leaves from $\{3, 4, 5\}$ (optimal)	10%	10%	15%	50%	90%
Two leaves from $\{3, 4, 5\}$ and one from $\{9, 10\}$	60%	70%	50%	40%	10%
One leaf from $\{3, 4, 5\}$ and two from $\{9, 10\}$	30%	20%	35%	10%	0%

Figure 7.9: Stability of Monte Carlo for Tree in Figure 7.8 and 20 Runs

to distinguish both schedules with a high enough probability. Hence, l needs to be $Oof3^{n}$ which results in the Monte Carlo algorithm being as intractable as the dynamic programming algorithm. The higher the chosen number of iterations, the better the algorithm seems to be able to distinguish schedules with smaller differences. On the other hand, the results suggest that Monte Carlo does not perform better than HLF and is of higher complexity.

Chapter 8

Taking a Closer Look at Two-Leaves Subtrees

8.1 Yet Another Way to Calculate the Expected Processing Time

From 5.1 and Corollary 5.1 we know two ways to calculate the expected processing time for a strategy α. Both of these are based on the probabilities of execution paths (an execution path being the order in which nodes are finished, a permutation over all nodes) under strategy α. This corresponds to using Algorithm 1 for calculating the expected processing time. We will now find another way to calculate the expected processing time.

Let B be a tree and D be the subtree DAG for B and let B_0 be the subtree corresponding to only the root of B. Suppose we could calculate the probability of reaching any given subtree B' under strategy α. If we can find an anti-chain of "independent" subtrees in the subtree DAG, whose expected processing times are easily calculated, then we would be able to give the total expected processing time. An anti-chain C is a set of trees that have the following properties:

1. No tree in the chain is reachable (in D) from any other tree in the chain.
2. Every execution path (a path from B to B_0) includes exactly one element from the chain C.

To be able to calculate the total expected processing time $\mathbb{E}(T_\alpha(B))$, all trees in any path from B to a node in C should have at least three leaves.

Theorem 8.1 (Expected Processing Time by Subtree Weights). *Let B ($n = |B|$) be a tree, D its subtree DAG, and C an anti-chain with the above stated property that $\forall p \in paths_D(B, B_0)$: let $B_p = p \cap C \wedge$ let $p' = path(B, B_p) :\Rightarrow \forall B' \in p' : \big(leaves(B') \geq 3 \vee B' = B_p\big)$.*

The total expected processing time is then

$$\mathbb{E}(T_\alpha(B)) = \sum_{B' \in C} P_\alpha[B'] \cdot \left(\mathbb{E}(T_\alpha(B')) + \frac{1}{3}(n - |B'|)\right)$$

Proof. Let the nodes of B be $V = \{1, \ldots, n\}$. For any subtree B' of B, let the nodes of B' be $V' = \{i_1, \ldots, i_{|B'|}\}$.

The probability of reaching B' is the sum over all execution paths p with $\forall i \in V' : p(i) > n - |B'|$ (that is B' occurs during p).

$$P_\alpha[B'] = \sum_{p \in CP(B) \wedge \forall i \in V' : p(i) > n - |B'|} P_\alpha^B[p]$$

The probability that an execution path p of B ends with an execution path p' of B' is:

$$P[p' \sqsupset_{\mathit{suff}} p] = P_\alpha[B'] \cdot P_\alpha^{B'}[p'] = \sum_{p \in CP(B) \wedge p' \sqsupset_{\mathit{suff}} p} P_\alpha^B[p]$$

The expected total processing time for B' is (by Theorem 5.1):

$$\mathbb{E}(T_\alpha(B')) = \sum_{p' \in CP(B')} P_\alpha^{B'}[p'] \cdot \left(s_2(p') \cdot \frac{1}{3} + (s_1(p') - s_2(p')) \cdot \frac{1}{2} + (|B'| - s_1(p')) \right)$$

If p ends with p' we will call p' a suffix of p denoted by $p' \sqsupset_{\mathit{suff}} p$ or $p = \mathit{suffix}_{|B'|}(p)$ (suffix of the length of $|B|$). Since before B' is reached there are always at least three leaves left, if $p \in CP(B)$ and $p' \in CP(B')$ where p' is a suffix of p, then

$$\forall i \in V' : p(i) = n - |B'| + p'(i) : s_2(p) = n - |B'| + s_2(p') \text{ and } s_1(p) = n - |B'| + s_1(p')$$

Because C is an anti-chain the following holds true:

$$\begin{aligned} &\forall p \in CP(B) : \exists B' \in C : \\ &\qquad \left(\left(\forall i \in V' : p(i) > n - |B'| \right) \wedge \left(\nexists B'' \in C \setminus \{B'\} : \left(\forall i \in V'' : p(i) > n - |B''| \right) \right) \right) \end{aligned}$$

The total execution time is therefore

$$\begin{aligned} & \mathbb{E}(T_\alpha(B)) \\ = & \sum_{p \in CP(B)} P_\alpha^B[p] \cdot \left(s_2(p) \cdot \frac{1}{3} + (s_1(p) - s_2(p)) \cdot \frac{1}{2} + (n - s_1(p)) \right) \\ = & \sum_{B' \in C} \left(\sum_{p \in CP(B) \wedge \forall i \in V' : p(i) > n - |B'|} P_\alpha^B[p] \cdot \left(s_2(p) \cdot \frac{1}{3} + (s_1(p) - s_2(p)) \cdot \frac{1}{2} + (n - s_1(p)) \right) \right) \\ = & \sum_{B' \in C} \left(\sum_{p \in CP(B) \wedge \forall i \in V' : p(i) > n - |B'|} P_\alpha^B[p] \cdot \left(\left(n - |B'| + s_2(\mathit{suffix}_{|B'|}(p)) \right) \cdot \frac{1}{3} \right. \right. \\ & \quad + \left((n - |B'| + s_1(\mathit{suffix}_{|B'|}(p))) - (n - |B'| + s_2(\mathit{suffix}_{|B'|}(p))) \right) \cdot \frac{1}{2} \\ & \quad \left. \left. + \left(n - (n - |B'| + s_1(\mathit{suffix}_{|B'|}(p))) \right) \right) \right) \\ = & \sum_{B' \in C} \left(\sum_{p \in CP(B) \wedge \forall i \in V' : p(i) > n - |B'|} P_\alpha^B[p] \cdot \left(\frac{n - |B'|}{3} + s_2(\mathit{suffix}_{|B'|}(p)) \cdot \frac{1}{3} \right. \right. \\ & \quad \left. \left. + \left(s_1(\mathit{suffix}_{|B'|}(p)) - s_2(\mathit{suffix}_{|B'|}(p)) \right) \cdot \frac{1}{2} + \left(|B'| - s_1(\mathit{suffix}_{|B'|}(p)) \right) \right) \right) \end{aligned}$$

$$
\begin{aligned}
= & \sum_{B'\in C} \Bigg(\frac{n-|B'|}{3} \Bigg(\sum_{p\in CP(B)\wedge \forall i\in V' : p(i)>n-|B'|} P_\alpha^B[p] \Bigg) \\
& \quad + \sum_{p'\in CP(B')} \Bigg(\sum_{p\in CP(B)\wedge p' \sqsupset_{suff} p} P_\alpha^B[p] \cdot \Big(s_2(\mathit{suffix}_{|B'|}(p)) \cdot \frac{1}{3} \\
& \quad +\big(s_1(\mathit{suffix}_{|B'|}(p)) - s_2(\mathit{suffix}_{|B'|}(p))\big) \cdot \frac{1}{2} + \big(|B'| - s_1(\mathit{suffix}_{|B'|}(p))\big)\Big)\Bigg)\Bigg) \\
= & \sum_{B'\in C} \Bigg(\frac{n-|B'|}{3} P_\alpha[B'] \\
& \quad + \sum_{p'\in CP(B')} \Bigg(\sum_{p\in CP(B)\wedge p' \sqsupset_{suff} p} P_\alpha^B[p] \cdot \Big(s_2(p') \cdot \frac{1}{3} \\
& \quad +\big(s_1(p') - s_2(p')\big) \cdot \frac{1}{2} + \big(|B'| - s_1(p')\big)\Big)\Bigg)\Bigg) \\
= & \sum_{B'\in C} \Bigg(\frac{n-|B'|}{3} P_\alpha[B'] \\
& \quad + \sum_{p'\in CP(B')} \Bigg(s_2(p') \cdot \frac{1}{3} + \big(s_1(p') - s_2(p') \cdot \frac{1}{2}\big) + \big(|B'| - s_1(p')\big)\Bigg) \\
& \qquad \cdot \Bigg(\sum_{p\in CP(B)\wedge p' \sqsupset_{suff} p} P_\alpha^B[p] \Bigg)\Bigg) \\
= & \sum_{B'\in C} \Bigg(\frac{n-|B'|}{3} P_\alpha[B'] \\
& \quad + P_\alpha[B'] \cdot \sum_{p'\in CP(B')} \Bigg(s_2(p') \cdot \frac{1}{3} + \big(s_1(p') - s_2(p')\big) \cdot \frac{1}{2} + \big(|B'| - s_1(p')\big)\Bigg) \cdot P_\alpha^{B'}[p']\Bigg) \\
= & \sum_{B'\in C} P_\alpha[B'] \cdot \Bigg(\mathbb{E}(T_\alpha(B') + \frac{1}{3}(n-|B'|) \Bigg)
\end{aligned}
$$

$\square$

This result could lead to an algorithm, maximizing the probability of reaching cheap subtrees, provided

- an anti-chain C can be found with the stated property,
- the expected processing times of the elements of the anti-chain can be calculated, and
- the chain is small enough to be evaluated efficiently.

Obviously any level of the subtree DAG (all nodes representing subtrees with the same number of nodes) is an anti-chain. Choosing a high enough level will satisfy the stated property. Unfortunately the number of nodes in the level can be large and the problem of calculating the expected processing times of the chain elements is only a little easier than the original problem. Finally a way of optimizing the probabilities that a cheap tree is met needs to be found.

Figure 8.1 shows the DAG for the tree in Figure 5.3 with the probabilities of the subtrees shown by their shade of gray (the DAG is the same as the one in Figure 5.4 on page 19).

In the rest of this chapter we will try to find a cure for some of this points.

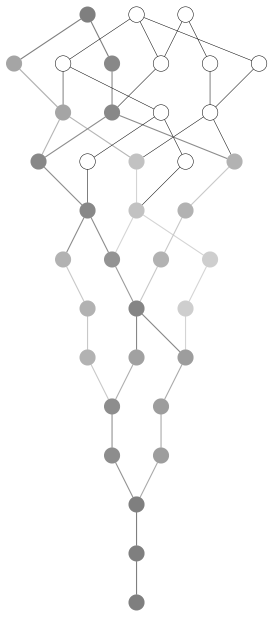


Figure 8.1: Probabilities in Subtree DAG for Example Tree in Figure 5.3 for the Optimal Schedule (subtrees represented by empty nodes are not reached at all, the shade of gray of a node represents the probability from light low to dark high).

8.2 The Optimal Expected Processing Time for Two-Leaves-Trees

In the following we will show how the optimal expected processing time for two-leaves-trees can be found. Note, that a strategy has no choice, once there are only two leaves left, and hence the optimal expected processing time for any strategy and for the optimal strategy is the same for any two-leaves-tree.

We will describe all trees with two leaves by three numbers $(a|k|l)$:

a - The number of nodes from the root to the first node where the tree branches (excluding that node).

k - The number of nodes in the left branch.

l - The number of nodes in the right branch.

Obviously, for any two-leaves-tree B, $a + k + l + 1 = |B|$. See Figure 8.2 for an example of this classification.

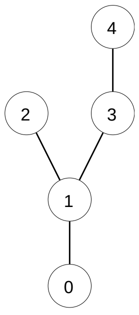


Figure 8.2: Two-Leaves-Tree Example for $(1|1|2)$.

For the calculation of the expected processing time of B we will first have a look at trees with $a = 0$. At every step a machine from one of the branches finishes and the branch is decreased. Either branch is equally likely, so both possibilities can be weighted with $1/2$. A machine is expected to finish after $1/2$ of the expected processing time for one task because two machines are working in parallel. If there is only one branch left, the remaining branch and the branching node will be processed with a single machine. The expected processing time q can thus be described by the following recursion:

$$q(k,l) = \begin{cases} k+1 & \text{if } l = 0, \\ l+1 & \text{if } k = 0, \\ \big(1 + q(k-1,l) + q(k,l-1)\big)/2 & \text{otherwise.} \end{cases}$$

If $a \geq 0$, the nodes below the branching node are also processed with a single machine, hence the total processing time of a two-leaves-tree $(a|k|l)$ is

$$p(a,k,l) = q(k,l) + a$$

Observation 8.1 (Expected Processing Time of Two-Leaves-Tree). *If B is a two-leaves-tree with two branches of the size k and l and the height of the branching node a, its expected processing time is $p(a,k,l)$*

Algorithm 10 $\mathbf{p}(B)$

```
Require: $leaves(B) = 2$
1:  proc  $p(B)$ :
2:  $l_1 \longleftarrow$ first leaf of $B$
3:  $l_2 \longleftarrow$ second leaf of $B$
4:  $n \longleftarrow lca(l_1, l_2)$
5:  $a \longleftarrow height(n)$
6:  $k \longleftarrow height(l_1) - a$
7:  $l \longleftarrow height(l_2) - a$
8:  for $i$ from 1 to $k$ do
9:      $t(0,i) \longleftarrow i + 1$
10: end for
11: for $i$ from 1 to $k$ do
12:     $t(i,0) \longleftarrow i + 1$
13: end for
14: for $i$ from 1 to $k$ do
15:     for $j$ from 1 to $l$ do
16:         $t(i,j) \longleftarrow \big(t(i-1,j) + t(i,j-1) + 1\big)/2$
17:     end for
18: end for
19: return(t(k,l) + a)
```

Lemma 8.1 (Calculating the Total Expected Processing Time of Two-Leaves-Trees). *Algorithm 10 calculates the total expected processing time of a two-leaves-tree B in $\mathcal{O}(|B|^2)$.*

Proof. Algorithm 10 is a simple dynamic programming version of the recursive definition of $q(a,k,l)$. Its running time is dominated by the loops in lines 14 and 15, hence its running time is $\mathcal{O}(k \cdot l) = \mathcal{O}(|B|^2)$. □

A simple consequence is the following corollary.

Corollary 8.1. *For a given tree B the total expected processing times of all two-leaves-tree subtrees B' can be calculated in $\mathcal{O}(|B|^2)$.*

A non-recursive version of the equation for $p(a,k,l)$ can be derived, looking at the dynamic programming tableau of Algorithm 10 and summing the values added in each entry separately (the initialization parts with $l = 0$ or $k = 0$ and the $+1/2$-part). The following theorem proves the correctness of the derived formula.

(Using dynamic programming to calculate all values for the binomial coefficients, the non-recursive form should also be evaluatable in time $\mathcal{O}(|B|^2)$).

Theorem 8.2 (Total Expected Processing Time of Two-Leaves-Trees). *Let*

$$\begin{aligned} s(a,k,l) &= \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{l+i-1} \cdot \binom{l+i-2}{i-1} \cdot (k-i+2) \\ &+ \sum_{j=1}^{l} \left(\frac{1}{2}\right)^{k+j-1} \cdot \binom{k+j-2}{j-1} \cdot (l-j+2) \\ &+ \sum_{i=1}^{k} \sum_{j=1}^{l} \left(\frac{1}{2}\right)^{k-i+l-j+1} \cdot \binom{k-i+l-j}{l-j} \\ &+ a \end{aligned}$$

If $l > 0 \wedge k > 0$, then $s(a,k,l) = p(a,k,l)$.

