Chapter 1

TODO-Lists

1.1 Strict TODOs

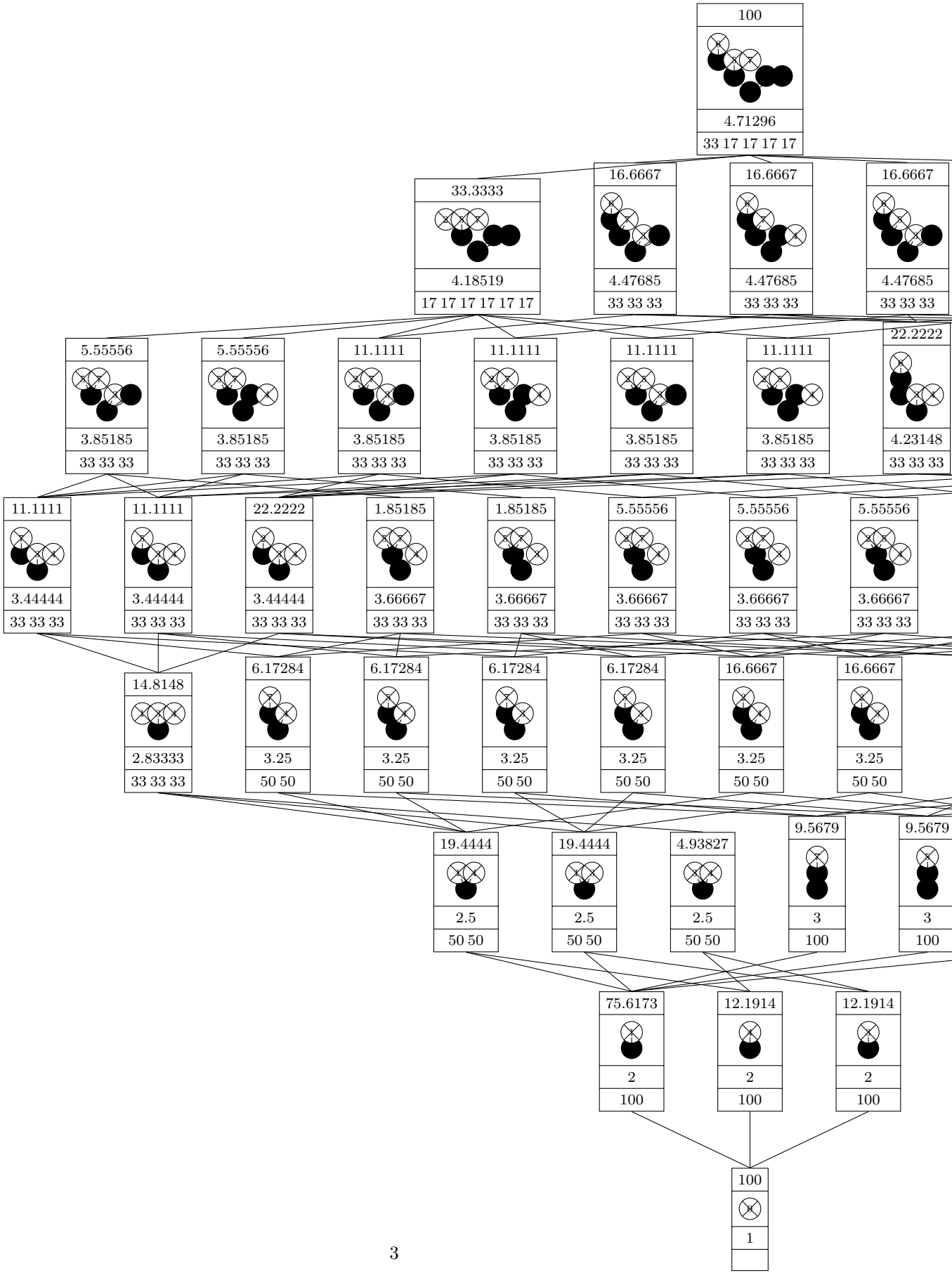
- Show that more processors can not be worse.DONE: Ok.
- Show that static and dynamic list-scheduling can not be optimal.DONE:

1.2 Current questions

- Gibt es situationen, wo im Optimum auf HLF-level tasks geschedult sind, auf (HLF-1)-level ungeschedulte tasks sind, und auf einem niedrigeren level wieder tasks geschedult sind?DONE: Ja! Siehe suboptimal HLF Beispiele
- Falls bei einer parallel chain oder einem degenerate intree bereits 2 tasks geschedult sind: Ist es dann optimal, einen HLF-task dazu zu nehmen?
- Benchmarks: Compare usual LEAF scheduler to SCLEAF and SCLEAF with conjecture that as many topmost tasks as possible shall be sheduled (esp. no. of snapshots is important).DONE: Erledigt. Verbesserung von 40%.
- P3: Is it possible, that there are – at one level of the optimal snapshot DAG – multiple snapshots containing the same intree, but a different set of scheduled tasks? REMARK: Probably not w.l.o.g.
- For each snapshot resulting from an optimal schedule, at least one top-most task is scheduled? REMARK: Seems so.
- If for an intree only non-top tasks are scheduled, you can exchange one of the non-top tasks with a top-tasks and obtain a better run time?REMARK: Seems so
- "Topmost singles:" Are topmost tasks, whose successors have *exactly one* predecessor, always scheduled?DONE: No! See chapter "suboptimal strategies" (number of topmost tasks).
- Are there intrees where we have to schedule topmost tasks of *three different (root-)subtrees? (Root-subtree: Subtree rooted at a direct predecessor of the root.)*DONE: Yes. Siehe suboptimal-Kapitel, irgendein non-HLF-optimal-Beispiel.
- If a tree T is non-HLF-optimal, can there a be a supertree S of this tree ($T \subseteq S$) such that S is HLF-optimal? DONE: Yes, refer to section **??**.
- Are chain collections optimally scheduled by HLF?DONE: Possibly so. Check proof once again.
- Let's assume we force the *first step* to be HLF. Is then the best possible solution with exactly this HLF-beginning again HLF?DONE: No – consider the non-HLF examples!

1.3 Other TODOs

- TikZ export very slow due to `generate_tikz_nodes`.
- Appendix containing all tree sequences for which optimal schedule is strictly non-HLF.


100
4.71296
33 17 17 17 17
33.3333
4.18519
17 17 17 17 17 17
16.6667
4.47685
33 33 33
16.6667
4.47685
33 33 33
16.6667
4.47685
33 33 33
5.55556
3.85185
33 33 33
5.55556
3.85185
33 33 33
11.1111
3.85185
33 33 33
11.1111
3.85185
33 33 33
11.1111
3.85185
33 33 33
11.1111
3.85185
33 33 33
22.2222
4.23148
33 33 33
11.1111
3.44444
33 33 33
11.1111
3.44444
33 33 33
22.2222
3.44444
33 33 33
1.85185
3.66667
33 33 33
1.85185
3.66667
33 33 33
5.55556
3.66667
33 33 33
5.55556
3.66667
33 33 33
5.55556
3.66667
33 33 33
14.8148
2.83333
33 33 33
6.17284
3.25
50 50
6.17284
3.25
50 50
6.17284
3.25
50 50
6.17284
3.25
50 50
16.6667
3.25
50 50
16.6667
3.25
50 50
19.4444
2.5
50 50
19.4444
2.5
50 50
4.93827
2.5
50 50
9.5679
3
100
9.5679
3
100
75.6173
2
100
12.1914
2
100
12.1914
2
100
100
1

Chapter 2

Theoretical foundations

2.1 Probability theory

Our computations are based on fundamental probability theoretic considerations. We therefore introduce some important concepts, where we rely on established notation (i.e. we use $\Pr[]$ to denote probabilities, $\mathbb{E}[]$ for expected values, etc).

2.1.1 Exponential distribution

The well-known exponential distribution is the central distribution we are dealing with in this text. All definitions and theorems within this subsection are along the lines of [**schickinger2001diskrete**].

Definition 2.1. *A continuous random variable is* exponentially distributed *with parameter λ if it has density*

$$f(x) = \begin{cases} \lambda \cdot e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

Note that the above definition also determines the distribution function F of an exponentially distributed random variable as follows:

$$F(x) = \begin{cases} 1 - e^{\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Theorem 2.2. *Let X be an exponentially distributed random variable. Then, the expectancy of X is $\mathbb{E}[X] = \frac{1}{\lambda}$.*

Proof. We can compute the expectancy for X as follows:

$$\begin{aligned} \mathbb{E}[X] &= \int_{-\infty}^{\infty} x \cdot f(x)\, dx = \\ &= \int_{0}^{\infty} x \cdot \lambda e^{-\lambda x}\, dx \\ &= \left[-\frac{e^{-\lambda x} \cdot (\lambda x + 1)}{\lambda} \right]_0^{\infty} \\ &= \frac{1}{\lambda} \end{aligned}$$

□

Theorem 2.3 (Scalability)**.** *Let X be an exponentially distributed random variable with parameter λ and let $a \in \mathbb{R}^+$. Then, the random variable aX is exponentially distributed with parameter $\frac{\lambda}{a}$.*

Proof. We compute the probability that the random variable aX is less than x:

$$\Pr[aX \leq x] = \Pr\left[X \leq \frac{x}{a}\right] = 1 - e^{-\frac{\lambda}{a} \cdot x}$$

This result is equivalent to the density function of an exponentially distributed random variable with parameter $\frac{\lambda}{a}$. □

Theorem 2.4. *Let $X_1, \ldots, X_n$ be exponentially-distributed random variables with respective parameters $\lambda_1, \ldots, \lambda_n$. Then, the ranvom variable $Z := \min_{i \in \{1,\ldots,n\}} \{X_i\}$ is exponentially distributed with parameter $\lambda = \lambda_1 + \cdots + \lambda_n$.*

Proof. We prove the claim by induction.

Suppose, we have two exponentially distributed random variables X_1 resp. X_2 with parameters λ_1 resp. λ_2. We then can compute

$$\begin{aligned}
\Pr[min\{X_1, X_2\} \geq x] &= \Pr[X_1 \geq x \wedge X_2 \geq x] = \\
&= \Pr[X_1 \geq x] \cdot \Pr[X_2 \geq x] = \\
&= e^{-\lambda_1 x} \cdot e^{-\lambda_2 x} = \\
&= e^{-\lambda_1 x - \lambda_2 x} = \\
&= e^{-(\lambda_1 + \lambda_2) \cdot x},
\end{aligned}$$

from which we can conclude that $\min\{X_1, X_2\}$ is exponentially distributed with parameter $\lambda_1 + \lambda_2$. By induction, we obtain our claim. □

Definition 2.5 (Memorylessness)**.** *A random variable X is called* memoryless *if*

$$\Pr[X > t + s \mid X > s] = \Pr[X > t]$$

Theorem 2.6. *Let X be an exponentially distributed random variable with parameter λ. Then, X is memoryless.*

Proof. We start by using the definition of conditional probability and rewrite until we arrive at our goal:

$$\begin{aligned}
\Pr[X > t + s \mid X > s] &= \frac{\Pr[X > t + s \wedge X > s]}{\Pr[X > s]} = \\
&= \frac{\Pr[X > t + s]}{\Pr[X > s]} = \\
&= \frac{e^{-\lambda \cdot (t+s)}}{e^{-\lambda s}} = \\
&= e^{-\lambda t} = \\
&= \Pr[X > t]
\end{aligned}$$

□

This is a very advantageous property that can be exploited in our considerations to follow.

Remark: It can even be shown that any memoryless continuous random variable is exponentially distributed, but theorem **??** is sufficient for our needs.