We need the following lemmas to establish the proof:

Lemma 8.2.

$$S_n = \sum_{i=0}^{n} \left(\frac{1}{2}\right)^{i} \cdot (n-i) = \left(\frac{1}{2}\right)^{n-1} + 2n - 2$$

Proof. (By Induction on n)

For $n = 0$, left hand side:

$$lhs = \left(\frac{1}{2}\right)^{0} \cdot 0$$

For $n = 0$, right hand side:

$$rhs = \left(\frac{1}{2}\right)^{0-1} + 2 \cdot 0 - 2 = 0$$

From n to $n+1$:

$$\begin{aligned} S_{n+1} &= \sum_{i=0}^{n+1} \left(\frac{1}{2}\right)^{i} \cdot (n+1-i) \\ &= n+1+\sum_{i=1}^{n+1} \left(\frac{1}{2}\right)^{i} \cdot (n+1-i) \\ &= n+1+\frac{1}{2} \cdot \sum_{i=1}^{n+1} \left(\frac{1}{2}\right)^{i-1} \cdot (n-(i-1)) \\ &= n+1+\frac{1}{2} \cdot \sum_{i=0}^{n} \left(\frac{1}{2}\right)^{i} \cdot (n-i)) \\ &= n+1+\frac{1}{2} \cdot S_n \\ &= n+1+\frac{1}{2} \cdot \left(\left(\frac{1}{2}\right)^{n-1} + 2n - 2\right) \\ &= n+1+\left(\frac{1}{2}\right)^{n} + n - 1 \\ &= \left(\frac{1}{2}\right)^{(n+1)-1} 2 \cdot (n+1) - 2 \end{aligned}$$

□

Lemma 8.3. *If $k > 0$m then*

$$\binom{r-1}{k} + \binom{r-1}{k-1} = \binom{r}{k}$$

Proof. (See [GKP94], p. 159). □

We are now ready to prove the theorem.

Proof of Theorem 8.2. (By Induction)
Let $s'(k,l) = s(a,k,l) - a$, we will then only need to prove $q(k,l) = s'(k,l)$.
For $l = 1$, right hand side:

$$\begin{aligned}
s'(k,1) &= \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{1+i-1} \cdot \binom{1+i-2}{i-1} \cdot (k-i+2) \\
&+ \sum_{j=1}^{1} \left(\frac{1}{2}\right)^{k+j-1} \cdot \binom{k+j-2}{j-1} \cdot (1-j+2) \\
&+ \sum_{i=1}^{k} \sum_{j=1}^{1} \left(\frac{1}{2}\right)^{k-i+1-j+1} \cdot \binom{k-i+1-j}{1-j} \\
&= \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{i} \cdot \binom{i-1}{i-1} \cdot (k-i+2) \\
&+ \left(\frac{1}{2}\right)^{k} \cdot \binom{k+1-2}{1-1} \cdot (1-1+2) \\
&+ \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{k-i+1} \cdot \binom{k-i+1-1}{1-1} \\
&= \left(\sum_{i=1}^{k} \left(\frac{1}{2}\right)^{i} \cdot 1 \cdot (k-i+2)\right) + \left(\frac{1}{2}\right)^{k} \cdot \binom{k-1}{0} \cdot 2 + \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{k-(i-1)} \cdot \binom{k-i}{0} \\
&= \left(\sum_{i=1}^{k} \left(\frac{1}{2}\right)^{i} \cdot (k-i+2)\right) + \left(\frac{1}{2}\right)^{k} \cdot 2 + \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{i} \\
&= \left(\frac{1}{2} \cdot \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{i-1} \cdot \big((k+1)-(i-1)\big)\right) + \left(\frac{1}{2}\right)^{k} \cdot 2 + 1 - \left(\frac{1}{2}\right)^{k} \\
&= \left(\frac{1}{2} \cdot \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^{i} \cdot \big((k+1)-i\big)\right) + \left(\frac{1}{2}\right)^{k} + 1 \\
&= \left(\frac{1}{2} \cdot \sum_{i=0}^{k+1} \left(\frac{1}{2}\right)^{i} \cdot \big((k+1)-i\big)\right) - \left(\frac{1}{2}\right)^{k+1} \cdot 1 - \left(\frac{1}{2}\right)^{k+2} \cdot 0 + \left(\frac{1}{2}\right)^{k} + 1 \\
&= \frac{1}{2} \cdot \left(\left(\frac{1}{2}\right)^{k} + 2k\right) + \left(\frac{1}{2}\right)^{k+1} + 1 \\
&= \left(\frac{1}{2}\right)^{k} + k + 1
\end{aligned}$$

For $l = 1$, left hand side:

$$\begin{aligned}
q(k,1) &= (q(k-1,1)+k+1+1)/2 \\
&= ((q(k-2,1)+k+1)/2+k+2)/2 \\
&= (((q(k-3,1)+k-1+1)/2+k+1)/2+k+2)/2 \\
&\vdots \\
&= \underbrace{((\ldots(\ldots(}_{k\text{-times}}q(0,1)\underbrace{+3)/2\ldots k+2-j+1)/2\ldots+k+1)/2+k+2)/2}_{k\text{-times}} \\
&= \left(\frac{1}{2}\right)^{k-1}+\sum_{i=1}^{k}(k+3-i)\cdot\left(\frac{1}{2}\right)^{i} \\
&= \left(\frac{1}{2}\right)^{k-1}+\frac{1}{2}\cdot\sum_{i=1}^{k}\big(k+2-(i-1)\big)\cdot\left(\frac{1}{2}\right)^{i-1} \\
&= \left(\frac{1}{2}\right)^{k-1}+\frac{1}{2}\cdot\sum_{i=0}^{k-1}(k+2-i)\cdot\left(\frac{1}{2}\right)^{i} \\
&= \left(\frac{1}{2}\right)^{k-1}+\frac{1}{2}\cdot\left(-\left(\frac{1}{2}\right)^{k}\cdot 2-\left(\frac{1}{2}\right)^{k+1}\cdot 1-\left(\frac{1}{2}\right)^{k+2}\cdot 0+\sum_{i=0}^{k+2}(k+2-i)\cdot\left(\frac{1}{2}\right)^{i}\right) \\
&= \left(\frac{1}{2}\right)^{k-1}-\left(\frac{1}{2}\right)^{k}-\left(\frac{1}{2}\right)^{k+2}+\frac{1}{2}\cdot\sum_{i=0}^{k+2}(k+2-i)\cdot\left(\frac{1}{2}\right)^{i} \\
&= \left(\frac{1}{2}\right)^{k-1}-\left(\frac{1}{2}\right)^{k}-\left(\frac{1}{2}\right)^{k+2}+\frac{1}{2}\cdot\left(\left(\frac{1}{2}\right)^{k+2-1}+2(k+2)-2\right) \\
&= \left(\frac{1}{2}\right)^{k-1}-\left(\frac{1}{2}\right)^{k}-\left(\frac{1}{2}\right)^{k+2}+\left(\frac{1}{2}\right)^{k+2}+(k+2)-1 \\
&= \left(\frac{1}{2}\right)^{k}+k+1
\end{aligned}$$

Since the case $k = 1$ is equivalent the basis of the induction is established.
We will prove that $s'(k,l) = q(k,l) = \big(q(k-1,l)+q(k,l-1)+1\big)/2 = \big(s'(k-1,l)+s'(k,l-1)+1\big)/2$ (the recursion works for the sum) for each of the three sum parts individually for $l > 1$ and $k > 1$:
Let $s_1(k,l) = \sum_{i=1}^{k}\left(\frac{1}{2}\right)^{l+i-1}\cdot\binom{l+i-2}{i-1}\cdot(k-i+2)$, let $s_2(k,l) = \sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k+j-1}\cdot\binom{k+j-2}{j-1}\cdot(l-j+2)$, and let $s_3(k,l) = \sum_{i=1}^{k}\sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k-i+l-j+1}\cdot\binom{k-i+l-j}{l-j}$
Clearly, $s'(k,l) = s_1(k,l)+s_2(k,l)+s_3(k,l)$.
We start to prove $s_1(k,l) = \frac{1}{2}\cdot s_1(k-1,l)+\frac{1}{2}\cdot s_1(k,l-1)$:

$$\begin{aligned}
&\frac{1}{2}\cdot s_1(k-1,l)+\frac{1}{2}\cdot s_1(k,l-1) \\
=\;& \frac{1}{2}\cdot\sum_{i=1}^{k}\left(\frac{1}{2}\right)^{l-1+i-1}\cdot\binom{l-1+i-2}{i-1}\cdot(k-i+2) \\
&+\frac{1}{2}\cdot\sum_{i=1}^{k-1}\left(\frac{1}{2}\right)^{l+i-1}\cdot\binom{l+i-2}{i-1}\cdot(k-1-i+2)
\end{aligned}$$

$$\begin{aligned}
&= \frac{1}{2}\cdot\sum_{i=1}^{k}\left(\frac{1}{2}\right)^{l+i-2}\cdot\binom{l+i-3}{i-1}\cdot(k-i+2)\\
&\quad+\frac{1}{2}\cdot\sum_{i=2}^{k}\left(\frac{1}{2}\right)^{l+i-2}\cdot\binom{l+i-3}{i-2}\cdot(k-i+2)\\
&= \sum_{i=1}^{k}\left(\frac{1}{2}\right)^{l+i-1}\cdot\left(\binom{l+i-3}{i-1}+\binom{l+i-3}{i-2}\right)\cdot(k-i+2)\\
&= \sum_{i=1}^{k}\left(\frac{1}{2}\right)^{l+i-1}\cdot\binom{l+i-2}{i-1}\cdot(k-i+2)\qquad\text{(using Lemma 8.3)}\\
&= s_1(k,l)
\end{aligned}$$

The proof for $s_2(k,l) = \frac{1}{2}\cdot s_2(k-1,l)+\frac{1}{2}\cdot s_2(k,l-1)$ is analogous to the previous one.
That leaves to prove $s_3(k,l) = \frac{1}{2}\cdot s_3(k-1,l)+\frac{1}{2}\cdot s_3(k,l-1)+\frac{1}{2}$:

$$\begin{aligned}
\frac{1}{2}\cdot s_3(k-1,l)+\frac{1}{2}\cdot s_3(k,l-1)+\frac{1}{2} &= \frac{1}{2}\cdot\sum_{i=1}^{k-1}\sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k-i+l-j}\cdot\binom{k-i+l-j-1}{l-j}\\
&\quad+\frac{1}{2}\cdot\sum_{i=1}^{k}\sum_{j=1}^{l-1}\left(\frac{1}{2}\right)^{k-i+l-j}\cdot\binom{k-i+l-j-1}{l-j-1}+\frac{1}{2}
\end{aligned}$$

The two sums can be swapped, hence we will add the missing element to each sum, so that both sum over l and k. $\binom{l-j-1}{l-j}$ evaluates to zero for $j<l$ because the factor $l-j-1-(l-j)+1=0$ will appear in the denominator at the last position. The missing element for the first sum is

$$\frac{1}{2}\cdot\sum_{j=1}^{l}\left(\frac{1}{2}\right)^{l-j}\cdot\binom{l-j-1}{l-j}=\frac{1}{2}\cdot\left(\frac{1}{2}\right)^{0}\cdot\binom{0-1}{0}=\frac{1}{2}.$$

The missing element for the second sum is ($\binom{x}{-1}=0$ by definition)

$$\frac{1}{2}\cdot\sum_{i=1}^{k}\left(\frac{1}{2}\right)^{k-i}\cdot\binom{k-i-1}{-1}=0$$

Hence we need to subtract $\frac{1}{2}$ when adding the missing elements:

$$\begin{aligned}
&= \frac{1}{2}\cdot\sum_{i=1}^{k}\sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k-i+l-j}\cdot\binom{k-i+l-j-1}{l-j}-\frac{1}{2}\\
&\quad+\frac{1}{2}\cdot\sum_{i=1}^{k}\sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k-i+l-j}\cdot\binom{k-i+l-j-1}{l-j-1}+\frac{1}{2}\\
&= \frac{1}{2}\cdot\sum_{i=1}^{k}\sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k-i+l-j}\cdot\left(\binom{k-i+l-j-1}{l-j}+\binom{k-i+l-j-1}{l-j-1}\right)\\
&= \sum_{i=1}^{k}\sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k-i+l-j+1}\cdot\binom{k-i+l-j}{l-j}\\
&= s_3(k,l)
\end{aligned}$$

As a result,

$$
\begin{aligned}
&\big(s'(k-1,l)+s'(k,l-1)+1\big)/2\\
&= \big(s_1(k-1,l)+s_2(k-1,l)+s_3(k-1,l)+s_1(k,l-1)+s_2(k,l-1)+s_3(k,l-1)+1\big)/2\\
&= \frac{1}{2}\big(s_1(k-1,l)+s_1(k,l-1)\big)+\frac{1}{2}\big(s_2(k-1,l)+s_2(k,l-1)\big)+\frac{1}{2}\big(s_3(k-1,l)+s_3(k,l-1)+1\big)\\
&= s_1(k,l)+s_2(k,l)+s_3(k,l)\\
&= s'(k,l)
\end{aligned}
$$

□

A closed form for the sums could not be derived.

8.3 Two-Leaves-Subtrees as Pseudo Anti-Chain

Two-leaves-subtrees have the nice property that their expected processing time is relatively easy to calculate. Unfortunately they do not form an anti-chain. If we still want to use the result of Theorem 8.1, we can do the following:

1. Calculate the probabilities that the two-leaves-subtree at hand is the first subtree reached that has less than three leaves, or
2. adapt the weights of the expected processing time for a two-leaves-subtree, such that it takes into account that smaller subtrees are also reached.

Since the probabilities of reaching smaller subtrees from greater ones are fixed, this gives us basically the choice, whether we want to include the discounting of the probability in our calculation or whether we want to discount the weights.

Combining Observation 8.1 with Theorem 8.1, the weight that a two-leaves-subtree $(a|k|l)$ has with respect to the tree B with size n is:

$$w(k,l,a,n)=p(a,k,l)+\frac{n-a-k-l-1}{3}$$

The probability that a subtree $(a|k|l)$ is the first subtree to be reached with two leaves is

$$P^f_\alpha[(a|k|l)]=P_\alpha[(a|k|l)]-\frac{1}{2}P_\alpha[(a|k+1|l)]-\frac{1}{2}P_\alpha[(a|k|l+1)]$$

Let C_2 be the set of all two-leaves-subtrees of B, replacing this in the equation from Theorem 8.1, we get:

Corollary 8.2 (Expected Processing Time by Two-Leaves-Subtree Weights).

$$\mathbb{E}(T_\alpha(B))=\sum_{(a|k|l)\in C_2}P^f_\alpha[(a|k|l)]\cdot w(k,l,a,n)$$

Lets define a discounted weight $w_d(k,l,a,n)$ as:

$$w_d(k,l,a,n)=\begin{cases}\frac{1}{6} & \text{if } k>1 \text{ and } l>l,\\ \frac{2k-l+2a+n+4}{6} & \text{if } k>1 \text{ and } l=l,\\ \frac{2l-k+2a+n+4}{6} & \text{if } k=1 \text{ and } l>l,\\ \frac{4a+2n+9}{6} & \text{if } k=1 \text{ and } l=l.\end{cases}$$

Lemma 8.4 (Discounted Two-Leaves-Subtree Weights).

$$\sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]\cdot w(k,l,a,n) = \sum_{(a|k|l)\in C_2} P_\alpha[(a|k|l)]\cdot w_d(k,l,a,n)$$

Proof.

$$\begin{aligned}
&\sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]\cdot w(k,l,a,n)\\
=\;&\sum_{(a|k|l)\in C_2} \Big(P_\alpha[(a|k|l)] - \frac{1}{2}P_\alpha[(a|k+1|l)] - \frac{1}{2}P_\alpha[(a|k|l+1)]\Big)\cdot w(k,l,a,n)\\
=\;&\sum_{(a|k|l)\in C_2} P_\alpha[(a|k|l)]\cdot A(k,l,a,n)
\end{aligned}$$

where

$$\begin{aligned}
A(k,l,a,n) &= \begin{cases} w(k,l,a,n) - w(k-1,l,a,n)/2 - w(k,l-1,a,n)/2 & \text{if } k>1 \text{ and } l>l,\\ w(k,l,a,n) - w(k-1,l,a,n)/2 & \text{if } k>1 \text{ and } l=l,\\ w(k,l,a,n) - w(k,l-1,a,n)/2 & \text{if } k=1 \text{ and } l>l,\\ w(k,l,a,n) & \text{if } k=1 \text{ and } l=l. \end{cases}\\
&= \begin{cases} p(a,k,l) + \frac{n-a-k-l-1}{3} - p(a,k-1,l)/2 - \frac{n-a-k-l-2}{6} & \\ -p(a,k,l-1)/2 - \frac{n-a-k-l-2}{6} & \text{if } k>1 \text{ and } l>l,\\ p(a,k,l) + \frac{n-a-k-l-1}{3} - p(a,k-1,l)/2 - \frac{n-a-k-l}{6} & \text{if } k>1 \text{ and } l=l,\\ p(a,k,l) + \frac{n-a-k-l-1}{3} - p(a,k,l-1)/2 - \frac{n-a-k-l}{6} & \text{if } k=1 \text{ and } l>l,\\ p(a,k,l) + \frac{n-a-k-l-1}{3} & \text{if } k=1 \text{ and } l=l. \end{cases}\\
&= \begin{cases} q(k,l) + a - q(k-1,l)/2 - a/2 - q(k,l-1)/2 - a/2 & \\ -\frac{n-a-k-l-1-(n-a-k-l-2)}{3} & \text{if } k>1 \text{ and } l>l,\\ q(k,l) + a - q(k-1,l)/2 - a/2 + \frac{n-a-k-l-2}{6} & \text{if } k>1 \text{ and } l=l,\\ q(k,l) + a - q(k,l-1)/2 - a/2 + \frac{n-a-k-l-2}{6} & \text{if } k=1 \text{ and } l>l,\\ q(k,l) + a + \frac{n-a-k-l-1}{3} & \text{if } k=1 \text{ and } l=l. \end{cases}\\
&= \begin{cases} q(k,l) - q(k-1,l)/2 - q(k,l-1)/2 - 1/2 + 1/2 - \frac{1}{3} & \text{if } k>1 \text{ and } l>l,\\ q(k,1) + a/2 - q(k-1,1)/2 + \frac{n-a-k-l-2}{6} & \text{if } k>1 \text{ and } l=l,\\ q(1,l) + a/2 - q(1,l-1)/2 + \frac{n-a-k-l-2}{6} & \text{if } k=1 \text{ and } l>l,\\ q(1,1) + \frac{3a+n-a-3}{3} & \text{if } k=1 \text{ and } l=l. \end{cases}\\
&= \begin{cases} \frac{1}{6} & \text{if } k>1 \text{ and } l>l,\\ q(k-1,1)/2 + q(k,0)/2 + 1/2 - q(k-1,1)/2 & \\ +\frac{3a+n-a-k-l-2}{6} & \text{if } k>1 \text{ and } l=l,\\ q(0,l)/2 + q(1,l-1)/2 + 1/2 - q(1,l-1)/2 & \\ +\frac{3a+n-a-k-l-2}{6} & \text{if } k=1 \text{ and } l>l,\\ \frac{5}{2} + \frac{2a+n-3}{3} & \text{if } k=1 \text{ and } l=l. \end{cases}\\
&= \begin{cases} \frac{1}{6} & \text{if } k>1 \text{ and } l>l,\\ k/2 + 1/2 + 1/2 + \frac{3a+n-a-k-l-2}{6} & \text{if } k>1 \text{ and } l=l,\\ l/2 + 1/2 + 1/2 + \frac{3a+n-a-k-l-2}{6} & \text{if } k=1 \text{ and } l>l,\\ \frac{9+4a+2n}{6} & \text{if } k=1 \text{ and } l=l. \end{cases}
\end{aligned}$$

$$\begin{aligned}
&= \begin{cases} \frac{1}{6} & \text{if } k > 1 \text{ and } l > l, \\ \frac{3k+6+3a+n-a-k-l-2}{6} & \text{if } k > 1 \text{ and } l = l, \\ \frac{3l+6+3a+n-a-k-l-2}{6} & \text{if } k = 1 \text{ and } l > l, \\ \frac{9+4a+2n}{6} & \text{if } k = 1 \text{ and } l = l. \end{cases} \\
&= \begin{cases} \frac{1}{6} & \text{if } k > 1 \text{ and } l > l, \\ \frac{2k-l+2a+n+4}{6} & \text{if } k > 1 \text{ and } l = l, \\ \frac{2l-k+2a+n+4}{6} & \text{if } k = 1 \text{ and } l > l, \\ \frac{9+4a+2n}{6} & \text{if } k = 1 \text{ and } l = l. \end{cases} \\
&= w_d(k, l, a, n)
\end{aligned}$$

□

With $w_d(k, l, a, n)$ we can use two-leaves-subtrees as pseudo-anti-chain. The only things still unknown are the probabilities of reaching two-leaves-subtrees. We will try to deal with that in the next chapter.

8.4 HLF Revisited

The results of the previous sections give another clue, why HLF works for the two machine problem but not for the three machines one. If there are only two machines, the subtrees that matter in the sense of a weight by probability approach are the one-leaf-subtrees. Each such subtree corresponds exactly to a single leaf, the higher that leaf, the larger the expected processing time for being left with the corresponding tree. Hence in the subtree view it seems intuitively right to schedule the highest level leaves first.

For three machines and two-leaves-subtrees the problem is not as clear. For a tree B with n nodes the two-leaves-subtree that is largest in the number of nodes must not necessarily be the one with the largest weight. Consider subtrees $(0|2|3)$ and $(0|1|3)$. The subtrees' expected processing times are $p(0, 2, 3) = 71/16$ and $p(0, 1, 3) = 33/8$. The weights in the sum for the expected processing time of B are

$$w(2, 3, 0, n) = \frac{71}{16} + \frac{n - 0 - 2 - 3 - 1}{3} = \frac{117 + 16n}{48} = \frac{39}{16} + \frac{n}{3} = 2.4375 + \frac{n}{3}$$

and

$$w(1, 3, 0, n) = \frac{33}{8} + \frac{n - 0 - 1 - 3 - 1}{3} = \frac{59 + 8n}{24} = \frac{59}{24} + \frac{n}{3} = 2.458\bar{3} + \frac{n}{3}$$

Hence, $w(1, 3, 0, n) > w(2, 3, 0, n)$, the largest subtree does not have the largest weight. Therefore only scheduling the highest nodes cannot be optimal.

8.5 A Lower and an Upper Bound on the Total Expected Processing Time

Theorem 8.1 helps us to determine a lower and an upper bound on the total expected processing time for a given tree B:

Lemma 8.5 (A Lower and an Upper Bound on the Total Expected Processing Time). *Let $L_{min} = (a_{min}|k_{min}|l_{min})$ be the two-leaves-subtree, having the smallest weight $w(k_{min}, l_{min}, a_{min}, |B|)$, and let $L_{max} = (a_{max}|k_{max}|l_{max})$ be the two-leaves-subtree, having the largest weight $w(k_{max}, l_{max}, a_{max}, |B|)$. Then $w(k_{min}, l_{min}, a_{min}, |B|)$ is a lower and $w(k_{max}, l_{max}, a_{max}, |B|)$ an upper bound for the total expected processing time of B.*

Proof. From Corollary 8.2:

$$\begin{aligned}
\mathbb{E}(T_\alpha(B)) = &\sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]\cdot w(k,l,a,n)\\
&\geq \sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]\cdot w(k_{min},l_{min},a_{min},|B|)\\
&= w(k_{min},l_{min},a_{min},|B|)\cdot \sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]1\\
&= w(k_{min},l_{min},a_{min},|B|)
\end{aligned}$$

Similarly,

$$\begin{aligned}
\mathbb{E}(T_\alpha(B)) = &\sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]\cdot w(k,l,a,n)\\
&\leq \sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]\cdot w(k_{max},l_{max},a_{max},|B|)\\
&= w(k_{max},l_{max},a_{max},|B|)\cdot \sum_{(a|k|l)\in C_2} P^f_\alpha[(a|k|l)]1\\
&= w(k_{max},l_{max},a_{max},|B|)
\end{aligned}$$

□

Chapter 9

The Probability of Reaching a Two-Leaves-Subtree

9.1 Working Towards a Two-Leaves-Subtree

Given a tree B and a two-leaves-subtree L we want to be able to determine the probability that L is the first subtree reached with only two leaves left. For given tree B and subtree L we classify the nodes as nodes belonging to L, nodes that are above L, called 'descends' (set D), and nodes that are not above L, called 'on-descends'(set N). See Figure 9.1 for a schematic view. The descends are all nodes that are descends of the leaves of L, but not in L. The non-descends are all nodes that "grow out at the side" of L, simply $N = (B \setminus L) \setminus D$.

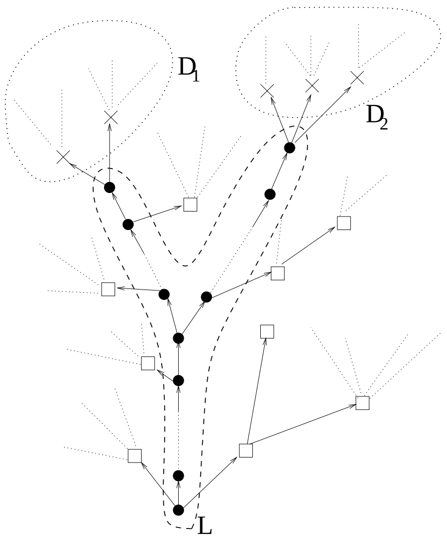


Figure 9.1: A Schematic View for Classifying Tree Nodes Based on a Selected Two-Leaves-Subtree (set L of the two-leaves-subtree's nodes are black dots, the set $D = D_1 \cup D_2$ of the two-leaves-subtree's descend nodes are crosses, and the set N of the two-leaves-subtree's non-descends are boxes)

For any subset of nodes of B that represent a forest, we can calculate a worst case scheduling scenario as the maximal possible number of nodes only schedulable with two or one machine. For a single tree these correspond to the largest one-leaf-subtree and the largest two-leaves-subtree (note, that this is not the expected worst case, but the worst case as it can occur during the actual processing of the tasks).

The highest node corresponds to the largest one-leaf-subtree. The largest two-leaves-subtree can easily be calculated as the largest one-leaf-subtree with the largest additional branch:

Lemma 9.1. *One leaf of the largest two-leaves-subtree L of a tree B must also be a highest leaf.*

Proof. Let r be the root of B. Assume the statement does not hold, then there must be two leaves a,b that define L. Without loss of generality, suppose a is higher than b. Let $c = lca(a, b)$. Then the size of L is $dist(c, r) + dist(a, c) + dist(b, c) + 1$. Let h be the highest leaf ($h \neq a$ and $h \neq b$ by assumption). Clearly,

$$dist(h, r) \geq dist(a, r) \geq dist(b, r) \tag{9.1}$$

Let $d_a = lca(a, h)$, and let $d_b = lca(b, h)$, we will make a case distinction:

a) $d_a \in path(c, a)$: $dist(d_a, h) \geq dist(d_a, a)$ (by 9.1), hence $dist(c, r) + dist(a, c) + dist(b, c) + 1 \leq dist(c, r) + dist(d_a, c) + dist(h, d_a) + dist(b, c) + 1$.

b) $d_b \in path(c, b)$: analogous.

c) $d_a = d_b \wedge d_a \in path(r, c)$, then $dist(h, d_a) \geq dist(a, d_a) \geq dist(b, d_a)$.

In any case the assumption leads to a contradiction. □

Let $|N| = l_3 + l_2 + l_1$, where $l_2 + l_1$ is the size of a maximal forest with two leaves in N and l_1 is a maximal one-leaf-subtree in N. Let $|D| = k_3 + k_2 + k_1$, where $k_2 + k_1$ is the size of a maximal forest with two leaves in D and k_1 is a maximal one-leaf-subtree in D.

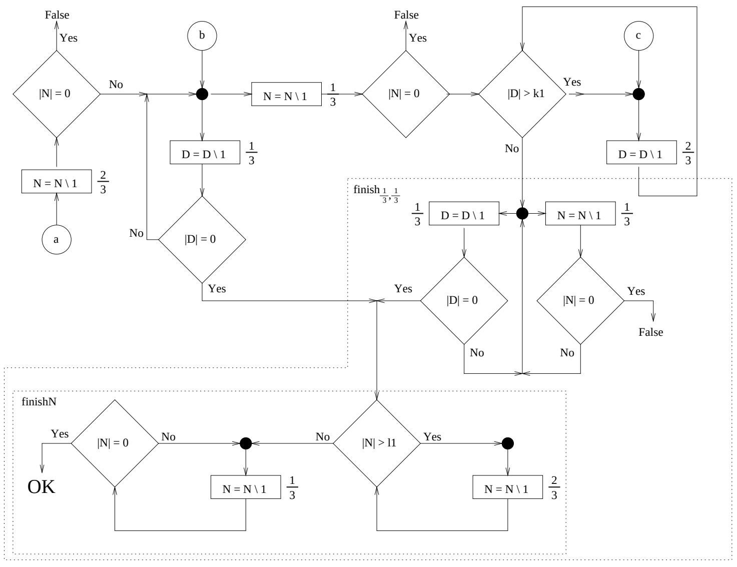


Figure 9.2: Formulae of Case 2 as Diagram

We will now estimate the worst case probability P for reaching a given subtree L of B under the assumption that a set of scheduled nodes α has already been selected and that in subsequent steps the strategy will try to reach L. L can be reached as the first two-leaves-subtree by processing all nodes of D before the last node of N and processing all nodes of $D \cup N$ before any node of L. The process is divided into three major cases of which the complicated ones are shown in Figures 9.2 and 9.3. At the filled black nodes random choices are made, leading to the events in the square boxes. The diamonds represent branches that are decided by the current state (the number of nodes in the sets N, D). These branches depend on the outcome of the previous random choices. The probabilities of the choices are written next to the corresponding occurring event. To reduce the complexity, parts of the diagrams have been summarized to functions that are marked with dotted lines. These functions form "building blocks" of the formulae, when evaluated by a program.

We will later estimate the time needed to evaluate the formulae below. The complete evaluation will include summing up and multiplying a number of elements which are in turn powers of $\frac{1}{2}$, $\frac{1}{3}$, or $\frac{2}{3}$, or which are binomials. Note that all parameters are $\mathcal{O}(n)$, so that there can be at most $\mathcal{O}(n^2)$ binomials that need to be evaluated and $\mathcal{O}(n)$ powers. Using some sort of dynamic programming algorithm and a look-up scheme, all binomials and powers can be calculated in $\mathcal{O}(n^2)$ and each element in the sum is a simple look-up costing $\mathcal{O}(1)$.