2.1.2 Minimum of continuous, identically distributed random variables

Theorem 2.7. *Let $X_1, \ldots, X_n$ be independent, identically distributed, continuous random variables and let $i \in \{1, \ldots, n\}$. Then*

$$\Pr\left[X_i = \min_{j \in \{1,\ldots,n\}} \{X_j\}\right] = \frac{1}{n}. \tag{2.1}$$

Proof. As no random variable is "preferred" over another (they all have the same distribution), it is clear that

$$\Pr\left[X_1 = \min_{j\in\{1,\dots,n\}}\{X_j\}\right] = \Pr\left[X_2 = \min_{j\in\{1,\dots,n\}}\{X_j\}\right] = \dots = \Pr\left[X_n = \min_{j\in\{1,\dots,n\}}\{X_j\}\right].$$

Because one of the random variables *must* be the minimum, we can deduce (**??**). □

2.1.3 Law of total probability

Theorem 2.8. *Let A be an event and let $\mathcal{B} = \{B_1, B_2, B_3, \dots\}$ a finite or countably infinite set of pairwise disjoint events with $A \subseteq \bigcup_{B\in\mathcal{B}} B$. Then, we have*

$$\Pr[A] = \sum_{B\in\mathcal{B}} \Pr[A \mid B] \cdot \Pr[B],$$

where $\Pr[A \mid B]$ denotes the conditional probability of A under the condition that B is fulfilled.

Proof. According to the requirements for $\mathcal{B}$, we have

$$A = (A \cap B_1) \cup (A \cap B_2) \cup (A \cap B_3) \cup \dots$$

Because two events B_i and B_j are disjoint (if $i \neq j$), we have that $(A \cap B_i)$ and $(A \cap B_j)$ are disjoint, as well. Thus, we can compute the probability

$$\Pr[A] = \Pr[(A \cap B_1) \cup (A \cap B_2) \cup \dots] = \Pr[A \cap B_1] + \Pr[A \cap B_2] + \dots$$

Now, solving the definition of conditional probability $\Pr[A \mid B_i] = \frac{\Pr[A\cap B_i]}{\Pr[B_i]}$ for $\Pr[A \cap B_i]$, and substituting each term accordingly, results in the desired equation. □

Theorem **??** can be used to derive its counterpart for expected values:

Theorem 2.9. *Let X be a random variable in a probability space Ω and $\mathcal{B}\{B_1, B_2, B_3, \dots\}$ a finite or countably infinite set of pairwise disjoint events with $A \subseteq \bigcup_{B\in\mathcal{B}} B$. Then,*

$$\mathbb{E}[X] = \sum_{B\in\mathcal{B}} \mathbb{E}[X|B] \cdot \Pr[B].$$

Proof. We use the definition of the expected value and theorem **??** to obtain the desired result. □

2.2 Intrees

As we will see later, we will constantly deal with *intrees*. In this section we develop simple notation for such trees. We assume that the reader is familiar with the concept of undirected trees and develop our notation on top of the one for undirected trees. For a more detailed introduction on what trees are, see e.g. [**diestel2005graph**].

Definition 2.10 (Intree)**.** *Let I' be a undirected tree. Let v be a vertex within I'. Let I be the directed version of I' in such a way that all edges are directed towards vertex v. Then we call I an* intree *or a* rooted tree *with* root *v.*

If there is a path from t_1 to t_2, we call t_1 a predecessor *of t_2. Moreover, we call t_2 a* successor *of t_1.*

If a task t in I has no predecessors, we call t a leaf*.*

Remark: If there is an edge (t_1, t_2) in I, we can emphasize that by speaking of t_1 as a *direct* predecessor of t_2 and of t_2 as a *direct* successor of t_1.

Throughout this thesis, if not stated otherwise, we use the terms intree and tree interchangeably, because we are mainly dealing with intrees.

Definition 2.11 (Level)**.** *Let I be an intree. Let v be a vertex within I. We define $level(v)$ be number of edges along the (unique) path from v to the root.*

The concept of levels is illustrated in figure **??**. Note that our definition of levels implies that the lowest level carries the number 0.

Definition 2.12 (Topmost leaf)**.** *A leaf t of an intree I is called a* topmost leaf *if for all vertices v of I, we have $level(t) \geq level(v)$.*

We introduce intuitive (but non-standard) notation for modifying an intree by adding or deleting leaves.

Definition 2.13 (Removal of leaves)**.** *Let I be an intree. Let v be a vertex of I. We call v a* leaf *if v has no predecessors.*

Let now x be a leaf of I. By $I \setminus \{x\}$ we denote the intree that is obtained if we remove x and the outgoing edge of x from I.

Definition 2.14 (Addition of leaves)**.** *Let I be an intree. Let v be an intree and t be a vertex in I. Let (v, t) be an edge. Then, we denote by $I \cup \{(v, t)\}$ the intree obtained by adding v to I as a direct predecessor of t.*

If the successor t of v is clear from the context, we sometimes simply write $I \cup \{v\}$ instead of $I \cup \{(v, t)\}$.

Definition 2.15 (Siblings)**.** *We call a vertex v a sibling of another vertex u if v and u have the same direct successor.*

Describing intrees We conclude this short introduction on intrees with an intuitive way of describing intrees. Most readers are probably acquainted with a visual description of intrees such the one in figure **??**. For textual representations (and even more important: for use in command line), we rely on another representation.

- We number the tasks (resp. vertices), beginning with 0 in ascending order.
- We always assign the intree's root the number 0.
- We iterate through all the tasks *beginning at 1* in ascending order, and write up only their *successor*.

Consider the intree in figure **??**: It has the edges

$$(1, 0), (2, 0), (3, 1), (4, 1), (5, 2), (6, 3), (7, 3), (8, 3), (9, 6), (10, 8).$$

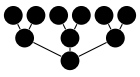
Carrying out the procedure given above yields the sequence

$$(0, 0, 1, 1, 2, 3, 3, 3, 6, 8).$$

Note that since the root carries number 0, we can start iterating at 1. Moreover, this representation enforces that the first number in such a sequence describing an intree is always 0. This, of course, introduces some redundancy, but, on the other hand, enables us to describe the tree consisting of only a root: It is denoted by an empty sequence: ().

That means, that the sequence $(x_1, x_2, \ldots, x_n)$ describes an intree with $n + 1$ tasks where task i is a requirement for task x_i.

Note that this representation still is not the most concise one in the sense that it allows several possible sequences for one intree. E.g. the sequences $(0, 0, 0, 1, 2, 2, 3, 3, 3)$, $(0, 0, 0, 1, 1, 1, 2, 2, 3)$, $(0, 1, 0, 0, 3, 4, 3, 4, 4)$ and $(2, 0, 0, 5, 0, 5, 3, 3, 5)$ all describe the following intree:



However, we will most of the time use a notion where we assume that for a tree sequence $(x_1, x_2, \ldots, x_n)$ we have $\forall i \in \{1, 2, \ldots, n\}\, x_i \leq i$. This is always possible because we can start a breadth-first search (BFS) at task 0 and assign numbers to the tasks representing the order in which BFS visits the tasks.

That means that there are $n!$ possibilities for tree sequences with exactly n tasks. On the other hand, there are less (distinct) intrees of size n:

Theorem 2.16. *Let T_n denote the number of intrees containing n tasks. Then,*

$$T_n \sim C \cdot r^{-n} \cdot n^{-1.5} \quad \text{as } n \to \infty,$$

where $C = 0.4399237\ldots$ and $r = 0.3383219\ldots$.

Proof. See [**asymptotic˙enum˙odlyzko**]. □

The first values for T_n are $0, 1, 1, 2, 4, 9, 20, 48, 115, 286, 719, 1842, 4766, 12486, 32973, \ldots$ [**oeisrootedtrees**].

2.2.1 Interpretation as dependency graphs

Those intrees can naturally be used to describe dependencies between different tasks, as long as each task is the requirement for *at most one* other task. Each vertex in an intree then represents a task. If t_1 is a (direct) predecessor of t_2, this means that the task associated with t_1 must be executed before the task associated with t_2. Since we represent tasks by vertices, we will often use the terms task and vertex interchangeably.

TODO: Vielleicht ein Beispiel aus der echten Welt.

Definition 2.17 (Ready tasks)**.** *Let I be an intree of tasks (represented by vertices) and. If t is a leaf of I, we call the corresponding task (resp. – for simplicity – the whole vertex)* ready*.*

The intree $(0, 0, 1, 1, 2, 3, 3, 3, 6, 8)$ (see figure **??**) has the ready tasks 4, 5, 7, 9 and 10.

2.2.2 Labelled and unlabelled intrees

We can draw such intrees in a very straightforward manner: We draw the root at the bottom and its direct predecessors one level above. For each predecessor, we inductively repeat this procedure to obtain a "top-to-bottom-description" of the tree.

Figure **??** shows an intree (the one given by the sequence $(0, 0, 1, 1, 2, 3, 3, 3, 6, 8)$), where tasks 8 is a requirement for task 6, which itself is – like task 7, 8 and (indirectly) task 10 – a requirement for task 3. This figure also illustrates the fact that – in an intree – each task is a direct requirement for *at most* one other task.

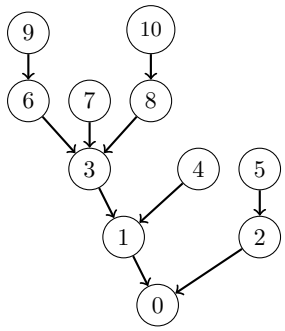
However, we are mostly interested in the *structure* of the tree, which is why we most of the time omit the labellings of the vertices (i.e. we omit the task names) and rely on a *unlabelled* representation as shown in figure **??**.

The mathematical concept behind unlabelled trees can be described by the notion of *isomorphisms*.

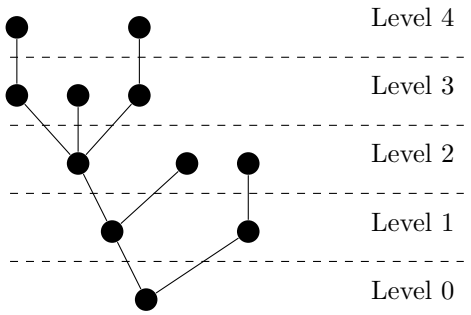
Definition 2.18 (Intree isomorphism)**.** *We call two intrees I_1 and I_2* isomorphic*, if there exists a bijective function that maps vertices of I_1 onto vertices of I_2 such that (v, t) is an edge in I_1 if and only if $(f(v), f(t))$ is an edge in I_2 and the root of I_1 is mapped onto the root of I_2.*

Remark: The above definition of isomorphisms is somwhat non-standard in the sense that usually isomorphisms are defined for undirected trees. For this work, we restrict ourselves onto our definition (except where explicitly stated otherwise).

As an example for isomorphic intrees, consider $(0, 0, 1, 1, 2, 3, 3, 3, 6, 8)$, $(0, 1, 2, 3, 0, 5, 1, 2, 2, 9)$ and $(0, 1, 0, 3, 3, 4, 4, 7, 4, 9)$.

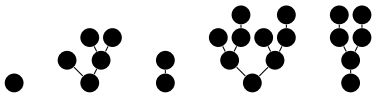


(a) Labelled version with vertex labels and edges drawn as arrows.

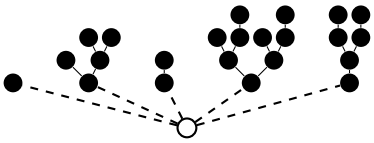


(b) Unlabelled version without arrows, edges are implicitly directed towards the root.

Figure 2.1: Graphical representation of an intree $((0, 0, 1, 1, 2, 3, 3, 3, 6, 8))$ with 5 levels (numbered 0 to 4). All edges are implicitly directed towards the root, which is drawn at the bottom of the tree. Most of the time, the *structure* of the tree is enough, so we will omit vertex names most of the time.



(a) Original in-forest (each component is a separate intree).



(b) Adding a new root that is connected with all intrees' roots.

Figure 2.2: Converting an in-forest to an intree.