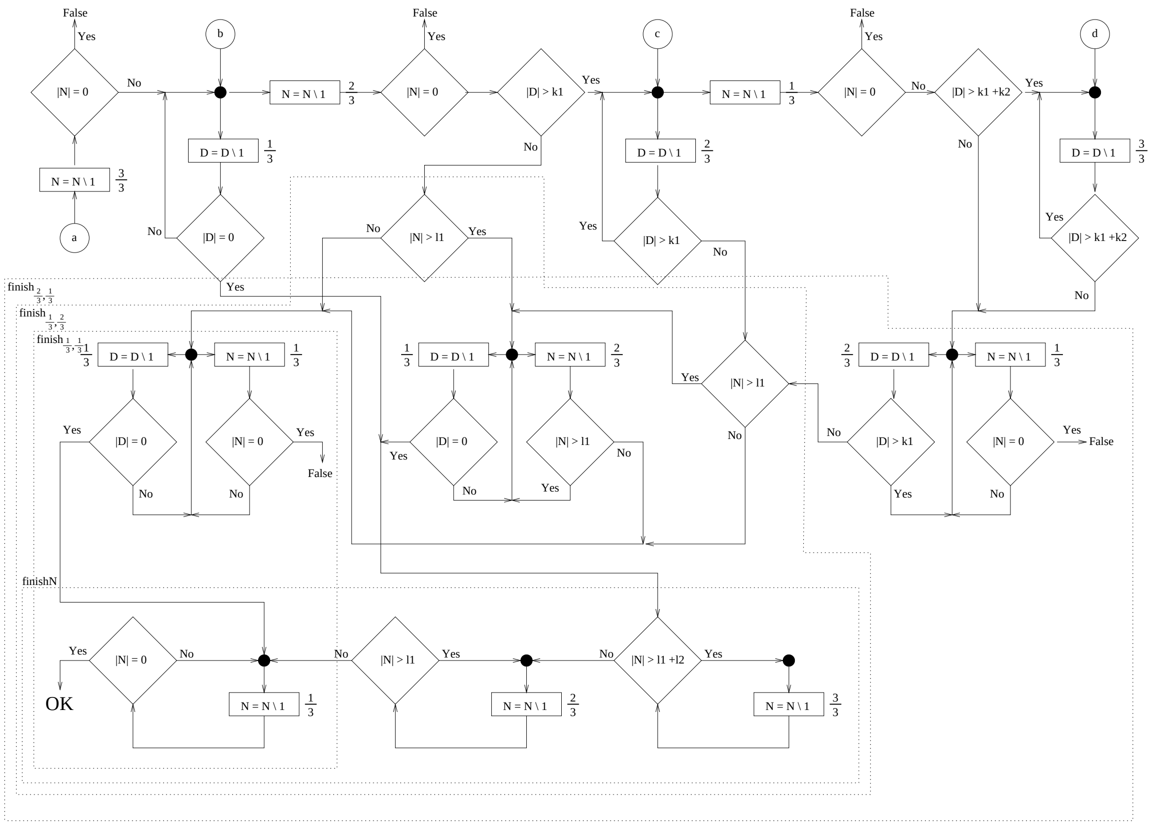


Figure 9.3: Formulae of Case 3 as Diagram

9.1.1 Special Cases

Some probabilities are very easy to calculate. For these special cases we can even give the exact probabilities.

A) $|N| = 0$

There are no non-descend nodes. The last node to be finished, before L can be reached, is a descend. Hence, the tree, one step before L is reached, is also a two-leaves-subtree. The probability of reaching L as the first two-leaves-subtree is therefore zero.

$\Rightarrow P = 0$

B) $|D| = 0$

There are no nodes above the subtree. By our assumption the leaves of L will be scheduled as late as possible. The probability depends upon how many leaves are already scheduled:

(i) $|\alpha \cap L| = 2$
$\Rightarrow P = \left(\frac{1}{3}\right)^{|N|}$

(ii) $|\alpha \cap L| = 1$
$\Rightarrow P = \left(\frac{2}{3}\right)^{(l_2+l_3)} \left(\frac{1}{3}\right)^{l_1}$

(iii) $|\alpha \cap L| = 0$
$\Rightarrow P = \left(\frac{2}{3}\right)^{l_2} \left(\frac{1}{3}\right)^{l_1}$

9.1.2 General Cases

In the general case any leaves might be scheduled. It is assumed that the scheduler tries to schedule leaves of L as late as possible. It is also assumed that nodes in D are scheduled first, afterwards the nodes from N. Since the points where the number of leaves in either D or N drops below one or two are not known exactly, we assume a "Worst Case", that is we assume that a largest subtree (or sub-forest) will remain with one or two leaves left.

Case 1) $|\alpha \cap L| = 2$

It follows, that $D = \emptyset$. This is equivalent to the special case B.

Case 2) $|\alpha \cap L| = 1$

Throughout the process of scheduling towards reaching L, nodes from N and D are finished. Let ΔN be the number of nodes already finished from N. Let ΔD be the number of nodes already finished from D.

If L is reached, the last task finished must be from N. Hence there may be a number (possibly one) of nodes in N that are finished before L is reached. This is expressed in the following function.

$$finishN^2(l_1, l_2, l_3, \Delta N) := \begin{cases} \left(\frac{1}{3}\right)^{l_1} \left(\frac{2}{3}\right)^{l_3+l_2-\Delta N} & \text{if } \Delta N \leq l_2 + l_3, \\ \left(\frac{1}{3}\right)^{l_1+l_2+l_3-\Delta N} & \text{if } \Delta N > l_2 + l_3. \end{cases}$$

Before that there is a phase where there is only one leaf left to schedule in D and one leaf from N is already scheduled. This phase is captured in the following function (the remaining leaves from D are scheduled interleaved with some, but not all, leaves from N in arbitrary order):

$$finish^2_{\frac{1}{3},\frac{1}{3}}(l_1, l_2, l_3, |D|, |N|, \Delta D, \Delta N) :=$$
$$\sum_{i=0}^{|N|-\Delta N-1} \left(\frac{1}{3}\right)^i \left(\frac{1}{3}\right)^{|D|-\Delta D} \binom{|D| - \Delta D + i}{i} finishN^2(l_1, l_2, l_3, \Delta N + i)$$

There are $\mathcal{O}(n)$ elements in the above sum.

With this building blocks the formulae will be easier to describe:

(a) $|\alpha \cap N| = 2$

If L is reached, then one of the machines working at a node in N finishes first. It is assigned to D. Three things can happen: Either all nodes from D are scheduled and finished before the next node from N, or enough nodes from D are finished so that only one leaf is left in D, or the second node scheduled initially in N is finished while there are still two leaves left to be scheduled in D.

$$\Rightarrow P = \frac{2}{3}\Bigg(\left(\frac{1}{3}\right)^{|D|} finishN^2(l_1, l_2, l_3, 1) + \sum_{i=k_3+k_2}^{|D|-1} \left(\frac{1}{3}\right)^i \frac{1}{3} finish^2_{\frac{1}{3},\frac{1}{3}}(l_1, l_2, l_3, |D|, |N|, i, 2) + \sum_{i=0}^{k_3+k_2-1} \left(\frac{1}{3}\right)^i \frac{1}{3} \left(\frac{2}{3}\right)^{k_3+k_2-i} finish^2_{\frac{1}{3},\frac{1}{3}}(l_1, l_2, l_3, |D|, |N|, k_3+k_2, 2)\Bigg)$$

Because $finish^2_{\frac{1}{3},\frac{1}{3}}$ can sum up to $\mathcal{O}(n)$ elements, the evaluation of the above formula may take $\mathcal{O}(n^2)$ steps.

(b) $|\alpha \cap N| = 1$

This case is essentially the same as the case (a) only that a machine from N must not finish before a leaf in D is scheduled.

$$\Rightarrow P = \left(\frac{1}{3}\right)^{|D|} finishN^2(l_1, l_2, l_3, 0) + \sum_{i=k_3+k_2}^{|D|-1} \left(\frac{1}{3}\right)^i \frac{1}{3} finish^2_{\frac{1}{3},\frac{1}{3}}(l_1, l_2, l_3, |D|, |N|, i, 1) + \sum_{i=0}^{k_3+k_2-1} \left(\frac{1}{3}\right)^i \frac{1}{3} \left(\frac{2}{3}\right)^{k_3+k_2-i} finish^2_{\frac{1}{3},\frac{1}{3}}(l_1, l_2, l_3, |D|, |N|, k_3+k_2, 1)$$

Because $finish^2_{\frac{1}{3},\frac{1}{3}}$ can sum up to $\mathcal{O}(n)$ elements, the evaluation of the above formula may take $\mathcal{O}(n^2)$ steps.

(c) $|\alpha \cap N| = 0$

There are already two machines working at nodes from D. Nodes from D are finished, until there is only one leaf left to schedule in D.

$$\Rightarrow P = \left(\frac{2}{3}\right)^{k_3+k_2} finish^2_{\frac{1}{3},\frac{1}{3}}(l_1, l_2, l_3, |D|, |N|, k_3+k_2, 0)$$

The evaluation of the above formula may take $\mathcal{O}(n)$ steps (excluding binomials).

Case 3) $|\alpha \cap L| = 0$

As for case (2), let ΔN be the number of nodes already finished from N and let ΔD be the number of nodes already finished from D.

We again start with defining some building block functions used to describe the complexity.

The last thing that happens is that the remaining nodes from N are finished.

$$finishN^3(l_1, l_2, l_3, \Delta N) := \begin{cases} \left(\frac{1}{3}\right)^{l_1} \left(\frac{2}{3}\right)^{l_2} & \text{if } \Delta N \le l_3, \\ \left(\frac{1}{3}\right)^{l_1} \left(\frac{2}{3}\right)^{l_2+l_3-\Delta N} & \text{if } \Delta N > l_3 \wedge \Delta N \le l_2 + l_3, \\ \left(\frac{1}{3}\right)^{l_1+l_2+l_3-\Delta N} & \text{if } \Delta N > l_2 + l_3. \end{cases}$$

One way to finish occurs, when the last sequence of finishing nodes is either N or D with probability $p = \frac{1}{3}$ each (i.e. only one leaf is left in each D and N).

$$finish^3_{\frac{1}{3},\frac{1}{3}}(|D|,|N|,\Delta D,\Delta N) := \sum_{i=0}^{|N|-\Delta N-1} \left(\frac{1}{3}\right)^i \left(\frac{1}{3}\right)^{|D|-\Delta D} \binom{|D|-\Delta D+i}{i} \left(\frac{1}{3}\right)^{|N|-\Delta N-i}$$

The above sum has $\mathcal{O}(n)$ elements.

The next building block will be the point after which D has only one leaf left.

$$finish^3_{\frac{1}{3},\frac{2}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,\Delta D,\Delta N) :=$$
$$\begin{cases} finish^3_{\frac{1}{3},\frac{1}{3}}(|D|,|N|,\Delta D,\Delta N) & \text{if } \Delta N \geq l_2+l_3, \\ \sum_{i=0}^{l2+l3-1-\Delta N} \left(\frac{1}{3}\right)^{|D|-\Delta D} \left(\frac{2}{3}\right)^i \binom{|D|-\Delta D+i}{i} finishN^3(l_1,l_2,l_3,\Delta N+i) & \\ \quad + \sum_{i=0}^{|D|-\Delta D-1} \left(\frac{1}{3}\right)^i \left(\frac{2}{3}\right)^{l2+l3-\Delta N} \binom{l2+l3-\Delta N+i}{i} & \\ \qquad \cdot finish^3_{\frac{1}{3},\frac{1}{3}}(|D|,|N|,\Delta D+i,l2+l3) & \text{if } \Delta N < l_2+l_3. \end{cases}$$

The above sum has $\mathcal{O}(n^2)$ elements because it might call $finish^3_{\frac{1}{3},\frac{1}{3}}$ $\mathcal{O}(n)$ times.

The next building block will be the point after which D has less than three leaves left.

$$finish^3_{\frac{2}{3},\frac{1}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,\Delta D,\Delta N) :=$$
$$\sum_{i=0}^{N-\Delta N-1} \left(\frac{2}{3}\right)^{k_3+k_2-\Delta D} \left(\frac{1}{3}\right)^i \binom{k_3+k_2-\Delta D+i}{i}$$
$$\cdot\, finish^3_{\frac{1}{3},\frac{2}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,k_3+k_2,\Delta N+i)$$

The last building block's sum has $\mathcal{O}(n^3)$ elements because it might call $finish^3_{\frac{2}{3},\frac{1}{3}}$ $\mathcal{O}(n)$ times.

With this building blocks the formulae will be:

(a) $|\alpha \cap N| = 3$

The only thing that can happen at the beginning is that a node from N is finished. After that there are two machines working at nodes in N, and one machine working at nodes in D. This last machine can now finish a number of nodes from D, such that either D is completely finished, only one leaf is left in D, two leaves are left in D, or three leaves are left in D before another node from N is finished.

$$
\begin{aligned}
\Rightarrow P = & \left(\frac{1}{3}\right)^{|D|} finishN^3(l_1, l_2, l_3, 1) \\
& + \sum_{i=k_3+k_2}^{|D|-1} \left(\frac{1}{3}\right)^i \frac{2}{3} finish^3_{\frac{1}{3},\frac{2}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, i, 2) \\
& + \sum_{i=k_3}^{k_3+k_2-1} \left(\frac{1}{3}\right)^i \frac{2}{3} \cdot \Bigg(\\
& \qquad \left(\frac{2}{3}\right)^{k_2+k_3-i} finish^3_{\frac{1}{3},\frac{2}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, k_3+k_2, 2) \\
& \qquad + \sum_{j=0}^{k_2+k_3-i-1} \left(\frac{2}{3}\right)^j \frac{1}{3} finish^3_{\frac{2}{3},\frac{1}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, i+j, 3) \Bigg) \\
& + \sum_{i=0}^{k_3-1} \left(\frac{1}{3}\right)^i \frac{2}{3} \cdot \Bigg(\\
& \qquad \left(\frac{2}{3}\right)^{k_2+k_3-i} finish^3_{\frac{1}{3},\frac{2}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, k_3+k_2, 2) \\
& \qquad + \sum_{j=k_3-i}^{k_2+k_3-i-1} \left(\frac{2}{3}\right)^j \frac{1}{3} finish^3_{\frac{2}{3},\frac{1}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, i+j, 3) \\
& \qquad + \sum_{j=0}^{k_3-i-1} \left(\frac{2}{3}\right)^j \frac{1}{3} finish^3_{\frac{2}{3},\frac{1}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, k_3, 3) \Bigg)
\end{aligned}
$$

Because $finish^3_{\frac{2}{3},\frac{1}{3}}$ is called $\mathcal{O}(n^2)$ times, the evaluation of the above formula may take $\mathcal{O}(n^5)$ steps.

(b) $|\alpha \cap N| = 2$

This case is essentially the same than the case (a) only that a machine from N must not finish before a leaf in D is scheduled.

$$
\begin{aligned}
\Rightarrow P = & \left(\frac{1}{3}\right)^{|D|} finishN^3(l_1, l_2, l_3, 0) \\
& + \sum_{i=k_3+k_2}^{|D|-1} \left(\frac{1}{3}\right)^i \frac{2}{3} finish^3_{\frac{1}{3},\frac{2}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, i, 1) \\
& + \sum_{i=k_3}^{k_3+k_2-1} \left(\frac{1}{3}\right)^i \frac{2}{3} \cdot \Bigg(\\
& \qquad \left(\frac{2}{3}\right)^{k_2+k_3-i} finish^3_{\frac{1}{3},\frac{2}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, k_3+k_2, 1) \\
& \qquad + \sum_{j=0}^{k_2+k_3-i-1} \left(\frac{2}{3}\right)^j \frac{1}{3} finish^3_{\frac{2}{3},\frac{1}{3}}(l_1, l_2, l_3, k_1, k_2, k_3, |D|, |N|, i+j, 2) \Bigg)
\end{aligned}
$$

$$+\sum_{i=0}^{k_3-1}\left(\frac{1}{3}\right)^i\frac{2}{3}\cdot\Bigg(\left(\frac{2}{3}\right)^{k_2+k_3-i} finish^3_{\frac{1}{3},\frac{2}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,k_3+k_2,1)$$
$$+\sum_{j=k_3-i}^{k_2+k_3-i-1}\left(\frac{2}{3}\right)^j\frac{1}{3}finish^3_{\frac{2}{3},\frac{1}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,i+j,2)$$
$$+\sum_{j=0}^{k_3-i-1}\left(\frac{2}{3}\right)^j\frac{1}{3}finish^3_{\frac{2}{3},\frac{1}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,k_3,2)\Bigg)$$

Because $finish^3_{\frac{2}{3},\frac{1}{3}}$ is called $\mathcal{O}(n^2)$ times, the evaluation of the above formula may take $\mathcal{O}(n^5)$ steps.