2.2.3 Extension to in-forests

It is worth mentioning that – for our needs – there is a straightforward extension of intrees to in-forests for our scenario. In-forests are graphs whose components are all intrees. To convert an in-forest to an intree, we simply add an auxiliary new root and connect all roots of the intrees within the in-forest to the new root. This way, we clearly obtain a new intree.

To manipulate intrees, we define some non-standard notation that is, however, very convenient and should be easily understandable.

Chapter 3

Schedules

We will now combine the knowledge from chapter **??** and introduce the problem we will be working on.

3.1 Computational setting TODO: Better title.

We now describe our assumptions concerning the tasks to be processed and our machines.

Intree constraint We assume that the dependencies between the tasks under consideration can be described as an intree (as introduced in section **??**).

Random task processing times We assume that we do not know the processing time for a certain task in advance. We instead assume that we can model the task time for each task within the intree by an exponentially distributed random variable as defined in section **??**. We furthermore assume that all tasks are exponentially distributed with the *same* parameter λ. Note that we can assume w.l.o.g. – according to theorem **??** explaining exponential variables' scaling – that all tasks have parameter $\lambda = 1$.

Parallel processors We assume that we have a certain number (we will focus on 2 or 3) of *identical* processors. These processors work in parallel and can be used to carry out the individual tasks. For simplicity we assume that switching from one task to another takes no time at all.

Tasks are "atomic" We assume that one single task can be processed by one processor exclusively, i.e. it is not possible to save time by "splitting" one single task over several processors. That, moreover, means that there might be idle processors if we have more ready tasks than processors.

Non-preemtive scheduling We will focus on non-preemtive scheduling, i.e. once a task is processor is assigned a task, this task has to be processed *completely* before the processor can be used for anything else. Note that the restriction on non-preemtive scheduling is no problem when we work with two processors (as shown in [**chandyreynoldslargepaper1979**]) — in this case, the optimal schedule is (without loss of generality) non-preemtive. However, for three processors, preemtive schedules might be better than non-preemtive ones. We will later show an example intree for this fact (see section **??**).

However, as shown in [**chandyreynoldslargepaper1979**] an optimal schedule (without loss of generality) only preemts a task directly after another task finishes.

3.1.1 Processing a whole intree of tasks

Assuming that we have a certain amount p of identical processors, we can process all tasks in an intree (of course in a valid order). Naturally, We have to do this step by step.

That is, we assign some (at most p) ready tasks to processors, wait until the first assigned task finishes, assign a new task (if available) to the now idle processor, and continue with the subtree obtained

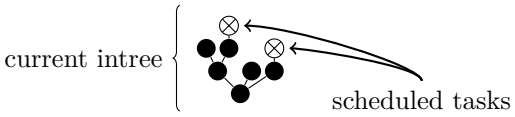


Figure 3.1: A snapshot is determined by its intree and a collection of currently scheduled tasks.

by removing the finished task. That is, in each point of time, we know the current intree containing all tasks not yet finished and all tasks currently being processed. We therefore introduce the notation of *snapshots*.

Definition 3.1 (Schedules and snapshots)**.** *Let I be an intree of tasks that is processed. A* schedule *is the order in which tasks are processed. Since task times are unknown, this order is not deterministic and can vary from execution to execution.*

Let S be a schedule of I. A snapshot *of S a pair containing*

- *the current subtree $I' \subseteq I$ describing which tasks are not finished yet,*
- *a set X holding the tasks that are currently being processed (i.e. a set of* currently scheduled tasks*).*

While the above definition is well suited for theoretical considerations, we – most of the time – are satisfied with a concise graphical representation, that will look as shown in figure **??**, i.e. it is depicted as an intree, whose scheduled tasks are marked by a crossTODO: Schauen, ob ich das geändert habe!.

Consider now a snapshot, i.e. the current intree and a set of currently scheduled tasks. At this point, there are several possibilities:

- Any of the currently scheduled tasks may be the first task to finish.
- If a certain task finishes, the corresponding processor gets idle and can be assigned new work. That is, we may be able to choose a new task[1] and assign it to the now idle processor.

  Note that we can – in principle – easily choose a certain task with some probability and another task with another probability, thus introducing more possibilities.

That is, we have several possibilities here. The first of them (which task finishes first) can not be influenced in our scenario. The second, however, leaves us a choice that we can influence. Depending on which task we choose next, we might get a better or worse overall run time.

Definition 3.2 (Scheduling strategy)**.** *A* scheduling strategy *determines the probability that a certai tasks should be the next tasks to be scheduled.*

That means, we use the notion of a *probabilistic scheduler* in the sense that the scheduler is not forced to restrict on *one single* task, but can instead specify probabilities for each ready, currently unscheduled task. Note that we can easily "simulate" a deterministic scheduler by assigning probability 1 to one single ready task, and probability 0 to all other possibilities.

One desirable goal is to always choose the next task to be scheduled (resp. the respective probabilities) in such a way that the overall expected run time is minimal. We will now see how we can compute the expected run time for a schedule.

3.1.2 Two prominent scheduling strategies

We will now present two important scheduling strategies that we will use to research the problem throughout this work.

Definition 3.3 (HLF scheduling)**.** *A* highest level first *scheduler (or HLF scheduler) assigns, if there are idle processors, ready tasks whose levels are maximal.*

[1] It may be the case that we are not able to select any new task because all leaves are already scheduled. In this case, the processor has to stay idle and can not get assigned a new task.

Note that the above definition of HLF leaves some freedom if there are several tasks that could be chosen by HLF. If this is the case, then there are (basically) two possibilities:

- Choose *one* task according to a certain pattern or randomly.
- Assign to *each* possibly chosen task a certain probability.

Note that we can consider the first possibility as a special case of the second one. HLF scheduling is optimal for intrees whose task times are equal scheduled on any number of processors [**hu:1961:hlfoptimalforknowntimesi**
and for intrees whose task times are exponentially distributed (with same parameter) scheduled on two processors (see [**chandyreynoldsshortpaper1975**] and chapter **??**).

Another (very trivial) scheduler is the scheduler that tries all possibilities:

Definition 3.4 (LEAF scheduling)**.** *A LEAF scheduler, if a processor gets free and can be assigned a new task, assigns each ready task with the same probability to this processor. In the beginning, each possible combination of scheduled tasks is taken with the same probability.*

LEAF schedulers are useful to examine all possible schedules and play an important role when we research how many snapshots have to be examined *at most*. Note that a LEAF scheduler examines *all possible* schedules. The resulting schedule, thus, *must* contain the optimal choices as a part of it. That means, we can compute an optimal schedule by first scheduling the intree with the LEAF scheduler, and afterwards excluding the "wrong" (i.e. suboptimal) choices.

3.2 Visualizing a schedule

If we are speaking of a schedule, we talk about the complete processing of an intree of tasks.

We can think of it as a connected directed acyclic graph or DAG (see [**diestel2005graph**] for more information), where each vertex represents one single snapshot. The edges between the snapshots then represent transitions that indicate that a certain task has finished and (possibly) a new task has chosen to be scheduled. We can naturally assign the edges weights that represent the corresponding probability that a task has finished and the new task has been picked.

As stated before, we have two sources of non-determinism:

- Each of the currently scheduled tasks can be the first task to finish. Depending on which task finishes first, we have to continue with another intree.
- When one task finishes, the corresponding processor becomes idle and can be used to process other tasks. This is the second source where we have different possibilities. However, for this point, we can decide ourselves how to continue.

We will look at the intree $(0, 0, 1, 2, 2, 2, 6, 6, 8)$ – it shall be processed by two processors – as an example. This intree looks as follows:

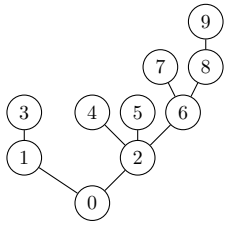
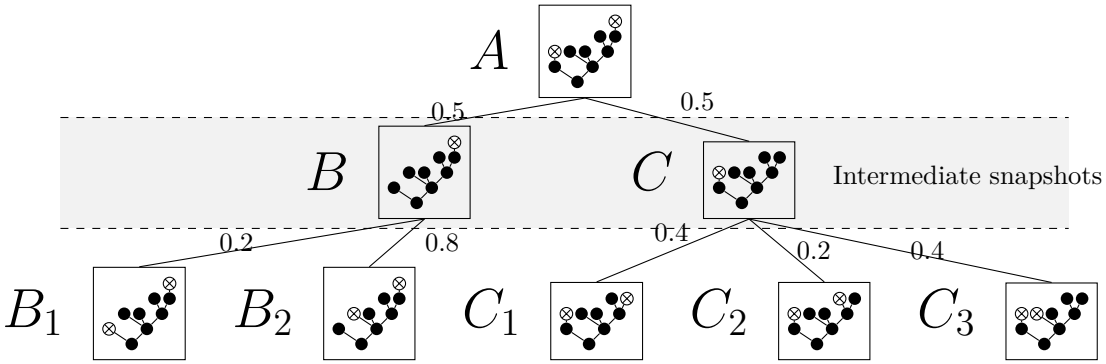


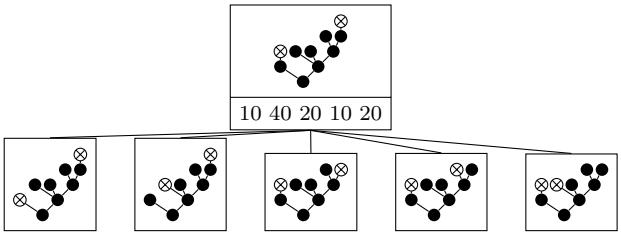
Figure **??** shows (the beginning) a two-processor schedule for the intree $(0, 0, 1, 2, 2, 2, 6.6, 8)$ in two variants: Figure **??** shows the schedule and incorporates situations where the scheduler can decide wich tasks should be scheduled next with wich probabilities ("intermediate snapshots"). At the beginning, the nodes 9 and 3 are scheduled (snapshot A). Then each of these two tasks is the first one to finish (each one with probability 50%). This intermediate situation, where exactly one of the two tasks is finished

and the scheduler can assign probabilities describing which task to pick next, are shown in snapshots B (this is the situation where task 3 finished first) and C (where 9 finished first).

In the situation described by the intermediate snapshot B, the scheduler can decide which task shall be chosen next with which probability. In the situation of snapshot B, the scheduler decides that with probability 20%, the task 1 shall be chosen as next task (snapshot B_1) and with probability 80% task 4 shall be chosen as next task (snapshot B_2). Similarily it decides in the situation denoted by C that tasks 8 resp. 4 shall be chosen with probability 40% each (C_1 resp. C_3) and task 7 shall be chosen with probability 20% (C_2).



(a) A snapshot DAG with intermediate snapshots. In the beginning, there are two tasks scheduled. Each task is the first to finish with probability 0.5 (see the corresponding edges). An intermediate snapshot describes a situation where the scheduler can assign probabilities to be scheduled as next task to the tasks. According to these probabilities, the processing continues with the corresponding probabilities.



(b) The same snapshot DAG without intermediate snapshots. Intermediate snapshots are eliminated and the edges are drawn directly between non-intermediate snapshots. The probabilities are in shown in percent in directly within the snapshot (in the respective order). The probabilities are obtained by multiplying the corresponding probabilities along the original edges that are needed to form the new edges.

Figure 3.2: An exemplary schedule DAG with and without intermediate snapshots. From now on, we will focus on snapshot DAGs without intermediate snapshots.

Figure **??** shows the version of the schedule that omits intermediate snapshots. From now on, we will restrict ourselves to the latter structure (without intermediate snapshots) because this representation is easier to maintain and – after some time to get accustomed to it – as easy to understand as the version with intermediate snapshots.

3.3 Computing the expected run time of a schedule

Computing the expected run time for a given schedule (more precisely for the whole processing of an intree according to a certain schedule) can be easily acchieved via a recursive formula that can be explained as follows:

- If the current intree I consists of exactly one task (the root), then we simply have to process this single task. Since one task has expected run time $\frac{1}{\lambda} = 1$ (remember: we assumed w.l.o.g $\lambda = 1$),

we know that the expected run time for I is 1.

- If the current intree consists of more than 1 task, there may be up to p tasks scheduled, where p denotes the number of processors. Let us assume that there are $r \leq p$ tasks scheduled (r may be smaller than p if there are less ready tasks than processors) and denote the set of these tasks by $X = \{x_1, x_2, \ldots, x_r\}$. According to theorem **??**, the probability that task x_i ($i \in \{1, 2, \ldots, r\}$) is the *first* task to finish is $\frac{1}{r}$. The expected run time for x_i is $\frac{1}{n}$ (by theorems **??** and **??**). Moreover, the probability that two tasks finish at exactly the same time is 0 (since the run times of tasks are exponentially – i.e. continuously – distributed).

  If task x_i finishes, the remaining intree is $I \setminus \{x_i\}$, with – due to non-preemtive scheduling – at least tasks within $\{x_1, x_2, \ldots, x_r\} \setminus \{x_i\}$ scheduled. By X_i we denote the set of tasks that are scheduled in the next step (i.e. $\{x_1, x_2, \ldots, x_r\} \setminus \{x_i\} \subseteq X_i$). The expected run time for $I \setminus \{x_i\}$ can then be computed recursively (see remark below).

  This means: If x_i is the first task to finish, we can use the expected run time is given by

  $$\frac{1}{r} + T_{X_i}(I \setminus \{x_i\}),$$

  where $\frac{1}{r}$ accounts for the expected run time of task x_i and $T_{X_i}(I \setminus \{x_i\})$ denotes the run time for the intree $I \setminus \{x_i\}$ if the tasks within X_i are scheduled (of course with respect to a specific scheduling strategy).

  We define events $F_1, \ldots, F_r$, where F_i describes the event that task x_i is the first task to finish. Note that the events F_i are pairwise disjoint, $\Pr[F_i] = \frac{1}{r}$ and that one of the events F_i *must* occur (meaning that $\Pr[F_1 \cup \cdots \cup F_r] = 1$).

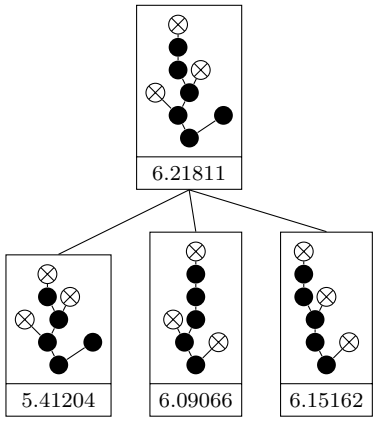
  Since *each* of the tasks $x_1, x_2, \ldots, x_r$ can be the first task to finish (each with probability $\frac{1}{r}$ according to theorem **??**), we can use theorem **??** and compute the overal run time by

$$\begin{aligned}
\mathbb{E}[T_X(I)] &= \sum_{i=1}^{r} \Pr[F_i] \cdot \mathbb{E}[T_X(I) \mid F_i] = && (3.1)\\
&= \sum_{i=1}^{r} \Pr[F_i] \cdot \mathbb{E}[T(x_i) + T_{X_i}(I \setminus \{x_i\})] = && (3.2)\\
&= \sum_{i=1}^{r} \frac{1}{r} \cdot \left(\frac{1}{r} + \mathbb{E}[T_{X_i}(I \setminus \{x_i\})] \right) = && (3.3)\\
&= \frac{1}{r} + \frac{1}{r} \cdot \sum_{i=1}^{r} \mathbb{E}[T_{X_i}(I \setminus \{x_i\})]. && (3.4)
\end{aligned}$$

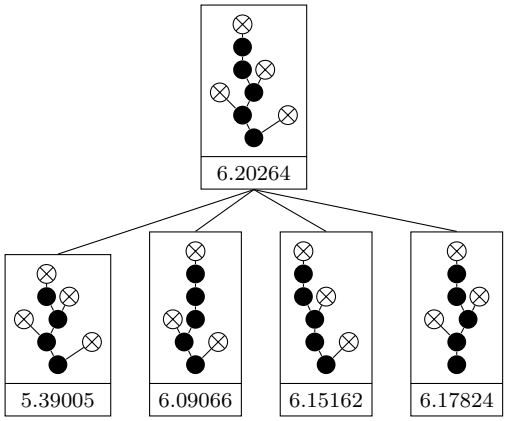
Remark: We can reformulate (**??**) to (**??**) because the expected run time to process the whole tree I under the condition F_i (i.e. under the knowledge that task x_i is the first to finish) can be computed by summing up the time that is needed by task x_i (thus the term $T(x_i)$) and the time needed for the remaining tree (denoted by $T_{X_i}(I) \setminus \{x_i\}$).

Since all tasks' processing times are assumed to be exponentially distributed, i.e. in particular memoryless (see theorem **??**), we can argue as follows: If one task finishes, the other scheduled tasks – of course – *might* have been processed up to a certain point. But since their processing times are memoryless, we can assume them to behave *as if* they had not been processed at all. That is, the remaining tree can be considered *as if* we just started all scheduled tasks. This, however, means that we are in the same situation as before, except that we are dealing with an intree that has one task less than the intree before. Thus, we can justify the reformulation from (**??**) to (**??**).

As an example, consider the tree in figure **??**. The topmost snapshots describe the first step of schedules for three resp. four processors on the same intree. We are the corresponding expected run times. We can assume that the run times of the snapshots in the second line have already been computed (recursively). Using these, we can compute the overal expected run time using theorem (**??**) as described in figures **??** and **??**.



(a) Three processors. Transition probabilities are $\frac{1}{3}$ each. Then, we can compute $\frac{1}{3} + \frac{1}{3} \cdot (5.41204 + 6.09066 + 6.15162) = 6.21811$.



(b) Four processors. Transition probabilities between snapshots are $\frac{1}{4}$ each, so we can compute the overal expected run time by $\frac{1}{4} + \frac{1}{4} \cdot (5.39005 + 6.09066 + 6.15162 + 6.17824) = 6.20264$

Figure 3.3: Computing the expected run time according to equation (**??**) for three resp. four processors on the same intree. Each box represents a single snapshot containing the intree (with scheduled tasks marked) and the expected run time for the respective snapshot.

Computing the runtime for an in-forset As mentioned in section **??**, we can also deal with in-forests by adding a new root connected to all intrees of the forest. To compute the expected run time for a specific schedule, we simply compute the expected run time for the intree resulting from the in-forest conversion and subtract 1. This can be done because the last task in the resulting intree to be processed *must* be the root we introduced. The expected run time for the root is 1, so we can simply subtract 1 to obtain the expected run time for the original in-forest.

3.4 Optimizing a given schedule

Now we know how to compute a schedule according to a specific scheduling strategy. We now describe an algorithm that can be used to improve the original schedule. It recursively discards "wrong choices", i.e. decisions that are suboptimal. The technique for it is shown in algorithm **??**.

Line **??** of algorithm **??** needs some explanation: We did not in particular specify how to "adjust" the probabilities properly. This is because there are different solutions. The only thing that must be kept in mind is that the sum of all probabilities for reaching a snapshot from the set $\textsc{BestSnaps}(osucs_t)$ must be equal to the probability that task t is the first task to finish (i.e. in our particular case, it must be equal to $\frac{1}{r}$ if there are r tasks scheduled for the respective snapshot).

One canonical way of assigning probabilities to snapshots within $\textsc{BestSnaps}(osucs_t)$ is to assign probability $\frac{1}{r \cdot |\textsc{BestSnaps}(osucs_t)|}$ to each of the snapshots within this set. Another way is to choose one particular element in $\textsc{BestSnaps}(osucs_t)$ and assign it probability $\frac{1}{r}$ while all other snapshots in $\textsc{BestSnaps}(osucs_t)$ get probability 0. This way, we eliminate further snapshots by making them impossible to reach.

An important aspect of this optimization routine is that we can – in an implementation – assure that it is executed *at most once* for each snapshot (by simply having a flag that indicates whether a snapshot was already subject to optimization). This means that we can assure that the total cost to optimize all snapshots of a schedule is bounded by $O\left(\sum_{s \in D} optcost(s)\right)$ where D denotes the snapshot DAG for the particular scheduler and $optcost(s)$ gives the cost to optimize a snapshot s. If we know the most expensive snapshot to optimize and call it s^* we can argue that $O\left(\sum_{s \in D} optcost(s)\right) \subseteq O\left(|D| \cdot optcost(s^*)\right)$.

Algorithm 3.1 Optimizing a given shapshot recursively

```
procedure OptimizeSnapshot(s)                    ▷ s is the snapshot to be optimized
    for t ∈ scheduled(s) do                      ▷ Iterate over all currently scheduled tasks
        sucs_t ← {s' | s' is a successor of s resulting if t is the first finishing task}
        osucs_t ← {OptimizeSnapshot(s') | s' ∈ sucs_t}
    end for
    newsucs ← ⋃_{t∈scheduled(s)} BestSnaps(osucs_t)
    return new snapshot with successors newsucs and adjusted transition probabilities
end procedure

procedure BestSnaps(snaps)
    return {s | s ∈ snaps, expectedTime(s) ≤ min{expectedTime(s') | s' ∈ snaps}}
end procedure
```

3.5 Computing the optimal schedule

We saw that we can optimize a given schedule and exploit this fact in conjunction with the LEAF scheduler (that corresponds to an exhaustive search). We first compute *all possible* schedules using the LEAF scheduler and – afterwards – optimize this using algorithm **??**. This way, we can compute the optimal schedule by an exhaustive search.

It is clear that the resulting schedule must be optimal because algorithm **??** chooses – for each scheduled task t that could be the first to finish – only the best solutions. Since a "complete schedule" from the LEAF schedule contains all possible schedules as a part of it, the optimization algorithm picks the best choices possible for each step.

Again, we can give the asymptotic run time of $O(|D| \cdot optcost(s^*))$ (see section **??**), and we can tell that the snapshot DAG D for an intree I scheduled on p processors contains *at most* $|\{T \mid T \subseteq I\}| \cdot n^p$ snapshots. This can be easily explained because each snapshot is associated with a subtree of I and for this subtree we have at most n^p possible choices for the tasks to be scheduled. Note that in practice there are much less snapshots than this given bound because e.g. it is in most cases not possible to have n^p combinations for the scheduled tasks.

3.6 Equivalent snapshots

We now move on to describing a basic insight about schedules. Foremost, we make the following – quite simple – observation: If two intrees I_1 and I_2 are isomorphic, then for each schedule S_1 for I_1, there is a schedule S_2 for I_2 that has exactly the same run time. We see this by examining the bijective function $f : I_1 \mapsto I_2$ and for S_2 scheduling exactly the task $f(t_1)$ if S_1 would choose t_1.

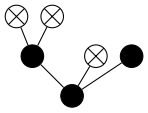
The second observation is that we can make the snapshot DAG more compact by avoiding snapshots that are – essentially – duplicates. We therefore extend the concept of isomorphisms from trees to snapshots in a straightforward way.

Definition 3.5 (Equivalent snapshots)**.** *Let $S = (I, X)$ be a snapshot with intree I and a set X of currently scheduled tasks. The snapshot $S' = (I', X')$ is called* equivalent *to S if there is an intree isomorphism from I onto I' such that the sets of scheduled tasks also correspond under this isomorhpsim (i.e. $X' = \{f(x) \mid x \in X\}$).*

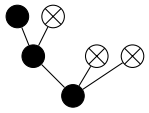
If S and S' are not equivalent, we call them distinct*.*

The requirement that scheduled tasks are "preserved" by the isomorphism is central to equivalence of snapshots, because otherwise, the corresponding run times may be unequal. Figure **??** shows two variants of the intree $(0, 0, 0, 1, 1)$ with different sets of scheduled tasks, resulting in two distinct snapshots.