(c) $|\alpha \cap N| = 1$

There are already two machines working at leaves in D. If the machine working at a node in N finishes before the two machines have finished enough nodes from D so that there are less than three leaves in D, three machines can work for some time at nodes in D.

$$\Rightarrow P=\sum_{i=0}^{k_3-1}\left(\frac{2}{3}\right)^i\frac{1}{3}finish^3_{\frac{2}{3},\frac{1}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,k_3,1)$$
$$+\sum_{i=k_3}^{k_3+k_2-1}\left(\frac{2}{3}\right)^i\frac{1}{3}finish^3_{\frac{2}{3},\frac{1}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,i,1)$$
$$+\left(\frac{2}{3}\right)^{k_3+k_2} finish^3_{\frac{1}{3},\frac{2}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,k_3+k_2,0)$$

Because $finish^3_{\frac{2}{3},\frac{1}{3}}$ is called $\mathcal{O}(n)$ times, the evaluation of the above formula may take $\mathcal{O}(n^4)$ steps.

(d) $|\alpha \cap N| = 0$

Already, three machines are working at nodes in D. They continue to decrease D until there are only two leaves left. This is expressed in one of the building blocks:

$\Rightarrow P = finish^3_{\frac{2}{3},\frac{1}{3}}(l_1,l_2,l_3,k_1,k_2,k_3,|D|,|N|,k_3,0)$

The evaluation $finish^3_{\frac{2}{3},\frac{1}{3}}$ may take $\mathcal{O}(n^3)$ steps.

9.1.3 Usage in an Algorithm

First note, that the evaluation of the above formulae may take $\mathcal{O}(n^5)$ steps (the $\mathcal{O}(n^2)$ for the binomials do not really matter). A tree has at most $\mathcal{O}(n^2)$ two-leaves-subtrees, so comparing all subtrees takes $\mathcal{O}(n^7)$. This makes only sense in relation to the $\mathcal{O}(n^3 2^{cn})$ of the dynamic programming algorithm of chapter 6.2.

An algorithm can use the above formula to select a schedule that is best at reaching the selected subtree in *the worst case* (see section 9.2.3).

9.2 Avoiding a Two-Leaves-Subtree

The idea of the previous section can be applied to the opposite goal, to avoid two-leaf-subtrees. For an algorithm to reach a good total expected processing time it should try to reach subtrees with low expected processing time and avoid subtrees with high expected processing time. We will take a look at the latter here.

Before we start we take a closer look at the subtrees that result in a high expected processing time. As observed in section 8.4, the largest two-leaves-subtrees are not necessarily the ones having the highest weight. The following lemma will give us a small help in identifying the two-leaves-subtrees with the highest weight.

Lemma 9.2 (Two-Leaves-Subtrees with Highest Weight). *Given a tree B with size n the subtree $(a|k|l)$ having the highest weight $w(n, a, k, l)$ has at least one common leaf with B.*

Proof. Given any two-leaves-subtree $L = (a|k-1|l-1)$, where both leaves of L represent inner nodes in B, then there must exist another two-leaves-subtree $L' = (a|k|l)$ of B. The weight of L is $w(k-1, l-1, a, n)$, the weight of L' is $w(k, l, a, n)$. We will show that $w(k, l, a, n) \geq w(k-1, l-1, a, n)$.

$$\begin{aligned} & w(k,l,a,n) \geq w(k-1,l-1,a,n) \\ \Leftrightarrow \quad & p(a,k,l) + \frac{n-a-k-l-1}{3} \geq p(a,k-1,l-1) + \frac{n-a-k-l-1+2}{3} \\ \Leftrightarrow \quad & q(k,l) + a \geq q(k-1,l-1) + a + \frac{2}{3} \\ \Leftrightarrow \quad & q(k,l) \geq q(k-1,l-1) + \frac{2}{3} \end{aligned}$$

If $k > 1$ and $l > 1$, then we can evaluate the recursion of $q(k,l)$ once on each side:

$$\Leftrightarrow \quad \frac{1}{2}q(k-1,l) + \frac{1}{2}q(k,l-1) \geq \frac{1}{2}q(k-2,l-1) + \frac{1}{2}q(k-1,l-2) + \frac{2}{3}$$

The result can be splitted, such that if both equations hold true, the unsplit equation also holds true:

$$\begin{aligned} \Leftarrow \quad & \frac{1}{2}q(k-1,l) \geq \frac{1}{2}q(k-2,l-1) + \frac{1}{2}\cdot\frac{2}{3} \\ \wedge \; & \frac{1}{2}q(k,l-1) \geq \frac{1}{2}q(k-1,l-2) + \frac{1}{2}\cdot\frac{2}{3} \\ \Leftrightarrow \quad & q(k-1,l) \geq q(k-2,l-1) + \frac{2}{3} \\ \wedge \quad & q(k,l-1) \geq q(k-1,l-2) + \frac{2}{3} \end{aligned}$$

Therefore we can reduce the parameters by one, while all parameters stay above zero.

If the equation holds for all pairs (k', l'), where either $k' < k \wedge l' \leq l$ or $k' \leq k \wedge l' < l$, then the equation also holds for k, l.

We only need to show that the equation holds for all pairs $(1, l)$ and $(k, 1)$, $k, l \geq 1$.

For $k > 1$ and $l = 1$ we will prove the equation by induction on k:

$k = 1$:

$$q(0,0) + \frac{2}{3} = \frac{5}{3}$$

$$q(1,1) = \frac{5}{2} \geq \frac{5}{3}$$

$k > 1$:

$$\begin{aligned}
q(k,1) &= \frac{1}{2} + \frac{1}{2}q(k-1,1) + \frac{1}{2}q(k,0) \\
&= \frac{1}{2} + \frac{1}{2}q(k-1,1) + \frac{1}{2}(k+1) \\
&\geq \frac{1}{2} + \frac{1}{2}\left(q(k-2,0) + \frac{2}{3}\right) + \frac{1}{2}(k+1) \\
&= \frac{1}{2} + \frac{1}{2}(k-1) + \frac{1}{2}(k+1) + \frac{1}{3}
\end{aligned}$$

$$\begin{aligned}
&= \frac{1}{2}(2k+1) + \frac{1}{3} \\
&= k + \frac{1}{2} + \frac{1}{3} \\
&= k + \frac{5}{6} \\
&\geq k + \frac{2}{3} \\
&= q(k-1,0) + \frac{2}{3}
\end{aligned}$$

Since the case $k = 1$ and $l > 1$ is analogous, the equation $w(k,l,a,n) \geq w(k-1,l-1,a,n)$ holds for all $k > 0, l > 0$.

Therefore given any two-leaves-subtree, if we can "add" a leaf to each branch, the resulting subtree has a larger weight. □

Lemma 9.2 will allow us to focus on subtrees, where at most one leaf is an inner node of the super-tree.

We can now more easily find a heaviest two-leaves-subtree and we also know that the descends are all children, grandchildren, or descend from one leaf only. If an algorithm is working towards avoiding the highest weighted two-leaves-subtree, then it will schedule its leaves as early as possible. If there are descends above a leaf, these must be removed first, or all non-descends must be removed before all descends.

Unfortunately there is no known way to decide, whether it is better to remove non-descends and try to reach the higher tree, or whether it is better to remove descends and try to reach the lower tree. For both cases there exists a counter example. An example where $w(k,l+1,a,n) > w(k,l-1,a,n)$ is $w(8,7,a,n) = 58217/12288 + n/3 + a > 28991/6144 + n/3 + a = w(8,5,a,n)$. An example where $w(k,l+1,a,n) < w(k,l-1,a,n)$ is $w(8,6,a,n) = 1197/256 + n/3 + a > 3731/768 + n/3 + a = w(8,4,a,n)$.

We will *assume* that a strategy prefers the lower tree because the other tree leaf might be "accidently" removed, hence leading to a smaller tree after all. Also if the non-descends are removed, there is an earlier point with only two-leaves-left.

We will now calculate the probability P that L is reached under the assumption that it is avoided in favor of a smaller tree and that the worst case occurs (trees are reduced as quickly as possible to few leaves – similar to the assumptions of the preceding section).

9.2.1 Special Cases

The easy cases are treated first. These are the cases, where either no descends or no non-descends exist.

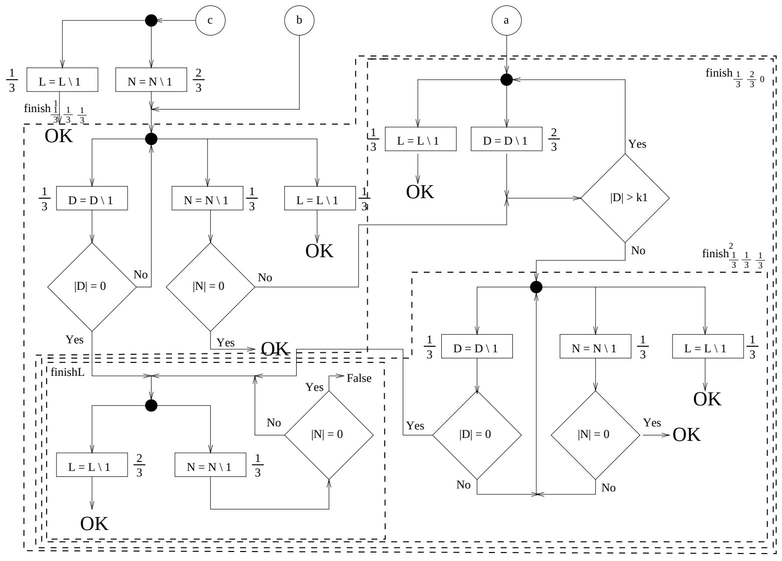


Figure 9.4: Formulae of Case 1 as Diagram

A) If $N = \emptyset$, then

(i) $D = \emptyset$

The tree is already reached, hence

$\Rightarrow P = 0$

(ii) $D \neq \emptyset$

As already mentioned in section 9.1.1, the tree can never be reached as the first two-leaves-subtree.

$\Rightarrow P = 1$

B) If $D = \emptyset$ and $|N| > 0$, then the leaves of L will be scheduled as soon as possible and L is reached only if all remaining nodes of N are finished before a single node from L is finished.

(i) $|\alpha \cap L| = 2$

L is only reached if N can be finished first with the remaining machine.

$$\Rightarrow P = 1 - \left(\frac{1}{3}\right)^{|N|}$$

(ii) $|\alpha \cap L| = 1$

L is only reached if N can be finished first with the remaining machines. In the first step there are two machines available, otherwise only one.

$$\Rightarrow P = 1 - \frac{2}{3} \cdot \left(\frac{1}{3}\right)^{|N|-1}$$

(iii) $|\alpha \cap L| = 0$

L is only reached if N can be finished first with the remaining machines. In the first step there are three, in the second step two machines available, otherwise only one.

$$\Rightarrow P = 1 - \frac{2}{3} \cdot \left(\frac{1}{3}\right)^{|N|-2}$$

9.2.2 General Cases

These are the cases, where neither $D = \emptyset$ nor $N = \emptyset$. Figures 9.4 and 9.5 give an outline of the possible paths similar to the diagrams in section 9.1. For the complexity estimation we will make the same assumptions as in section 9.1.

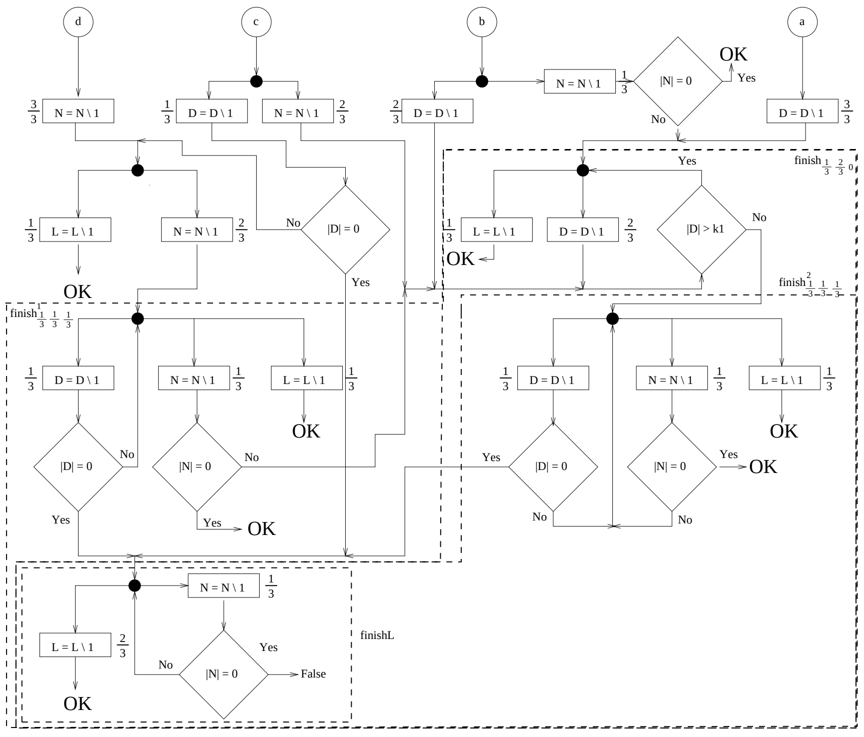


Figure 9.5: Formulae of Case 2 as Diagram

Case 1 If $|\alpha \cap L| = 1$, then all machines are tried to be assigned to D, until $D = \emptyset$. Let ΔN be the number of nodes already finished from N. Let ΔD be the number of nodes already finished from D.

The last chance to avoid L occurs when D is already finished, two leaves of L are scheduled and one leaf from N.

$$finishL^1(N, \Delta N) := 1 - \left(\frac{1}{3}\right)^{N - \Delta N}$$

The evaluation takes $\mathcal{O}(1)$ steps (remember that we calculate binomials and powers beforehand).

Before that a number of times the probability for each set may be equal. We can finish Ns and Ds, until either a node from L is finished, the last node from N is finished, or the last node from D is finished.

We assume that scheduling D is preferred to scheduling N. Then there may be a point, where there is only one leaf left in D. The following function captures this:

$$
\begin{aligned}
&finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}{}^{2}(N,\Delta N,D,\Delta D) := \\
&\qquad\qquad \frac{1}{3}\cdot\sum_{i=0}^{D-\Delta D-1}\sum_{j=0}^{N-\Delta N-1}\left(\frac{1}{3}\right)^{i+j}\binom{i+j}{j} \\
&\qquad\qquad +\sum_{i=0}^{D-\Delta D-1}\left(\frac{1}{3}\right)^{i+N-\Delta N}\binom{i+N-\Delta N}{i} \\
&\qquad +\sum_{i=0}^{N-\Delta N-1}\left(\frac{1}{3}\right)^{i+D-\Delta D}\binom{i+D-\Delta D}{i}\cdot finishL^{1}(N,\Delta N+i)
\end{aligned}
$$

The evaluation of the above sum takes $\mathcal{O}(n^2)$ steps.

If D has more than one leaf, two machines can be assigned to D for some time, which is captured in

$$
\begin{aligned}
&finish^{1}_{\frac{1}{3},\frac{2}{3},0}(k_1,N,\Delta N,D,\Delta D) := \\
&\qquad\begin{cases}
1-\left(\frac{2}{3}\right)^{D-\Delta D-k_1}+ & \\
\left(\frac{2}{3}\right)^{D-\Delta D-k_1}\cdot finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}{}^{2}(N,\Delta N,D,D-k_1) & \text{if } D-\Delta D>k_1, \\
finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}{}^{2}(N,\Delta N,D,\Delta D) & \text{otherwise.}
\end{cases}
\end{aligned}
$$

The evaluation of the above sum takes $\mathcal{O}(n^2)$ steps because of its call to $finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}{}^{2}$.

The case that a node from N is scheduled at the beginning and the machine can potentially be assigned to a node in D is described by

$$
\begin{aligned}
&finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}{}^{1}(k_1,N,\Delta N,D,\Delta D) := \\
&\qquad\qquad\qquad \frac{1}{2}-\frac{1}{2}\left(\frac{1}{3}\right)^{D-\Delta D}+ \\
&\qquad\qquad \left(\frac{1}{3}\right)^{D-\Delta D} finishL^{2}(N,\Delta N)+ \\
&\quad\begin{cases}
\frac{1}{2}-\frac{1}{2}\left(\frac{1}{3}\right)^{D-\Delta D} & \text{if } N-\Delta N=1, \\
\frac{1}{3}\sum_{i=0}^{D-\Delta D-1}\left(\frac{1}{3}\right)^{i} finish^{2}_{\frac{1}{3},\frac{2}{3},0}(k_1,N,\Delta N+1,D,\Delta D+i) & \text{otherwise.}
\end{cases}
\end{aligned}
$$

Because of the repeated calls to $finish^{2}_{\frac{1}{3},\frac{2}{3},0}$ the evaluation may take $\mathcal{O}(n^3)$ steps.