It can be easily seen that we essentially compute the same thing twice if we carry out computations for several equivalent snapshots. Therefore, we can combine equivalent snapshots into one snapshot.



(a) Intree $(0, 0, 0, 1, 1)$ with scheduled tasks 2, 4 and 5.



(b) Intree $(0, 0, 0, 1, 1)$ with scheduled tasks 2, 3 and 5.

Figure 3.4: Two non-equivalent (i.e. *distinct*) snapshots. This example shows that even if the underlying intrees are isomorphic, the containing snapshots are not necessarily equivalent, because there is no isomorphism between the two intrees that preserves scheduled tasks.

This transformation (treating equivalent snapshots as one single snapshot) requires us to adjust the probabilities in the snapshot DAG, which can be done straightforward by summing up all probabilities along the edges that are merged into one single edge. Figure **??** shows an example where we would originally have to deal with six snapshots, but can reduce this to only two snapshots by excluding equivalent snapshots.

From now on, if not stated otherwise, we restrict ourselves to snapshot DAGs that contain only distinct snapshots.

3.7 Snapshot DAGs revisited

We have seen transition probabilities between certain snapshots, how compute the expected run time of a schedule recursively, excluding equivalent snapshots. We now come back to the description of a schedule as a snapshot DAG (as introduced in section **??**) that already contained the transition probabilities as elements in the corresponding snapshots.

We augment this visualization by two new elements: The expected run time for a snapshot and probability of reaching a certain snapshot.

Figure **??** visualizes a schedule for the intree $(0, 0, 1, 1, 2)$ processed by three processors. This snapshot DAG contains – for each snapshot – the probability of reaching it, the corresponding intree, the expected run time and the probabilities for the successors.

3.8 Summary: Problem statement

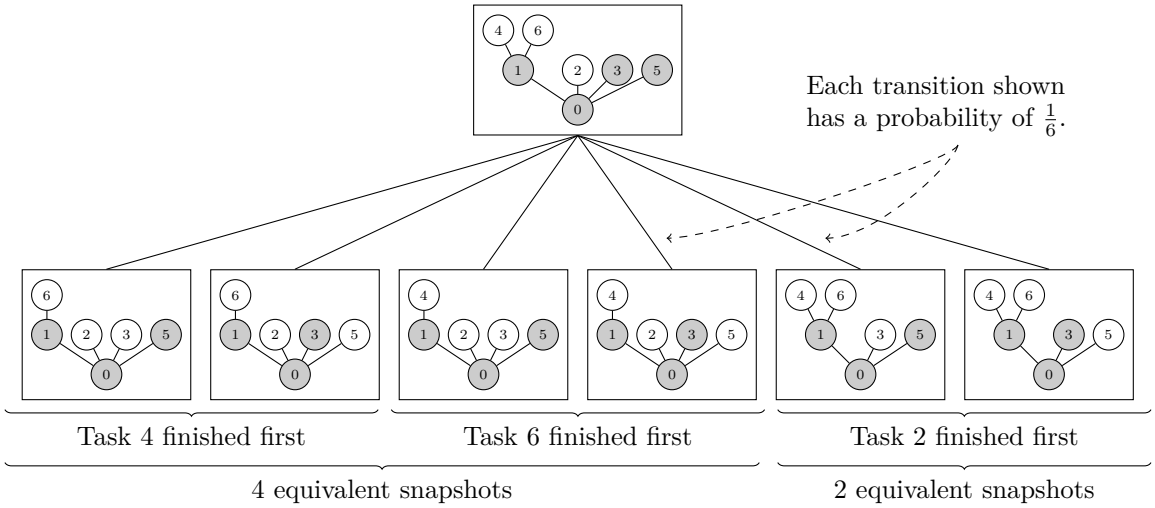
We now sum up what we have as input and what we want to do:

- Given an intree of tasks,...
- ...whose run times are independently, identically exponentially distributed, ...
- ...and that can not be preemted ...
- ...we want to obtain an optimal scheduling strategy, ...
- in the sense that we want to minimize the overal expected run time.

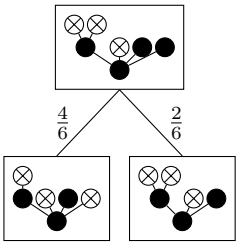
There is a convenient and de-facto standard notation for scheduling problems (see [**schedulingclassification**]) that basically consists of three fields α, β and γ separated by a vertical bar. We do not go into details about this notation, but we will briefly characterize our problem under consideration in this notation.

The first part, α, describes the *machine environment*. We assume that we have a set of identical machines which is indicated by $\alpha = P$. Because we are mainly dealing with 2 or 3 identical machines, we can be more concrete and set $\alpha = P2$ resp. $\alpha = P3$.

After we have specified the machine environment, we now describe the *job characteristics* with the symbol β. The field β that can be seen to consist of several parts $\beta_1, \beta_2, \ldots$, of which we concentrate on the most important ones:



(a) Schedule DAG showing all possible snapshots. Tasks currently scheduled are drawn white. Other tasks are drawn gray.



(b) Condensed DAG with distinct snapshots only.

Figure 3.5: Equivalent snapshots can be eliminated. Note that this changes the probabilities along the edges. To obtain the new probabilities, we sum up the probabilities all edges that are merged into the corresponding edge.

- We are mainly interested in a scenario where jobs can not be preemted, i.e. once a job is assigned it has to be processed until the end. Using the notation from [**schedulingclassification**] this is indicated by leaving β_1 empty.
- As said, we are dealing with intrees, this is – in [**schedulingclassification**] – denoted by $\beta_3 = tree$. To emphasize that we are dealing with *intrees* (in contrast to *outtrees*), we set $\beta_3 = intree$.
- We are dealing with tasks whose processing times are not known a priori but are assumed to behave like independent exponentially distributed variables with same parameters. This notation is not intended by [**schedulingclassification**] which is why we introduce a new notation by $\beta_7 = exp$.

The fields of β that we did not mention here are left empty.

In total, we have $\beta = intree, exp$ indicating that we have an intree structure of tasks whose processing times are assumed to be random variables as described.

Last, we specify the *optimality criteria* resp. the *objective* we want to acchieve. As said we are interested in keeping the expected overal run time low. The notation from [**schedulingclassification**] is not designed for stochastic scheduling problems, which is why we extend the notaion to incorporate the *expected makespan*. Following [**pinedo2008scheduling**] we could simply write $\gamma = C_{max}$ but to emphasize that we are talking about the *expected* makespan, we will set $\gamma = \mathbb{E}[C_{max}]$.

To sum up, we consider the scheduling problems described by

$$P2 \mid intree, exp \mid \mathbb{E}[C_{max}] \quad \text{and} \quad P3 \mid intree, exp \mid \mathbb{E}[C_{max}].$$

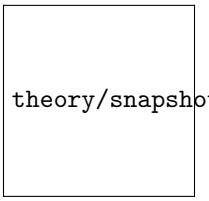


Figure 3.6: Snapshot DAG with each node containing the following elements of the respective snapshot: Probability of reaching it, intree (with scheduled tasks marked), expected run time, probability of successors.

Chapter 4

Warmup: Two Processors

While the main part of this work focuses on scheduling with tree processors, we start off by considering the – already solved – two-processor case to become familiar with the problem.

4.1 Optimal solution for two processors

For the two processor case, the *highest-level-first* policy is known to be optimal according to [**chandyreynoldsshortpaper**
This strategy at the beginning schedules two tasks whose distances to the root are as large as possible. If the HLF strategy can choose between several tasks, it can – for two processors – pick any of them. The expected run time will be the same, as shown in [**chandyreynoldsshortpaper1975**]. Figure **??** shows two HLF schedules for the intree $(0, 0, 1, 1, 2, 2, 2)$ and we can observe that snapshots on the same level have the same run time.

4.2 Profiles

We have seen in the previous section that certain snapshots have the same run time if processed by two processors using the HLF strategy. In fact, it has been proven in [**chandyreynoldsshortpaper1975**] that two intrees that have the same number of nodes on each level, have the same expected optimal run time (for two processors).

Moreover, we can conclude that we can compute the optimal expected finish time in polynomial time. This chapter provides a proof for this claim that mainly relies on so-called *profiles*. In this context, a profile is another structure that describes (some characteristics of) an intree in a compressed form.

4.2.1 Profiles of Intrees

If we consider the trees in figure **??**, we can compute that for two processors HLF always yields an expected run time of $\frac{49}{8}$ for each of them, which is optimal:

The intrees in figure **??** have the following in common: At each level, they have the same amount of tasks (six tasks at the topmost level, three in the middle one and one at the bottom level).

We can use the number of tasks per level as a (non-bijective) "encoding" of intrees. For now, we call this encoding a *profile* of the intree. The above intrees would all be encoded as a profile containing the numbers 6, 3 and 1 in that order. We denote the profile by $[\![6, 3, 1]\!]$.

Note that not all sequences of numbers can be used as profiles. In particular, the last number in a profile is (w.l.o.g.) 1 (since we have only one task as the root of the tree)[1]. Moreover, it can not be the case that there is a zero in a profile (since this would imply that there is *no task* on one specific level in the intree).

Moreover, we introduce a abbreviating notation for profiles.

[1] This, of course, introduces some overhead in notation, but we leave it as it is since it is easier to read this way.

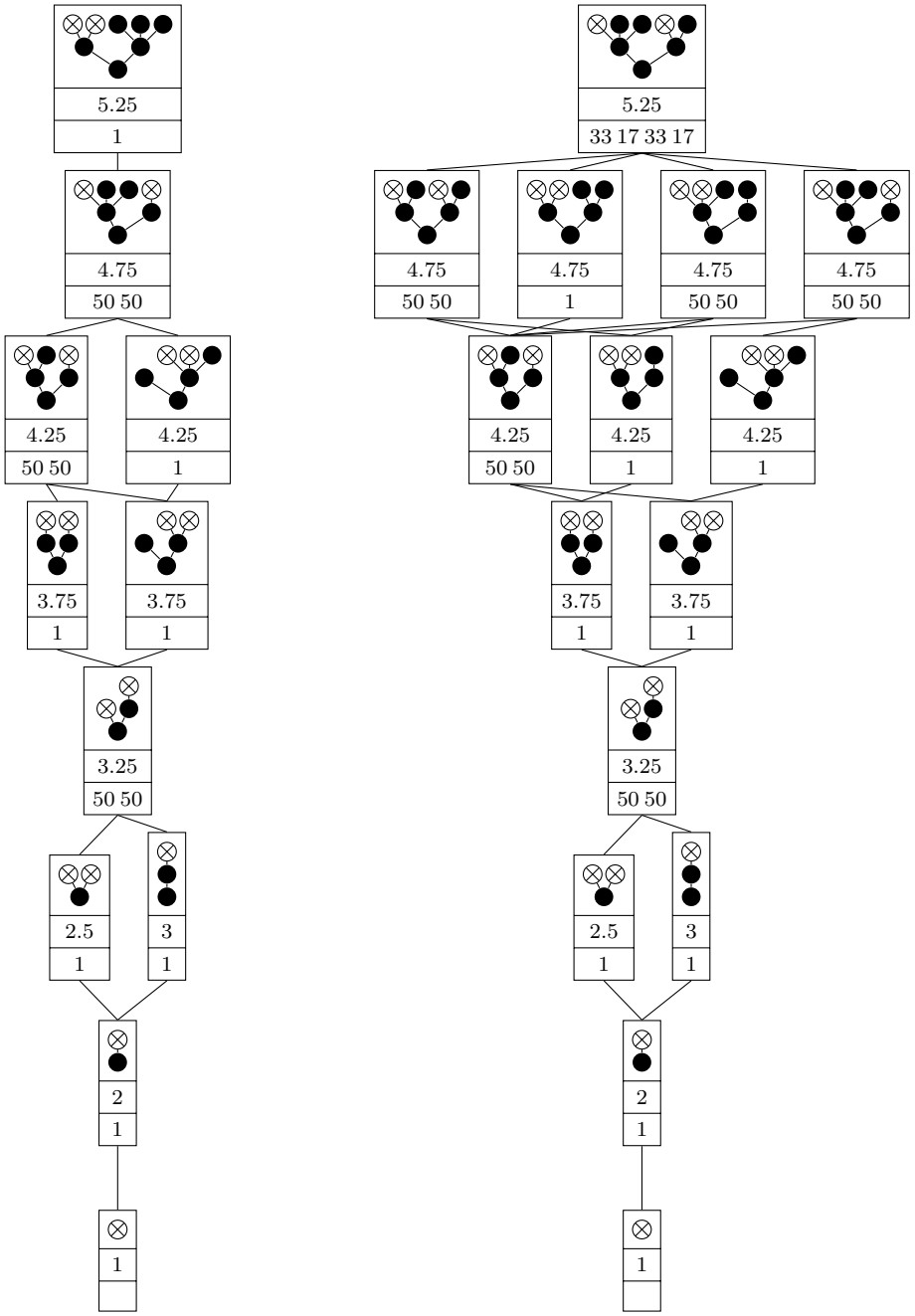


Figure 4.1: Two possible two-processor HLF schedules for intree $(0, 0, 1, 1, 2, 2, 2)$. Both schedules have the same run time. Note that snapshots on the same levels have same run time.