Using the above formulae, the probabilities can be estimated in $\mathcal{O}(n^3)$ steps for this case.

(a) $|\alpha\cap N|=0$

$$\Rightarrow P = finish^{2}_{\frac{1}{3},\frac{2}{3},0}(k_1,N,0,D,0)$$

(b) $|\alpha\cap N|=1$

$$\Rightarrow P = finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}{}^{1}(k_1,N,0,D,0)$$

(c) $|\alpha \cap N| = 2$

$$\Rightarrow P = \frac{1}{3} + \frac{2}{3} \cdot finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}(k_1, N, 1, D, 0)$$

Case 2 If $|\alpha \cap L| = 0$, then the first machine to finish is assigned to L. After that all machines are tried to be assigned to D, until $D = \emptyset$.

From the diagram for this case (Figure 9.5), we can observe that this case has the same building block functions as the previous case.

$$finishL^2(N, \Delta N) = finishL^1(N, \Delta N)$$

$$finish^{2}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}(N, \Delta N, D, \Delta D) = finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}(N, \Delta N, D, \Delta D)$$

$$finish^{2}_{\frac{1}{3},\frac{2}{3},0}(k_1, N, \Delta N, D, \Delta D) = finish^{1}_{\frac{1}{3},\frac{2}{3},0}(k_1, N, \Delta N, D, \Delta D)$$

$$finish^{2}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}(k_1, N, \Delta N, D, \Delta D) \qquad = \qquad finish^{1}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}(k_1, N, \Delta N, D, \Delta D)$$

The estimated probabilities can then be calculated as follows. Each case needs $\mathcal{O}(n^3)$ steps.

(a) $|\alpha \cap N| = 0$

$$\Rightarrow P = finish^{2}_{\frac{1}{3},\frac{2}{3},0}(k_1, N, 0, D, 1)$$

(b) $|\alpha \cap N| = 1$

$$\Rightarrow P = \frac{2}{3} \cdot finish^{2}_{\frac{1}{3},\frac{2}{3},0}(k_1, N, 0, D, 1) + \frac{1}{3} \cdot \begin{cases} 1 & \text{if } |N| = 1, \\ finish^{2}_{\frac{1}{3},\frac{2}{3},0}(k_1, N, 1, D, 0) & \text{otherwise.} \end{cases}$$

(c) $|\alpha \cap N| = 2$

$$\Rightarrow P = \frac{2}{3} \cdot finish^{2}_{\frac{1}{3},\frac{2}{3},0}(k_1, N, 1, D, 0) + \frac{1}{3} \cdot \begin{cases} finishL^2(N, 0) & \text{if } |D| = 1, \\ \frac{1}{3} + \frac{2}{3} \cdot finish^{2}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}(k_1, N, 1, D, 1) & \text{otherwise} \end{cases}$$

(d) $|\alpha \cap N| = 3$

$$\Rightarrow P = \frac{1}{3} + \frac{2}{3} \cdot finish^{2}_{\frac{1}{3},\frac{1}{3},\frac{1}{3}}(k_1, N, 2, D, 1)$$

9.2.3 Usage in an Algorithm

Evaluating the above formulae for a tree B and the heaviest two-leaves-subtree (or any other subtree with a common leaf with B) takes $\mathcal{O}(n^3)$ steps (including all binomials and powers).

The above formulae can be used to select a schedule that minimizes the *worst case* probability of reaching a heavy subtree.

9.3 Performance of Resulting Algorithms

The formulae from the previous sections can be used to approximate probabilities of reaching two-leaves-subtrees under different assumptions (either trying to avoid or trying to reach a subtree).

The formulae need a concrete subtree of B (e.g. defined by two nodes) for evaluation to bound l_1, l_2, l_3, k_1, k_2, and k_3. The results for each concrete subtree can be combined to results for a class of similar subtrees. The results for classes can be combined to a total result which is then minimized or maximized.

Hence there are three questions that need to be answered for a specific algorithm based on the above formulae:

1. Which classes of trees are taken into account (with respect to their weight $w(k, l, a, n)$)?
   a) For avoiding subtrees
   b) For trying to reach subtrees
2. How are the results for a class of subtrees combined?
   a) For avoiding subtrees
   b) For trying to reach subtrees
3. How are the results for all classes of subtrees combined and what value is minimized/maximized?

Intuitively the question 2 should be answered such that we add the probabilities, if we are trying to reach the subtrees of a given class (because we can reach subtree one or subtree two or … to reach the class), and that we multiply the probabilities, if we are trying to avoid the subtrees of a given class (because we do not want to reach subtree one nor subtree two, nor …). Empirically this was verified by the fact that if we were trying to avoid (reach) the heaviest (lightest) two-leaves-subtree and minimize (maximize) the probability that it is reached, then taking the product (sum) of all probabilities calculated for the representatives of a class performs best among the operations minimum, maximum, sum, and product.

The answer to question 1 is not as straight forward and is closely related to question 3. If we only choose to look at the class of heaviest subtrees, we need not combine any class results. If we look at all classes of subtrees we can either sum up the results, compare them in lexicographical order or weight them in another way.

Figure 9.6 shows an overview of the performance of selected algorithms. The first column describes an algorithm (or a question), the remaining columns give the number of non-optimal (or can-optimal) solutions found for each set of trees with k nodes . The first two lines give the number of trees and the number of trees with more than three leaves (the only ones where an algorithm is needed). The next two lines give the performance of an algorithm that just chooses the leaves with the highest or lowest DFS number (e.g. an arbitrary algorithm). The fifth line gives the performance of HLF for comparison (note, that there are only eight non-optimal trees, the remaining ones are at least can-optimal).

The algorithms 1 – 3 choose a single subtree-class and take the solution based on the probability of reaching the subtree (either while avoiding it or while trying to reach it). Interestingly, the algorithm avoiding the lightest tree performs better than the one avoiding the heaviest tree.

The algorithms 4 – 7 approximate probabilities for all classes of subtrees and decide on a sum weighted by the subtree weight $w(k, l, a, n)$. All four perform worse than the DFS-choosing algorithms!

The algorithms 8 and 9 are based on the idea to calculate all probabilities and to compare them as a vector in lexicographical order (where they are ordered by subtree weight $w(k, l, a, n)$ either ascending for algorithm 9 or descending for algorithm 8).

Figure 9.6 is by no means complete. There are many other possible ways to combine the approximated probabilities. We only give canonical and interesting combinations, others were tried but had equally disappointing results. Even combining probabilities for trying to avoid and trying to reach a tree (e.g. try to avoid heaviest while reaching lightest tree) did not enhance the results.

Algorithm	Failures on Trees With k Nodes								
(or Question)	4	5	6	7	8	9	10	11	12
Q: Number of trees	4	9	20	48	115	286	719	1842	4766
Q: Trees with more than 4 leaves		1	5	20	67	207	595	1655	4494
Choose leaves by DFS			3	15	56	180	533	n/a	n/a
Choose leaves by DFS (reverse)				1	10	46	175	n/a	n/a
HLF				1	8	33	116	372	1130
1. 1b=lightest, 2b=sum, 3=$\max(p)$				1	4	22	100	416	1568
2. 1a=heaviest, 2a=prod, 3=$\min(p)$			3	15	58	189	566	n/a	n/a
3. 1a=lightest, 2a=prod, 3=$\min(p)$					1	14	102	484	1861
4. 1a=all, 2a=prod, 3=$\min(\sum p \cdot w)$			3	15	58	189	566	n/a	n/a
5. 1a=all, 2a=prod, 3=$\max(\sum p \cdot w)$				1	12	50	188	n/a	n/a
6. 1b=all, 2b=sum, 3=$\max(\sum p \cdot w)$			3	15	58	189	566	n/a	n/a
7. 1b=all, 2b=sum, 3=$\min(\sum p \cdot w)$				1	12	55	199	n/a	n/a
8. 1a=all, 2a=prod, 3=$\min(\bar{p}_>)$			3	15	58	189	566	n/a	n/a
9. 1b=all, 2b=sum, 3=$\max(\bar{p}_<)$					1	8	47	206	785

Figure 9.6: Results for Various Parameterized Algorithms Based on Approximated Subtree Probabilities

One tree that seemed very hard for a lot of algorithms is shown in Figure 9.7. There are basically two scheduling alternatives, either scheduling nodes 2 (or 4), 6, and 7, or scheduling nodes 2, 4, and 6 (or 7). The latter is optimal with an expected value of $1471/324$, the former has the expected value of $1472/324$.

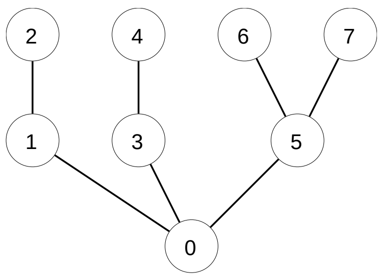


Figure 9.7: Counter Example for “Heavy-Tree-Avoidance” Strategy.

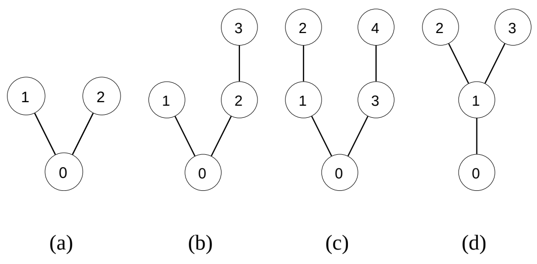


Figure 9.8: Two-Leaves-Subtrees of Tree in Figure 9.7

The two-leaves-subtrees of the tree in Figure 9.7 are shown in Figure 9.8. The table in Figure 9.9 gives the weight of each subtree and the probability of reaching it under the two different schedules. Obviously the probability of reaching the heaviest subtree is higher under the optimal schedule, which is compensated by the lower probability of reaching the second heaviest subtree. To no surprise, the same tree appears again

		Schedule 2,7,8	Schedule 2,4,7
Subtree	Weight	Probability	Probability
a	50/12	2/9	2/9
b	55/12	38/81	40/81
c	57/12	8/27	7/27
d	58/12	1/81	2/81

Figure 9.9: Weights and Probabilities for Subtrees in Figure 9.8

in section 10.1 as one of the trees having a smallest difference between two schedules.

Another reason for the failure of this approach might lie in the fact that each class of two-leaves-subtrees (e.g. $(0|2|2)$) is represented multiple times. Furthermore, some nodes take part in more instances than others and some of these nodes are no leaves. The further down a node is in the tree, the harder it seems to predict the success of schedules with respect to that node.

The formulae of sections 9.1 and 9.2 are inexact in multiple ways. Using a reasonable algorithm, the probability of being always left with the largest two-leaves- or one-leaf-subtree is rather small. On the other hand, the leaves cannot be scheduled in arbitrary order as the binomials in the formulae suggest. All this seems to lead to algorithms that perform hardly better than the DFS controlled selection of leaves.

Chapter 10

A Time-Based Approach

10.1 The Computation Tree and the Size of Numbers

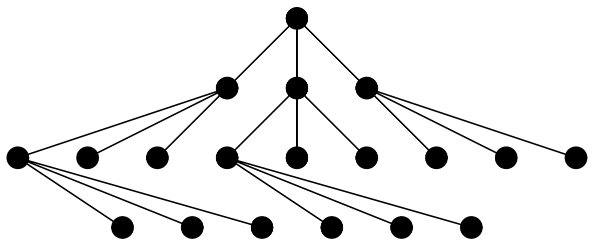
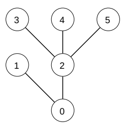
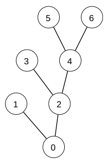


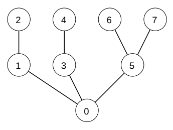
Figure 10.1: Example for a “Computation Tree”

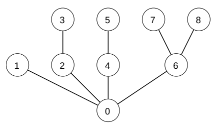
Let the computation tree $CT_\alpha(B)$ of a tree B be the tree of possible states under the schedule α, where a state is defined by a subtree, and where the leaves are the empty subtrees. $CT_\alpha(B)$ has depth $|B| + 1$, it is essentially the DAG of subtrees converted to a tree by ignoring isomorphism between subtrees, not merging nodes, and selecting only reached subtrees. We can label each edge with the inverse of the number of nodes scheduled in the tree represented by its parent (the expected length of the corresponding time interval in the execution of the schedule). Let each node be labeled additionally with the sum of the edge labels and the product of the edge labels from the root to that node. The expected processing time of a path represented by a leaf is the sum of the edge labels. The probability of reaching the leaf is the product of the edge labels. The total expected processing time is then the sum of the products of both labels of all leaves.
Let $CT^2_\alpha(B)$ be $CT_\alpha(B)$ pruned at the nodes representing two-leaves-subtrees of B. All edges in $CT^2_\alpha(B)$ are labeled with $1/3$. Let the leaves be labeled with the weight $w(k, l, a, |B|)$ of the represented subtree with respect to B (as defined in section 8.3). The depth of all leaves representing isomorphic subtrees is the same, and since all edges are labeled with $1/3$, the probabilities of reaching these trees are all the same ($\left(\frac{1}{3}\right)^{\text{depth}}$). Let there be $occ(a|k|l)$ leaves representing the class of two-leaves-subtrees $(a|k|l)$. $CT^2_\alpha(B)$ is clearly a DAG, in which the leaves obviously form a chain. By Theorem 8.1, the total expected processing time can be expressed as

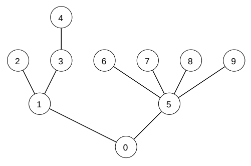
$$\mathbb{E}(T_\alpha(B)) = \sum_{(a|k|l) \text{ subtree class in } B} occ(a|k|l) \cdot \left(\frac{1}{3}\right)^{|B|-k-l-a-1} \cdot w(k, l, a, |B|) \tag{10.1}$$

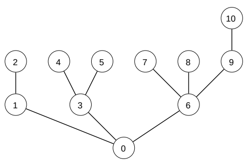

(a) 6 Nodes

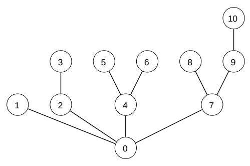

(b) 7 Nodes

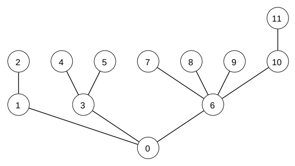

(c) 8 Nodes

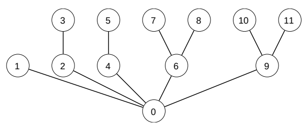

(d) 9 Nodes

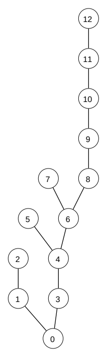

(e) 10 Nodes

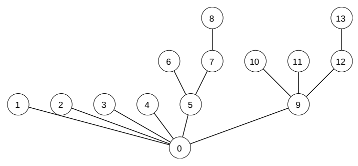

(f) 11 Nodes – relative

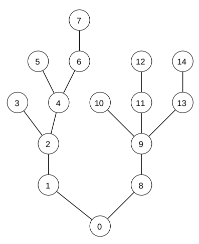

(g) 11 Nodes – absolute

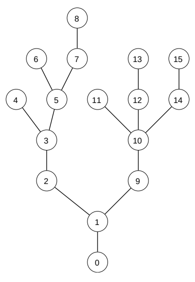

(h) 12 Nodes – relative

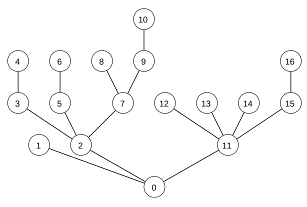

(i) 12 Nodes – absolute


(j) 13 Nodes


(k) 14 Nodes


(l) 15 Nodes


(m) 16 Nodes


(n) 17 Nodes

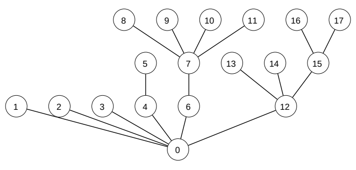

(o) 18 Nodes

Figure 10.2: Trees with Minimal Expected Processing Time Differences for Different Schedules