Definition 4.1 (Compact notation of profiles)**.** *For a profile p, we introduce a shorthand notation that groups successive ones. That is, instead of writing j consecutive ones, we simply write $(1)^j$.*

As a simple example, we rewrite $[\![2, 1, 1, 1, 5, 2, 1, 1, 1, 1, 1, 2, 1]\!]$ as $\left[\!\left[2, (1)^3, 5, 2, (1)^5, 2, (1)^1\right]\!\right]$.

Figure 4.2: Four intrees with the same profile ($[\![6,3,1]\!]$). All of them have expected run time of 49/8 if scheduled with HLF on two processors.

4.2.2 Profiles and HLF

For two processors and HLF-scheduling, we can easily conclude the successors of a profile. Let us first of all consider some examples here: If we have the profile $[\![5,4,2,1]\!]$, then two of the five topmost tasks *have to be scheduled* (since we are using HLF). If one of these two topmost tasks is finished, we reach $[\![4,4,2,1]\!]$ (see figure **??** for reference).

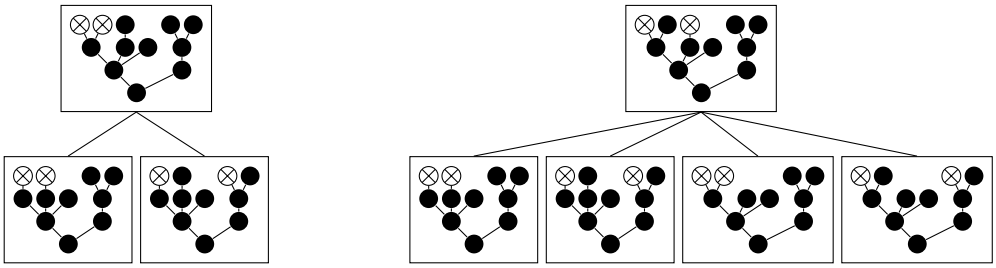


Figure 4.3: Intree with profile $[\![5,4,2,1]\!]$. *All* possible HLF-successors of the original intree have profile $[\![4,4,2,1]\!]$.

Another interesting case is $[\![1,5,2,1]\!]$, where the (single) topmost task and one of the five tasks on the second level are scheduled. If the topmost task is finished (which happens with probability $\frac{1}{2}$), we reach $[\![5,2,1]\!]$. If the scheduled task on the second level finishes first, we reach $[\![1,4,2,1]\!]$.

The last example we want to give here is $[\![1,1,1,3,1]\!]$. In this case, the single topmost task and one of the three tasks of the second lowest level have to be scheduled. If the topmost task finishes first (which happens with probability $\frac{1}{2}$), the resulting profile will be $[\![1,1,3,1]\!]$ (where again the topmost task and one of the tree tasks in the second lowest level are scheduled). If the other scheduled task finishes first, we reach $[\![1,1,1,2,1]\!]$, where the single topmost task and one of the remaining *two* tasks on the second lowest level are scheduled.

4.3 Expected runtime for two processor HLF using profiles

We now use the profile notation to compute the expected run time for an intree scheduled on two processors by HLF.

4.3.1 A recursive definition

Exploiting profile notation, we can define the recursive formulae that can be used to compute the optimal expected run time. Before we do so, we introduce an abbreviation to obtain a "slice" (i.e. a continuous sub-sequence) of a profile.

Definition 4.2 (Slice of a profile)**.** *Let $P = [\![n_1, n_2, \ldots, n_{r-1}, n_r]\!]$ be a profile of length r. Then for $i, j \in \{1, 2, \ldots, r\}$ with $i < j$, we define $P[i : j] = [\![n_i, n_{i+1}, \ldots, n_{j-1}, n_j]\!]$ (which is then of length $j - i + 1$).*

We denote concatenation of profiles P_1 and P_2 by $P_1 \cdot P_2$.

This definition, for example, gives $[\![3,6,2,7,1,8]\!]\,[2:4] = [\![6,2,7]\!]$ and $[\![1,2,3]\!] \cdot [\![4,5,6]\!] = [\![1,2,3,4,5,6]\!]$. We now continue with the recursive formulae aforementioned. We use the notation $\mathbb{E}[P]$ to compute the

n	-1	0	1	2	3	4	5	Closed term
$A_0(n)$	0	1	4	12	32	80	192	$(n+1)\cdot 2^n$
$A_1(n)$	0	1	5	17	49	129	321	$n\cdot 2^{n+1}+1$
$A_2(n)$	0	1	6	23	72	201	522	$(n-1)\cdot 2^{n+2}+n+5$
$A_3(n)$	0	1	7	30	102	303	825	$(n-2)\cdot 2^{n+3}+(n^2+11n+34)/2$
$A_4(n)$	0	1	8	38	140	443	1268	$(n-3)\cdot 2^{n+4}+\binom{n+3}{3}+4\cdot\left(\binom{n+1}{2}+4n+12\right)$

Table 4.1: Example values for $A_i(n)$. For more information about these sequences, see [**oeistwoprocsA0**][**oeistwoprocsA1**; **oeistwoprocsA2**; **oeistwoprocsA3**; **oeistwoprocsA4**].

expected run time for two-processor HLF on an intree with profile P. If $P = [\![n_1]\!] \cdot P'$ is a profile, then we can compute:

$$\mathbb{E}[P] = \mathbb{E}[[\![n_1]\!] \cdot P'] = \begin{cases} r, & \text{if } P = [\![(1)^r]\!] \\ \frac{n_1-1}{2} + \mathbb{E}[P'], & \text{if } n_1 \geq 2 \\ \frac{1}{2} + \frac{1}{2}\cdot(\mathbb{E}[P'] + \mathbb{E}[SUC(P)]), & \text{otherwise} \end{cases}, \tag{4.1}$$

where $SUC([\![n_1,\ldots,n_r]\!]) = [\![n_1, n_2, n_3, \ldots, n_{j-1}, n_j - 1, n_{j+1}, \ldots, n_r]\!]$ such that j is the minimum index such that $n_j > 1$.

If we consider the second case of equation (**??**) we see that we can simplify it to the following (by combining the second and third case):

$$\mathbb{E}[P] = \mathbb{E}[[\![n_1]\!] \cdot P'] = \begin{cases} r, & \text{if } P = [\![(1)^r]\!] \\ \frac{n_1}{2} + \frac{1}{2}\cdot(\mathbb{E}[P'] + \mathbb{E}[SUC([\![1]\!] \cdot P')]), & \text{otherwise} \end{cases}, \tag{4.2}$$

with SUC as defined before.

4.3.2 Solving the recurrence for special cases

Unfortunately, the recurrence relation in equation (**??**) does not significantly simplify the original problem. However, we were able to deduce a closed form that can be used for special cases.

Theorem 4.3. *Let* $\left[\!\left[n_1, (1)^{j-2}, n_j, (1)^{r-j}\right]\!\right]$ *be a profile (i.e. at most the first and one other entry of the profile are different from 1). Then it holds that*

$$\mathbb{E}\left[\left[\!\left[n_1, (1)^{j-2}, n_j, (1)^{r-j}\right]\!\right]\right] = r + \frac{A_0(n_1-2)}{2^{n_1-1}} + \frac{A_{j-1}(n_j-2)}{2^{n_j+j-2}},$$

where A_i *is inductively defined as follows:*

$$A_0(n) = (n+1)\cdot 2^n$$
$$A_{i+1}(n) = \sum_{k=0}^{n} A_i(k)$$

Before we prove theorem **??**, let us have a look at table **??** showing values for $A_i(n)$ for small values of i and n.

From this table and by looking at the definition of $A_i(n)$ we can deduce a simple lemma that will later be useful.

Note that there are closed expressions for $A_i(n)$ for $i \leq 5$ (and possibly for higher values of i, as well). However, these formulae are quite complex (also see table **??**) and we were not able to deduce a *simple* pattern according to which $A_i(n)$ can be constructed. It seems that $A_i(n)$ involves the term $(n+1-i)\cdot 2^{n+i}$ in some way, and the remaining term seems to be a polynomial in n.

Lemma 4.4. *Let* $A_i(n)$ *be as defined in theorem* **??**. *Then, we have*

$$A_{j-1}(n) + A_j(n-1) = A_j(n).$$

Proof. Proof is trivial by definition of $A_j(n) = \sum_{k=0}^{n} A_{j-1}(k) = A_{j-1}(n) + \sum_{k=0}^{n-1} A_{j-1}(k) = A_{j-1}(n) + A_j(n-1)$. □

We can now proof theorem **??**.

Proof of theorem **??**. We prove theorem **??** by complete induction. The base case $\mathbb{E}\left[\left[\!\left[(1)^r\right]\!\right]\right] = r$ is clear because in this case we can always schedule exactly one task. Since there are r tasks in total, this results in an expected run time of r (each task is expected to be exponentially distributed with expectation 1).

We now consider the special cases where *all elements but one* in the profile are 1. That is, we consider the profile, whose elements are all 1, exctept the element at position j, which will be n. That is, we examine

$$\left[\!\left[(1)^{j-1}, n, (1)^{r-j}\right]\!\right].$$

We can rewrite this using the definition and afterwards apply the induction hypothesis:

$$\begin{aligned}\mathbb{E}\left[\left[\!\left[(1)^{j-1}, n, (1)^{r-j}\right]\!\right]\right] &= \frac{1}{2} + \frac{1}{2} \cdot \left(\mathbb{E}\left[\left[\!\left[(1)^{j-2}, n, (1)^{r-j}\right]\!\right]\right] + \mathbb{E}\left[\left[\!\left[(1)^{j-1}, n-1, (1)^{r-j}\right]\!\right]\right]\right) = \\ &= \frac{1}{2} + \frac{1}{2} \cdot \left((r-1) + \frac{A_{j-2}(n-2)}{2^{n+(j-1)-2}} + r + \frac{A_{j-1}(n-3)}{2^{(n-1)+j-2}}\right)\end{aligned}$$

We now apply lemma Lemma **??** and obtain

$$\begin{aligned}\left[\!\left[(1)^{j-1}, n, (1)^{r-j}\right]\!\right] &= \frac{1}{2} + \frac{1}{2} \cdot \left((r-1) + r + \frac{A_{j-2}(n-2) + A_{j-1}(n-3)}{2^{n+j-3}}\right) = \\ &= \frac{1}{2} + \frac{1}{2} \cdot \left(2r - 1\frac{A_{j-1}(n-2)}{2^{n+j-3}}\right) = \\ &= \frac{1}{2} + r - \frac{1}{2}\frac{A_{j-1}(n-2)}{2^{n+j-2}} = \\ &= r + \frac{A_{j-1}(n-2)}{2^{n+j-2}}\end{aligned}$$

We conclude the proof by deriving the expected run time for $\left[\!\left[m, (1)^{j-2}, n, (1)^{r-j}\right]\!\right]$. We do this by applying the definition and thereby reducing the problem to $\left[\!\left[(1)^{j-1}, n, (1)^{r-j}\right]\!\right]$:

$$\begin{aligned}\mathbb{E}\left[\left[\!\left[m, (1)^{j-2}, n, (1)^{r-j}\right]\!\right]\right] &= \frac{m-1}{2} + \mathbb{E}\left[\left[\!\left[(1)^{j-1}, n, (1)^{r-j}\right]\!\right]\right] = \\ &= \frac{m-1}{2} + r + \frac{A_{j-1}(n-2)}{2^{n+j-2}} = \\ &= \frac{(m-1)\cdot 2^{m-2}}{2^{m-1}} + r + \frac{A_{j-1}(n-2)}{2^{n+j-2}} = \\ &= \frac{A_0(m-2)}{2^{m-1}} + r + \frac{A_{j-1}(n-2)}{2^{n+j-2}}\end{aligned}$$

This concludes the proof. □

Moritz Maaß has shown another property in [**MoritzMaasDiploma**]:

Theorem 4.5 (Intrees with exactly two leaves and intrees with same profiles)**.** *Let $l, k \in \mathbb{N}$, $a \in \mathbb{N}_0$ and*

$\left[\!\left[(1)^{l-k},(2)^{k},(1)^{a+1}\right]\!\right]$ *be a profile. Then, it holds that*

$$\begin{aligned}\mathbb{E}\left[\left[\!\left[(1)^{l-k},(2)^{k},(1)^{a+1}\right]\!\right]\right] = & \sum_{i=1}^{k}\left(\frac{1}{2}\right)^{l+i-1}\cdot\binom{l+i-2}{i-1}\cdot(k-i+2)\\ + & \sum_{j=1}^{l}\left(\frac{1}{2}\right)^{k+j-1}\cdot\binom{k+j-2}{j-1}\cdot(l-j+2)\\ + & \sum_{i=1}^{k}\sum_{j=1}^{l}\left(\frac{1}{2}^{k-i+l-j+1}\cdot\binom{ki+l-j}{l-j}\right)\\ + & a.\end{aligned}$$

For a proof of the above theorem, please refer to [**MoritzMaasDiploma**].

Even if we were not able to deduce a more general formula that holds if more entries in the profile differ from 1, this might serve as a starting point for a more advanced proof.

4.4 Profile DAGs

As seen in the previous chapter, a closed formula for $\mathbb{E}\left[[\![n_1,\ldots,n_r]\!]\right]$ seems to be quite complex. This is why we may compute the expected runtime just with the recursive approach given in equation (**??**).

Of course, it is an interesting question how complex this computation is. Therefore, we consider the *profile DAG*. The profile DAG is – intuitively – a coarsening of the original snapshot DAG. It is created the following way: We "merge" snapshots having the same profile, thereby decreasing the number of vertices in the DAG. Figure **??** shows a snapshot DAG and its corresponding profile DAG.

To be more mathematical: If the snapshot DAG is (V_s, E_s), then the profile DAG can be expressed as (V_p, E_p). These are defined as follows:

$$\begin{aligned}V_p &= \{\text{PROFILE}(v) \mid v \in V_s\}\\ E_p &= \{(\text{PROFILE}(v_1),\text{PROFILE}(v_2)) \mid (v_1,v_2)\in E_s\}\end{aligned}$$

The function PROFILE takes a snapshot as input and returns the profile corresponding the the snapshot's intree. Note that PROFILE then is a homomorphism between the snapthot and the profile DAG (see [**hell2004graphs**] for more information).

If we compute the expected run time via the profiles (as equation (**??**) suggests), we only need to compute the profiles arising in the profile DAG. We can cache intermediate results, so we do not need to compute the runtime of any profile twice.

Thus, it is an important question how big these profile DAGs can get. To tackle this question, we examine for a profile $P = [\![n_1,\ldots,n_r]\!]$ how many profiles of a certain length exist as successors of this profile.

We inspect the following example: Consider the profile $[\![4,3,5,1]\!]$. Its successing profiles of length 4 are

$$\begin{aligned}&[\![3,3,5,1]\!], [\![2,3,5,1]\!], [\![1,3,5,1]\!],\\ &[\![1,2,5,1]\!], [\![1,1,5,1]\!],\\ &[\![1,1,4,1]\!], [\![1,1,3,1]\!], [\![1,1,2,1]\!] \text{ and } [\![1,1,1,1]\!].\end{aligned}$$

We recognize that the first item in the succesing profiles has to be at most 4 (since the *original* first entry was exactly 4). Moreover, the second entry can only be less then 3 (*original* second entry: 3) if the first entry is 1. Similarily, the third entry in a successing profile can only be less then 5 (original third entry: 5) if the first and the second entry are 1. We illustrated this by putting the corresponding profiles into several lines in the description above.

More general: In a successing profile, the entry at a certain position can only be less than the original entry at this position if all entries *up to that position* are already 1.

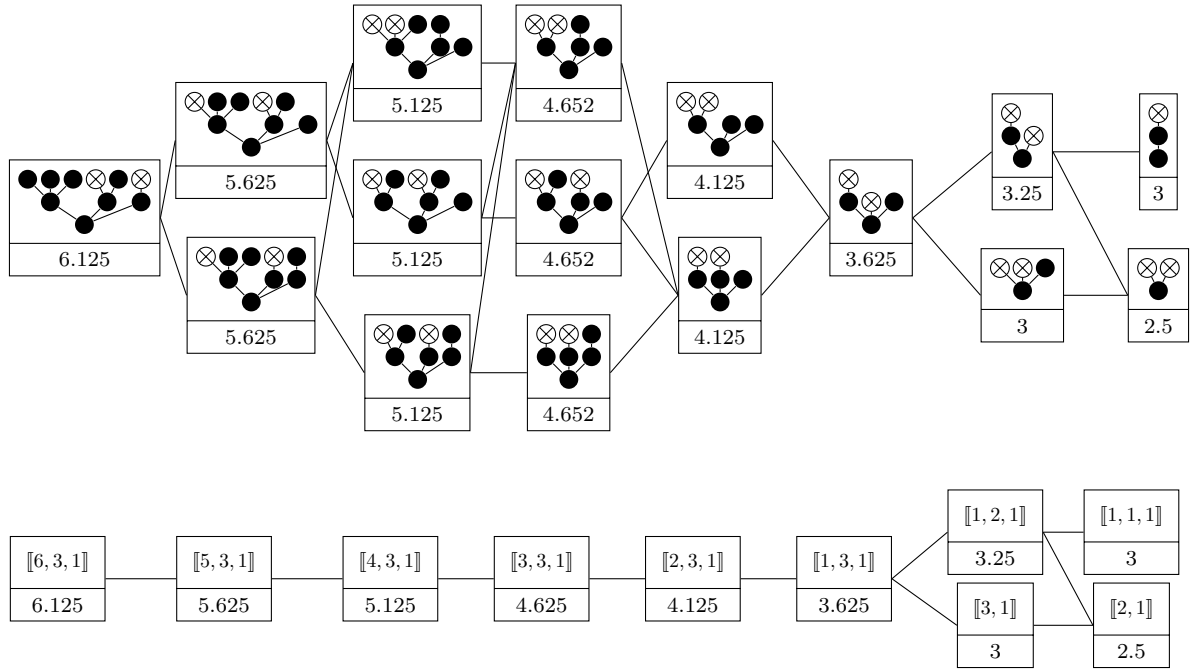


Figure 4.4: A snapshot DAG (HLF, two processors) and its corresponding profile DAG containing much less nodes than the original snaphsot DAG. Snapshots contain the respective intrees and the respective expected run times. If a snapshot has two outgoing transitions, the probability for each is 50%. If it has only one, the probability for it is, of course, 100%. Snapshots with same profiles have the same expected run time and can thus be merged.

Definition 4.6 (Successing profiles)**.** *Let $P = [\![n_1, \ldots, n_r]\!]$ be a profile of lenght r. The set of successing profiles of length j of original profile P is*

$$S_j^P := \left\{ [\![m_{r-j+1}, \ldots, m_r]\!] \mid \exists p \in \{r-j+1, \ldots, r\} \,.\, \bigwedge_{i=r-j+1}^{p-1} m_i = 1 \ \wedge m_p < n_p \right\}.$$

The set of all *successing profiles clearly is then*

$$S^P := \bigcup_{j=1}^{r} S_j^P.$$

As an example, consider the profile $[\![6, 3, 1]\!]$ whose set of successing profiles contains the following elements:

$[\![5, 3, 1]\!]$ $[\![4, 3, 1]\!]$ $[\![3, 3, 1]\!]$ $[\![2, 3, 1]\!]$ $[\![1, 3, 1]\!]$
$[\![1, 2, 1]\!]$ $[\![1, 1, 1]\!]$
$[\![3, 1]\!]$ $[\![2, 1]\!]$ $[\![1, 1]\!]$
$[\![1]\!]$

The size of the profile DAG (starting at a certain profile P) is then clearly denoted by the size of S^P (see figure **??** for reference).

Lemma 4.7 (Size of profile DAG)**.** *Let $P = [\![n_1, \ldots, n_r]\!]$ be a profile. Then,*

$$\left|S^P\right| = \sum_{j=1}^{r-1} j \cdot n_j - 0.5r^2 + 1.5r.$$

Proof. If $P = [\![n_1, \ldots, n_r]\!]$ is a profile, then the size of the set S_j^P is

$$1 + \sum_{i=j}^{r-1} (n_i - 1).$$

In a successing profile, the entry at position p can take exactly the values $1, 2, 3, \ldots, n_p - 1$ (where n_p is th entry at positon p in profile P). This yields the term $(n_i - 1)$. Moreover, we stated that the entry at position p can only be smaller if *all previous entries* are 1. Thus, we can simply sum um the terms for the different positions, yielding the above sum.

We now sum up over the different possible *lengths* (ranging from 1 to r) of the successor profiles, and afterwards apply well-known summation rules and formulae for triangular numbers:

$$\begin{aligned}
\left|S^P\right| &= \sum_{j=1}^{r} \left(1 + \sum_{i=j}^{r-1} (n_i - 1)\right) \\
&= \sum_{j=1}^{r} 1 + \sum_{j=1}^{r} \left(\sum_{i=j}^{r-1} n_i\right) - \sum_{j=1}^{r} \sum_{i=j}^{r-1} 1 \\
&= r - \sum_{j=1}^{r} (r - j) + \sum_{j=1}^{r} \left(\sum_{i=j}^{r-1} n_i\right) \\
&= r - \frac{r^2 - r}{2} + \sum_{j=2}^{r-1} \left(\sum_{i=j}^{r-1} n_i\right) \\
&= \sum_{j=1}^{r} \left(\sum_{i=j}^{r-1} n_i\right) + -0.5r^2 + 1.5r
\end{aligned}$$

It remains to be shown that $\sum_{j=1}^{r} \left(\sum_{i=j}^{r-1} n_i\right) = \sum_{j=1}^{r-1} (j-1) \cdot n_j$. This can be seen by considering the following:

$$\begin{aligned}
\sum_{j=2}^{r-1} \left(\sum_{i=j}^{r-1} n_i\right) = {} & n_1 + n_2 + n_3 + n_4 + \ldots + n_{r-2} + n_{r-1} + \\
& n_2 + n_3 + n_4 + \ldots + n_{r-2} + n_{r-1} + \\
& n_3 + n_4 + \ldots + n_{r-2} + n_{r-1} + \\
& + n_4 + \ldots + n_{r-2} + n_{r-1} + \\
& \ddots \quad \vdots + \\
& n_{r-2} + n_{r-1} + \\
& n_{r-1}
\end{aligned}$$

$$= n_1 + 2n_2 + 3n_3 + 4n_4 + \ldots + (r-2) \cdot n_{r-2} + (r-1) \cdot n_{r-1}$$

This shows that $\left|S^P\right| = \sum_{j=1}^{r-1} j \cdot n_j - 0.5r^2 + 1.5r$. □

Now that we know the size of the profile DAG, this imposes the question how many nodes a profile DAG has *in the worst case* if we consider an intree with exactly n tasks.