Tree	3^{-n-3}	Best Time and Schedule	2nd Best Time and Schedule	Absolute Difference	Relative Difference
Figure 10.2 (a)	$3.7037 \cdot 10^{-2}$	$\{3,4,5\}$ 4.0	$\{1,4,5\}$ 4.0555555555555553582	$5.5555555555555358183 \cdot 10^{-2}$	$1.3888888888888839546 \cdot 10^{-2}$
Figure 10.2 (b)	$1.2345 \cdot 10^{-2}$	$\{3,5,6\}$ 4.7592592592592595224	$\{1,5,6\}$ 4.7870370370370372015	$2.7777777777777679091 \cdot 10^{-2}$	$5.8365758754863605873 \cdot 10^{-3}$
Figure 10.2 (c)	$4.1152 \cdot 10^{-3}$	$\{2,4,7\}$ 4.5401234567901234129	$\{4,6,7\}$ 4.5432098765432096243	$3.0864197530862114149 \cdot 10^{-3}$	$6.798096532970309162 \cdot 10^{-4}$
Figure 10.2 (d)	$1.3717 \cdot 10^{-3}$	$\{3,5,8\}$ 4.747942386831275563	$\{5,7,8\}$ 4.7489711934156382256	$1.0288065843626625906 \cdot 10^{-3}$	$2.1668472372708734788 \cdot 10^{-4}$
Figure 10.2 (e)	$4.5725 \cdot 10^{-4}$	$\{4,8,9\}$ 5.4478737997256514447	$\{2,4,9\}$ 5.4481310013717427765	$2.5720164609133178146 \cdot 10^{-4}$	$4.7211381090414419133 \cdot 10^{-5}$
Figure 10.2 (f)	$1.5242 \cdot 10^{-4}$	$\{2,5,10\}$ 5.5891632373113848686	$\{4,5,10\}$ 5.5892775491540920285	n/a	$2.0452407248378835569 \cdot 10^{-5}$
Figure 10.2 (g)	$1.5242 \cdot 10^{-4}$	$\{3,6,10\}$ 5.4847584209724127291	$\{5,6,10\}$ 5.4848727328151190008	$1.1431184270627170463 \cdot 10^{-4}$	n/a
Figure 10.2 (h)	$5.0805 \cdot 10^{-5}$	$\{2,5,11\}$ 5.8988721231519578581	$\{4,5,11\}$ 5.8989102270995275035	n/a	$6.4595310381614519832 \cdot 10^{-6}$
Figure 10.2 (i)	$5.0805 \cdot 10^{-5}$	$\{3,5,8\}$ 5.6648757811309247145	$\{5,10,11\}$ 5.6649138850784934718	$3.8103947568757234876 \cdot 10^{-5}$	n/a
Figure 10.2 (j)	$1.6935 \cdot 10^{-5}$	$\{5,7,12\}$ 9.0803743236549312456	$\{2,7,12\}$ 9.0803844450160031698	$1.0121361071924184216 \cdot 10^{-5}$	$1.1146413915511630655 \cdot 10^{-6}$
Figure 10.2 (k)	$5.6450 \cdot 10^{-6}$	$\{8,11,13\}$ 6.3338018707626995152	$\{6,8,13\}$ 6.3338039876486753599	$2.1168859758446956221 \cdot 10^{-6}$	$3.3422042858909098194 \cdot 10^{-7}$
Figure 10.2 (l)	$1.8817 \cdot 10^{-6}$	$\{7,12,14\}$ 7.744037393732323693	$\{5,7,12\}$ 7.744037459885010577	$6.6152686883924616268 \cdot 10^{-8}$	$8.5424028217458808394 \cdot 10^{-9}$
Figure 10.2 (m)	$6.2723 \cdot 10^{-7}$	$\{8,13,15\}$ 8.7440373937323254694	$\{6,8,13\}$ 8.7440374598850123533	$6.6152686883924616268 \cdot 10^{-8}$	$7.5654624866245974647 \cdot 10^{-9}$
Figure 10.2 (n)	$2.0908 \cdot 10^{-7}$	$\{4,6,10\}$ 7.6064558305730187726	$\{6,10,16\}$ 7.6064558763082095183	$4.573519074568821452 \cdot 10^{-8}$	$6.0126807759617052044 \cdot 10^{-9}$
Figure 10.2 (o)	$6.9692 \cdot 10^{-8}$	$\{9,10,11\}$ 7.6668007361118162279	$\{11,16,17\}$ 7.6668007361118171161	$8.8817841970012523234 \cdot 10^{-16}$	$1.1584733323205696524 \cdot 10^{-16}$

Figure 10.3: Minimal Expected Processing Times for Different Schedules

If we can calculate bounds on $occ(a|k|l)$ or on the $\mathcal{O}(n^2)$ two-leaves-trees individually, we can also bound the optimal value of the expected total processing time for a given tree B. The formulae in sections 9.1 and 9.2 can be interpreted just as such.

From equation 10.1 we can also see that the number representing the total expected processing time for a tree B can grow quite large if expressed as an exact fraction. An upper bound for the total expected processing time is $n = |B|$. The smallest possible occurring two-leaves-subtree is $(0, 1, 1)$. Its last factor in the upper sum is therefore $\left(\frac{1}{3}\right)^{n-3}$. The resulting total expected processing time might hence be a fraction with a maximal denominator of 3^{n-3}, hence the maximal numerator is $n \cdot 3^{n-3}$. Using this upper bound, an upper bound on the size of the occurring numbers is

$$|\log_2(3^{n-3})| + |\log_2(n \cdot 3^{n-3})| = (n-3)\log_2(3) + (n-3)\log_2(3) + \log_2(n) = \mathcal{O}(n).$$

If numbers of this size occur, the unit cost model might no longer be appropriate. If the input size stays below twenty nodes, then 3^n fits into 32 bits and the results are exact to some extend. Above that size each arithmetic operation may take $\mathcal{O}(n)$ (multiplication and division even $\mathcal{O}(n \log_2(n))$). And this additional factor must be considered, unless some smaller bound on the numbers can be obtained. The result of Papadimitriou and Tsitsiklis (in [PT87], see Theorem 7.1 in this paper) can also be read as “the difference between the optimal strategy and another (namely HLF) can become arbitrarily small”. If an optimal algorithm considers such other solutions it might well be faced with very small differences (as low as $3^{-(n-3)}$). Figure 10.2 and Figure 10.3 support this hypothesis. It seems that we are already dealing with rounding errors in the trees with 18 nodes. The IEEE 754 double precision values have a fractional part of 52 bits, which suffices for exact values of sizes below 16 decimal digits.

10.2 Motivation

If we look at a concrete two-leaves-subtree S defined by its two leaves i, j, then there are four important decision points during the scheduling of B. At decision point t_1 the first leaf is scheduled, at decision point t_2 the second leaf is scheduled, at decision point t_3 one of the scheduled leaves is finished, and at decision point t_4 the remaining subtree is S (S is reached). See Figure 10.4 for a schematic view of this. Of course not all these points can (or do) occur for every subtree. If the subtree is reached, it will at least go through t_1, t_2, and t_4 ($t_1 = t_2$ only if both are 0). If the tree is not reached (i.e. a leaf of S is finished while there are still three leaves left in B), then the point t_1 might occur, if t_1 occurs, t_2 might occur, and if t_1 occurs, t_3 might occur.

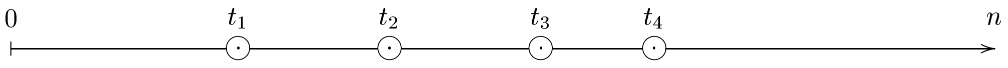


Figure 10.4: Schematic view of the Time Line of a Concrete Schedule.

Suppose, only t_1 and t_3 occur. What is the expected distance between them? The probability that $t_3 - t_1 = k$ is $\frac{1}{3} \cdot \left(\frac{2}{3}\right)^{k-1}$. Hence the expected value of k is

$$\sum_{k \geq 1} k \cdot \frac{1}{3} \cdot \left(\frac{2}{3}\right)^{k-1} = \frac{1}{3} \cdot \sum_{k \geq 0} k \cdot \left(\frac{2}{3}\right)^{k-1} = \frac{1}{3} \cdot \frac{1}{\left(1 - \frac{2}{3}\right)^2} = 3.$$

If t_1, t_2, and t_3 occur, the expected distance between t_2 and t_3 is

$$\sum_{k \geq 1} k \cdot \frac{2}{3} \cdot \left(\frac{1}{3}\right)^{k-1} = \frac{2}{3} \cdot \frac{1}{\left(1 - \frac{1}{3}\right)^2} = \frac{3}{2}.$$

The expected distance between t_1 and t_3 in this case depends upon the distance between t_1 and t_2, which in turn depends on the scheduling strategy. These values are no surprise: the expected processing time of a task is $\frac{1}{\lambda}$. With three (or two) machines every $\frac{1}{3\lambda}$ ($\frac{1}{2\lambda}$) a decision point occurs. Dividing gives the above fractions.

10.3 Total Expected Processing Time

In section 10.1 we defined the computation tree $CT^2_\alpha(B)$ for tree B and strategy α and derived the equation

$$\mathbb{E}(T_\alpha(B)) = \sum_{(a|k|l) \text{ subtree in } B} occ(a|k|l) \cdot \left(\frac{1}{3}\right)^{|B|-k-l-a-1} \cdot w(k,l,a,|B|)$$

Obviously, there can be multiple concrete subtrees in the class of two-leaves-subtrees $(a|k|l)$. Each concrete two-leaves-subtree can be uniquely identified by its two leaves. Let $occ(l_1, l_2)$ be the number of times that a two-leaves-subtree with leaves l_1 and l_2 occurs in $CT^2(B)$. Let $size(l_1, l_2)$ be the number of nodes in the subtree. For a given tree B there is also an easy mapping M between two nodes n_1, n_2 (that are no ancestors) and the class $(a|k|l)$ of two-leaves-subtrees that the such-defined subtree belongs to. The above equation can be rewritten as

$$\mathbb{E}(T_\alpha(B)) = \sum_{n_1,n_2 \in B, \nexists path(n_1,n_2)} occ(n_1,n_2) \cdot \left(\frac{1}{3}\right)^{|B|-size(n_1,n_2)} \cdot w(M(n_1,n_2),|B|)$$

For a given strategy α, a tree B with nodes $\{1,\ldots,n\}$, let t_i be the order number of node i, i.e. node i is scheduled as t_i-th node. t_i does not depend on the absolute processing lengths of the tasks, but rather on the finishing order of the tasks (starting with a fixed schedule, each order defines a unique resulting subtree that determines the next node to be scheduled by α).

For nodes i,j and fixed t_i, t_j we can calculate the probability $P[i,j]$ that the two-leaves-subtree will not occur (without loss of generality $t_i < t_j$):

$$\begin{aligned}
P[i,j] &= P_{i \text{ is finished before } j \text{ is scheduled}} + (1 - P_{i \text{ is finished before } j \text{ is scheduled}}) \cdot P_{j \text{ is finished before there are no other leaves left}} \\
&= \left(\sum_{k=1}^{t_2-t_1} \left(\frac{2}{3}\right)^{k-1} \cdot \frac{1}{3}\right) + \left(1 - \sum_{k=1}^{t_2-t_1} \left(\frac{2}{3}\right)^{k-1} \cdot \frac{1}{3}\right) \cdot \left(\sum_{k=1}^{|B|-size(i,j)-t_2} \left(\frac{1}{3}\right)^{k-1} \frac{2}{3}\right) \\
&= \left(1 - \left(\frac{2}{3}\right)^{t_2-t_1}\right) + \left(\left(\frac{2}{3}\right)^{t_2-t_1}\right) \cdot \left(1 - \left(\frac{1}{3}\right)^{|B|-size(i,j)-t_2}\right) \\
&= 1 - \left(\frac{2}{3}\right)^{t_2-t_1} + \left(\frac{2}{3}\right)^{t_2-t_1} - \left(\frac{2}{3}\right)^{t_2-t_1} \cdot \left(\frac{1}{3}\right)^{|B|-size(i,j)-t_2} \\
&= 1 - \left(\frac{2}{3}\right)^{t_2-t_1} \cdot \left(\frac{1}{3}\right)^{|B|-size(i,j)-t_2} \\
&= 1 - 2^{t_2} \cdot \left(\frac{3}{2}\right)^{t_1} \cdot \left(\frac{1}{3}\right)^{|B|-size(i,j)}
\end{aligned}$$

Hence, the probability that a subtree will occur under a given scheduling order is $2^{t_2} \cdot \left(\frac{3}{2}\right)^{t_1} \cdot \left(\frac{1}{3}\right)^{|B|-size(i,j)}$ – the later the first (or the second) node is scheduled, the higher the probability that the tree will occur. One can also see from this equation that the probability that a two-leaves-subtree occurs grows with its size by the factor three per node. Also, if the node of a subtree is scheduled, then the probability decreases at least by two-thirds or by one-half (in comparison to not scheduling the node in a given turn). When the probability that a two-leaves-subtree for nodes i,j occurs decreases, the corresponding tree occurs less often and $occ(i,j)$ decreases (the opposite is also true).

10.4 Resulting Algorithms

We will use the above formulae in Algorithm 11. The basic idea is to try to decrease the total expected processing time by reducing the probabilities reaching a subset of subtrees that contributes a largest part. A subtree contributes more if it has a large weight and is relatively large at the same time (the smaller the subtree the smaller is the "general" probability of reaching it – the factor $\left(\frac{1}{3}\right)^{|B|-size(i,j)}$). The algorithm iteratively schedules leaves that belong to such strongly contributing subtrees. After selecting a leaf the remaining subtree weights are adjusted.

In detail, we are only looking at feasible subtrees. A subtree is considered feasible, if it has a leaf in common with B and if it can be replaced by a tree with a smaller weight (that is it can be avoided in favor of something). All these trees are weighted (with the weights from section 8.3) and the weight is discounted by their "general" probability (the size induced factor $\left(\frac{1}{3}\right)^{|B|-size(i,j)}$).

The weight adjustment for a scheduled leaf k replaces the subtrees' (that k is a leaf of) weights by two-thirds of its weight (because the probability of reaching decreases by two-thirds for each time interval the first node is scheduled earlier – if a node is not scheduled at the current decision point it is scheduled the earliest one interval later).

Algorithm 11 A Discounted-Weight-Based Algorithm.

```
 1: Let $S$ be the set of all concrete subtrees that can be avoided in favor of a lighter tree.
 2: Let $\forall i \in B : w_n(i) = 0$ be an initially zero node weight.
 3: **for all** $t \in S$ **do**
 4:    Let $t$ be of type $M(t) = (a|k|l)$.
 5:    Let $S$ be the size of $t$
 6:    Let $w = w(k, l, a, |B|) \cdot 3^s$
 7:    Let $t$'s leaves be $i, j$.
 8:    **if** $i$ and $j$ are scheduled **then**
 9:       $w = 0$
10:    **else**
11:       **if** $i$ is scheduled **then**
12:          $w = \frac{2}{3} \cdot w$
13:       **end if**
14:       **if** $j$ is scheduled **then**
15:          $w = \frac{2}{3} \cdot w$
16:       **end if**
17:    **end if**
18:    Set $w_n(i) = w_n(i) + w$
19:    Set $w_n(j) = w_n(j) + w$
20: **end for**
21: Select node $k$ with highest weight
22: If less than three leaves are selected goto 1
```

The first tree that Algorithm 11 fails on is shown in Figure 10.5 (a) (the optimal schedule is $\alpha_{OPT} = \{5, 6, 7\}$, while Algorithm 11 schedules leaf 2).

Although Algorithm 11 performs worse than the best algorithm so far (see Figure 9.6), it has a better running time (Evaluation of algorithm number 9 takes time $\mathcal{O}(n^7)$, while Algorithm 11 can be evaluated in $\mathcal{O}(n^3)$).

We know that HLF performs asymptotically well, so it seems reasonable to try to keep the properties from Theorem 7.1 while improving the performance. The theorem holds for any HLF strategy, so we will simply adjust the strategy so that tie-breaks are solved in a favorable way. An algorithm based on that idea is given as Algorithm 12. The exchange steps are performed in line 10 based on the comparison of two nodes i, j under the assumption that either i is scheduled before j or vice versa. If one of the nodes is scheduled before the other, then the probabilities of the two-leaves-subtree associated with that node are decreased

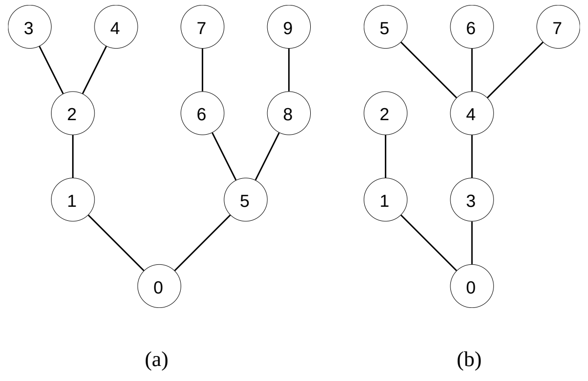


Figure 10.5: First Tree that (a) Algorithm 11 or (b) Algorithm 12 Fails on

by at least a factor $2/3$ (or $1/2$ if another node in that tree is already scheduled from a previous interval). Hence the difference in the contribution is $1/3$ ($1/2$, respectively) of the weight (which we discount by the "general" probability as above). The nodes with the largest contributions are preferred.

Algorithm 12 Improving HLF Through Exchange Steps.

```
1: Let $(a_i)_{1 \le i \le n}$ be an array with all nodes from $B$.
2: Sort $(a_i)_{1 \le i \le n}$ by the height of the leaves, s.t. $\forall i < j \in leaves(B) : height(a_i) \ge height(a_j)$ and
   $\forall a_i \in leaves(B), a_j \in B \setminus leaves(B) : i < j$.
3: $start \longleftarrow 1$
4: $end \longleftarrow 1$
5: while $end \le 3 \wedge a_{start} \in leaves(B)$ do
6:    while $height(a_{start}) = height(a_{end+1})$ do
7:       $end \longleftarrow end + 1$
8:    end while
9:    $end \longleftarrow end + 1$          $/*$ nodes in interval $[start, \ldots, end)$ all have the same height $*/$
10:   Sort $(a_i)_{start \le i \le end}$ by the minimal difference in the contribution of their subtrees to the total
      expected processing time when preferring the one over the other.
11: end while
12: Schedule leaves $a_1, a_2, a_3$.
```

The algorithm can collect the two-leaves-subtrees for each node at the beginning and calculate all the contributions of all nodes of an interval before sorting. Algorithm 12 can thus be evaluated in $\mathcal{O}(n^3)$.

As can be seen in Figure 9.6, Algorithm 12 performs better than any algorithm seen so far (while still maintaining the HLF property). The first tree that Algorithm 12 fails on is shown in Figure 10.5 (b) (the optimal schedule is $\alpha_{OPT} = \{3, 4, 9\}$, while Algorithm 12 schedules leaf $\{4, 7, 9\}$).

Algorithm	Failures on Trees With k Nodes								
(or Question)	4	5	6	7	8	9	10	11	12
Q: Number of trees	4	9	20	48	115	286	719	1842	4766
Q: Trees with more than 4 leaves		1	5	20	67	207	595	1655	4494
HLF				1	8	33	116	372	1130
Algorithm 11					1	10	55	232	847
Algorithm 12							1	8	31

Figure 10.6: Comparison of Previous Algorithms to the New Time-Based Ones

Chapter 11

Conclusion and Outlook

In this work we have dealt with the specific problem of scheduling tasks with independent identically, exponentially distributed processing times and in-tree constraints on three processors in parallel. From [CR75] we knew that an HLF strategy is not optimal, although it is optimal in the deterministic case [Hu61]. On the other hand, the HLF strategy can serve as an – asymptotically optimal – approximation. An algorithm would therefore need to fill the gap between the HLF approximation and the optimal solution while still being of reasonable complexity (the amount of time saved by an optimal solution compared to the time needed to calculate the optimal solution).

The only optimal algorithm found is described in chapter 6. The running time of this algorithm could be enhanced over the naive recursive version by a dynamic programming approach and the elimination of redundancies and isomorphic trees. This led to an improvement of the main term in the asymptotic running time from about 3^n to approximately 1.6^n. The algorithm still has exponential running time and its use for larger trees is hence prohibitive. The algorithm never the less served well in generating a large amount of solutions for trees with eighteen nodes or less, which were used to check against other strategies.

Figure 11.1 gives an overview of the tested strategies and their performance. There are four categories of algorithms shown in the figure. The first pair is selected from the HLF strategies, the second from Monte Carlo techniques, the third from the algorithms based on combinatorial estimation of the probabilities of reaching two-leaves-subtrees (chapter 9), and the last pair is based on node weights calculated from a time based view of the relation between scheduled nodes and two-leaves-subtrees (chapter 10). By means of an example we have shown that no static list scheduling strategy can be optimal (see section 7.4). The HLF strategy is not optimal either and we have found a smaller example that confirms this. On the other hand, it was shown by Papadimitriou and Tsitsiklis [PT87] that the HLF strategy is asymptotically optimal. The strategies can therefore be divided into strategies which may be made optimal but have no proven bound (3, 4, 5, 6, 8 in Figure 11.1) and strategies based on HLF that cannot be optimal but have a proven bound (1, 2, 7 in Figure 11.1). An apparent precondition for an optimal strategy is that the strategy should at least be able to generate the optimal initial assignments. The intermediate steps (that lead to the exclusion of the static list scheduling strategies) were not checked since no strategy was found that was optimal even in the first step.

We were able to break down the problem structure yielding new approaches to algorithms. One of the difficulties of having to deal differently with steps whether there are one, two, or three machines working in parallel could be eliminated. For all (sub-)trees with less than three leaves the expected processing time can be calculated in $\mathcal{O}(n^2)$ (see chapter 8). In the calculation of the total processing time of a tree with three or more leaves the processing times of its two-leaves-subtrees can be used in various ways as a basis (see chapters 8, 9, and 10). This same approach might be extended to the k-machines version of the problem. The time to calculate the expected processing times of the $(k-1)$-leaves-subtrees should then be of the order $\mathcal{O}(n^{k-1})$.

For practical purposes the algorithms developed in this thesis can be used as heuristics. Most algorithms presented in this work seem to improve over 'naked' HLF. Figure 11.1 shows the number of trees a strategy fails on ordered by the size of the trees. Since non-optimality of solutions in smaller trees is inherited by solutions to larger trees (e.g. in the intermediate steps) this quantitative improvement is also a qualitative improvement. The best performing algorithms are lexicographical extensions of HLF. Hence, they enjoy the property of asymptotic optimality proven by Papadimitriou and Tsitsiklis. The algorithms from chapter 9 make the heaviest use of the two-leaves-subtree problem structure. They have better results than the basic HLF strategy, but they are also quite complicated with a running time of $\mathcal{O}(n^7)$. They are on the other hand not based on something definitely non-optimal. The algorithm with the number 5 in Figure 11.1, based on a combinatorial estimation of the probabilities of reaching two-leaves-subtrees, generates correct solutions for all trees shown in Figure 7.3 on page 37 (which are the counter-examples to HLF). For a real-world problem we would select the algorithm in line 7 of Figure 11.1 with a running time of $\mathcal{O}(n^3)$ since it stays on the safe side by preselecting with HLF while successfully using the two-leaves-subtree problem structure. If time requirements are very tight, a simpler HLF-variant still seems the best choice to be made. The following considerations seem to argue against the use of a Monte Carlo technique.

Algorithm	Failures on Trees With k Nodes								
(or Question)	6	7	8	9	10	11	12	13	14
Q: Number of trees	20	48	115	286	719	1842	4766	12486	32973
Q: Trees with more than 4 leaves	5	20	67	207	595	1655	4494	n/a	n/a
1. HLF		1	8	33	116	372	1130	3352	9613
2. Best tie-breaking method for HLF (see Figure 7.4)						11	58	250	976
3. MC100 (HLF) rounded average (see Figure 7.7)		1	6	35	152	522	n/a	n/a	n/a
4. MC100 (random) rounded average (see Figure 7.7)		1	7	31	133	422	n/a	n/a	n/a
5. Two-Leaves-Tree/combinatorial-based best algorithm (see Figure 9.6, 9.)			1	8	47	206	785	n/a	n/a
6. Two-Leaves-Tree/combinatorial-based 2nd best algorithm (see Figure 9.6, 3.)			1	14	102	484	1861	n/a	n/a
7. Two-Leaves-Tree/node-weight-based best algorithm (HLF) (see Figure 10.6, Algorithm 12)					1	8	31	n/a	n/a
8. Two-Leaves-Tree/node-weight-based 2nd best algorithm (see Figure 10.6, Algorithm 11)			1	10	55	232	847	n/a	n/a

Figure 11.1: Overview of Results

The results obtained in this work also indicate that calculating an optimal solution is a difficult problem. The algorithm described in section 5.3 that calculates the solution value of a tree under a given scheduling strategy itself has exponential running time. If the size of numbers and their differences are as small as indicated in section 10.1, then their representation size is linear. Unless another way of calculating the value of a solution is found, it is unclear whether the problem belongs to NP at all. An alternating Turing machine (ATM) could be used to verify that the solution lies below a given value if it can guess a division of the solution to check at each universal node. The ATM's computation tree corresponds to the one in section 10.1. If a number is linear in size, then there are exponentially many divisions of it. Hence this construction does not even lead to a polynomial algorithm for an ATM.

Why does a third machine increase the complexity so much? Pinedo and Weiss remark in their paper [PW85] that the makespan under HLF with two machines is actually distributed independently from the in-tree constraints. This is clearly not the case for the three machines problem version. As shown in chapters

8, 9, and 10 the third machine leads to a problem structure, where the processing priority of a node is not only dependent of its children, but also depends on other nodes for which there are no precedence constraints.

A similar phenomenon occurs with the scheduling of tasks with (deterministic) unit execution time and arbitrary precedence constraints. For the problem $(P2|prec, p_i = 1|C_{max})$ a polynomial algorithm exists (first polynomial algorithm by Fujii et al. [FKN69, FKN71]). The general problem $(Pm|prec, p_i = 1|C_{max})$ with m machines is NP-hard as shown by Ullman [Ull75]. Whether the three machines problem is NP-hard or solvable in polynomial time is still an open question [GJ79]. An often observed phenomenon is that the transition from two to three in a question suffices to turn a tractable into an intractable problem (e.g. NODE-COLORING, KNF-SAT). If we look at graphs for the representation of problem structures, we can observe that in graphs with a maximal degree of two only circles and paths are possible, while any graph can be polynomial reduced one with a maximal degree of three.

The reasons why the algorithms developed in chapter 9 fail are discussed in section 9.3. That Algorithm 12 from chapter 10 fails is no surprise. We have already shown in section 7.4 that no static list scheduling policy can be optimal. This rules out any combination of static orders. All modified HLF algorithms are lexicographical orders and hence combinations of static orders. We could refine such orders infinitely many times and still not get an optimal algorithm.

A problem that arose repeatedly in searching for an optimal algorithm was the recursive dependency between the scheduling decisions. For deciding between two alternatives one needs to be able to evaluate these. This in turn requires the knowledge of the applied strategy which we are trying to identify. As an example for that, the algorithms in chapter 9 are inexact because of the estimation of the probability of reaching a two-leaves-subtree depends on how nodes are scheduled in the process. Similarly, for an optimal algorithm based on the ideas of chapter 10 it seems necessary to include information about inner nodes (e.g., one wishes to know the expected time of the scheduling of an inner node). This information influences the current scheduling decision, but the current scheduling decision influences the information needed.

What could be done next? To be able to classify the problem it seems very helpful to get a proven bound on the size of numbers and particularly on the solution difference of two schedules. This might rule out Monte Carlo techniques (see section 7.5). After all it seems that any practical method to establish the membership of the problem in a specific complexity class must tackle the problem of calculating the value of a solution first. On the other hand, the structure of the problem as uncovered in this work could be used in other heuristic techniques (e.g. genetic algorithms) in order to result in better schedules. The best performing HLF variants here can always be used as a cross check on the quality of a solution.

Other authors have extended the problem to different distributions [PW85, Fro88, PT87]. The exponential distribution allows to focus on trees and their structure. The results are independent of the parameter of the exponential distribution. Applying another distribution might either complicate the problem too much or make it easier. For the geometric distribution there exists a dependency between the distribution parameter and the effect of multiple machines working in parallel. The expected time of the first machine to finish with multiple machines decreases with a larger ‘event-probability’ p. This might reduce the interdependencies between tasks. Thus the problem may be easier and its examination might lead to further interesting results.

Bibliography

[Bru85] John Bruno. On Scheduling Tasks with Exponential Service Times and In-Tree Precedence Constraints. *Acta Informatica*, 22:139–148, 1985.

[Bru95] Peter Bruckner. *Scheduling Algorithms*. Springer, 1995.

[CR75] K. M. Chandy and P. F. Reynolds. Scheduling Partially Ordered Tasks with Probabilistic Execution Times. In *Proceedings of the Fifth Symposium on Operating System Principles*, pages 169–177. Operating System Reviews, 1975.

[FKN69] M. Fujii, T. Kasami, and K. Ninomiya. Optimal Sequencing of Two Equivalent Processors. *SIAM Journal of Applied Mathematics*, 17(4):784–789, 1969.

[FKN71] M. Fujii, T. Kasami, and K. Ninomiya. Erratum: Optimal Sequencing of Two Equivalent Processors. In *SIAM Journal of Applied Mathematics* [FKN69], page 141.

[Fro88] Esther Frostig. A Stochastic Scheduling Problem with Intree Precendence Constraints. *Operations Research*, 36(6):937–943, 1988.

[GJ79] Micheal R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freemann, San Francisco, 1979.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[Hu61] T.C. Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 9:841–848, 1961.

[Knu97] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, 3rd edition, Sep 1997.

[LLR82] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Recent Developments in Deterministic Sequencing and Scheduling: A Survey. In M. A. H. Dempster, J.K. Lenstra, and A.H.G. Rinnooy Kan, editors, *Deterministic and Stochastic Scheduling*, pages 35–73. Reidel, 1982.

[Onl] Online Encyclopedia of Integer Sequences. http://www.research.att.com/∼njas/sequences/. AT&T Labs Research.

[Pin95] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1995.

[PT87] Christos H. Papadimitriou and John N. Tsitsiklis. On Stochastic Scheduling With In-Tree Precedence Constraints. *SIAM Journal of Computing*, 16(1):1–6, Feb 1987.

[PW85] Michael Pinedo and Gideon Weiss. Scheduling Jobs with Exponentially Distributed Processing Times and Intree Precedence Constraints on Two Parallel Machines. *Operations Research*, 33:1381–1388, 1985.

[SS01] Thomas Schickinger and Angelika Steger. *Diskrete Strukturen*, volume 2. Springer, 2001.

[Ull75] J. D. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[Ull76] J. D. Ullman. Complexity of sequencing problems. In E.G. Coffman Jr., editor, *Computer and Job-Shop Scheduling Theory*, volume 10, pages 139–164. Wiley, New York, 1976.

[Urq95] Alasdair Urquhart. The Complexity of Propositional Proofs. *The Bulletin of Symbolic Logic*, 1(4):425–467, Dec 1995.