Lemma 4.8. *For an intree with n nodes and r levels (i.e. an intree having a profile with r entries), the maximum size of the profile DAG is reached if the profile is of the form*

$$\left[\!\left[(1)^{r-2}, n - r + 1, 1\right]\!\right].$$

Proof. We compute the number of nodes in the profile DAG resulting from an intree with profile $P^* = \left[\!\left[(1)^{r-2}, n-r+1, 1\right]\!\right]$. According to lemma **??**, it is exactly

$$\sum_{j=1}^{r-2} j \cdot 1 + (r-1) \cdot (n-r+1) = \frac{(r-1) \cdot (r-2)}{2} + (r-1) \cdot (n-r+1).$$

We consider now an arbitrary profile, but express it in terms of P^*. That is, we consider a profile $P = [\![n_1, \ldots, n_r]\!]$, where $n_i = 1 + b_i$, for $i \in \{1, 2, \ldots, r-2\}$ and $n_{r-1} = (n-r+1) + b_{r-1}$. Note that this means that each n_i (entry in P) is expressed as the corresponding entry in P^* plus some constant b_i chosen appropriately. Of course $n_r = 1$.

Two observations are important now:

- Since all $n_i \in \mathbb{N}$, we can condlude that $b_i \geq 0$ for $i \in \{1, 2, \ldots, r-2\}$.
- Since the number of all tasks has to be the same, we know that P^* and P must have the same number of tasks (namely n). This means that

  $$n = \sum_{i=1}^{r} n_i = \sum_{i=1}^{r-2} [1 + b_i] + [(n-r+1) + b_{r-1}] + 1.$$

  We simplify the above to

  $$\sum_{i=1}^{r-2} [1 + b_i] + [(n-r+1) + b_{r-1}] + 1 = \sum_{i=1}^{r-2} b_i + r - 2 + n - r + 1 + b_{r-1} + 1 = n + \sum_{i=1}^{r-1} b_i$$

  From this we can conclude that $b_{r-1} = -(\sum_{i=1}^{r-2})$.

We can now compute the number of nodes for profile P (remember: an arbitrary profile expressed through P^* and the b_i's):

$$\begin{aligned}
\sum_{j=1}^{r-1} j \cdot n_j &= \sum_{j=1}^{r-2} j \cdot (1 + b_j) + (r-1)(n-r+1+b_{r-1}) \\
&= \sum_{j=1}^{r-2} j + \sum_{j=1}^{r-2} j \cdot b_j + (r-1) \left(n - r + 1 - (\sum_{j=1}^{r-2} b_j) \right) \\
&= \frac{(r-2) \cdot (r-1)}{2} + (r-1)(n-r+1) + \sum_{j=1}^{r-2} j \cdot b_j - \sum_{j=1}^{r-2} (r-1) \cdot b_j \\
&= \underbrace{\frac{(r-2) \cdot (r-1)}{2} + (r-1)(n-r+1)}_{\text{Number of nodes in DAG for } P^*} + \sum_{j=1}^{r-2} b_j \cdot (j-r+1)
\end{aligned}$$

We recognize that the result contains the number of nodes in a profile DAG for profile $P^* = \left[\!\left[(1)^{r-2}, n-r+1, 1\right]\!\right]$ — and some additional term (namely $\sum_{j=1}^{r-2} b_j \cdot (j-r+1)$).

However, in this term we can see that $(j-r+1) < 0$ (since j ranges from 1 to r-2). This means, that the whole sum $\sum_{j=1}^{r-2} b_j \cdot (j-r+1) \leq 0$ (since all b_j are positive for $j \in \{1, 2, \ldots, r-2\}$). That means, this sum gets 0 *if and only if* all b_j are 0 for $j \in \{1, 2, \ldots, r-2\}$.

This, however, proves that the profile P^* is the profile with exactly r entries and n tasks that maximizes the number of nodes in the profile DAG. □

It remains to explain how to choose r in a way such that given the number of tasks n, we can construct a profile with r entries such that the resulting profile DAG has the maximum number of nodes over *all* intrees with n tasks.

Lemma 4.9 (Structure of worst-case profile)**.** *Given a natural number n, the profile maximizing the number of nodes in the profile DAG is of the form $P^* = \left[\!\left[(1)^{r-2}, n-r+1, 1\right]\!\right]$, where r is either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$ (one of both — can be chosen at will).*

Proof. The fact that the profile maximum the number of nodes in the profile DAG is of the form $\left[\!\left[(1)^{r-2}, n-r+1, 1\right]\!\right]$ follows directly from lemma **??**. Thus, we can restrict ourselves onto those if we look for profiles maximizing the number of nodes in a profile DAG for a certain number of tasks.

We consider the number of nodes in a profile DAG for a profile of the form $\left[\!\left[(1)^{r-2}, n-r+1, 1\right]\!\right]$, given by lemma **??**:

$$\sum_{j=1}^{r-2} j \cdot 1 + (r-1) \cdot (n-r+1) - 0.5r^2 + 1.5r = \frac{(r-2)\cdot(r-1)}{2} + (r-1)\cdot(n-r+1) - 0.5r^2 + 1.5r$$

Our goal is now to maximize this term by choosing r accordingly depending on n. We simplify the above to

$$r + r\,(n-r+1) - 1 = -r^2 + r \cdot (n+2) - 1$$

and recognize that this is a downward opened parabola having the derivative

$$3 + n - 2r,$$

meaning that its maximum is at $r = \frac{n+2}{2}$. Since this is only a natural number if n is even and since we have – as said before – a parabola, we can simply apply rounding to get the maximum for natural-values. This means, we can derive natural solution: $r^* = \lfloor \frac{n+2}{2} \rfloor$ or $r^* = \lceil \frac{n+2}{2} \rceil$ (can be chosen at will since we have a parabola). Without loss of generality, we focus on $r^* = \lfloor \frac{n+2}{2} \rfloor = \lfloor \frac{n}{2} \rfloor + 1$.

This means that the profile DAG has a maximum of nodes if we consider the profile

$$\left[\!\left[(1)^{r^*-2}, n-r^*+1, 1\right]\!\right] = \left[\!\left[(1)^{(\lfloor \frac{n}{2} \rfloor + 1)-2}, n - \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) + 1, 1\right]\!\right] = \left[\!\left[(1)^{\lfloor \frac{n}{2} \rfloor - 1}, \left\lceil \frac{n}{2} \right\rceil, 1\right]\!\right]$$

Similarily, we could have chosen $r^* = \lceil \frac{n}{2} \rceil + 1$. This shows the claim. □

We can now combine the lemmata to the following theorem:

Theorem 4.10 (Maximum size of profile DAG)**.** *For an intree with exactly n tasks, the profile DAG has at most $\lfloor \frac{n}{2} \rfloor \cdot \lceil \frac{n}{2} \rceil + 1$ nodes.*

Proof. Lemma **??** gives us that the worst-case profile is $P^* = \left[\!\left[(1)^{\lfloor \frac{n}{2} \rfloor - 1}, \lceil \frac{n}{2} \rceil, 1\right]\!\right]$, which we use to compute the number of nodes in the corresponding worst-case profile DAG according to lemma **??**:

$$\frac{\left(\left(\lfloor \frac{n}{2} \rfloor + 1\right) - 2\right) \cdot \left(\left(\lfloor \frac{n}{2} \rfloor + 1\right) - 1\right)}{2} + \left(\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) - 1\right) \cdot \left(\left\lceil \frac{n}{2} \right\rceil\right) - \frac{1}{2} \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)^2 + \frac{3}{2} \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)$$

We simplify the above to

$$\frac{\left(\lfloor \frac{n}{2} \rfloor - 1\right)\left(\lfloor \frac{n}{2} \rfloor\right)}{2} + \left\lfloor \frac{n}{2} \right\rfloor \cdot \left\lceil \frac{n}{2} \right\rceil - \frac{\left\lfloor \frac{n}{2} \right\rfloor^2 + 2 \cdot \lfloor \frac{n}{2} \rfloor + 1}{2} + \frac{3 \cdot \lfloor \frac{n}{2} \rfloor + 3}{2} \quad = \quad \left\lceil \frac{n}{2} \right\rceil \cdot \left\lfloor \frac{n}{2} \right\rfloor + 1,$$

proving the claim. □

4.5 Snapshot DAG

Theorem **??** shows in particular that the worst-case size of a profile DAG is quadratic in the number of tasks of the intree. Additionally, we can use use the theorem to derive a simple, loose bound on the size of the original snapshot DAG resulting from a HLF schedule for two processors.

Corollary 4.11 (Upper bound for the size of the snapshot DAG for P2)**.** *The size of a snapshot DAG for an intree containing n tasks and resulting from HLF scheduling for two processors is $O(n^4)$.*

Proof. Each profile contains at most n tasks (actually, *all but on* profile contain less than n tasks).

We recognize that there are less than n^2 snapshots resulting in the same profile P containing n tasks. This is clearly the case, because, if two processors are present, there are at most $\binom{n}{2} < n^2$ possibilities to choose the two tasks to be scheduled out of at most n tasks.

We now consider a worst case snapshot DAG, having – according to theorem **??** – size $\lfloor \frac{n}{2} \rfloor \cdot \lceil \frac{n}{2} \rceil + 1 \leq \frac{n^2}{4}$. Since each of these profiles corresponds to less than n^2 snapshots, it directly follows that the number of nodes in the snapshot DAG is less than $\frac{n^2}{4} \cdot n^2 = \frac{n^4}{4} \in O(n^4)$. □

The above bound is far from being tight. Still, it suffices to show that the size of a snapshot DAG can *not* be exponential in the number of nodes.

This, on the other hand, implies a poly-time algorithm that can be used to determine the expected run time for a given intree of tasks whose run times are exponentially distributed: We simply construct the whole snapshot DAG according to HLF (whose size is $O(n^4)$, i.e. polynomial) and compute the expected run time recursively for each snapshot.

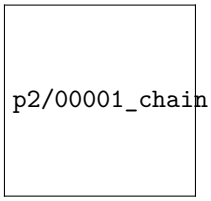
4.6 Summary

We have shown that we can compute the optimal expected run time in polynomial time and – moreover – that we can compute the exptected run time for special cases using relatively easy formulas.

We have also seen that we can consider profile DAGs instead of whole snapshot DAGs and can rely on them. In fact, it can be shown that – for a given intree – we can rely on an even simpler DAG that does not contain the profiles, but instead only the *length* of the corresponding profile (i.e. implicitly the length of the longest path from the root to one leaf) and the number of overall tasks.

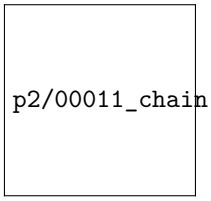
However, this simplification is only valid "within the snapshot DAG for a given intree", in the sense that there are different intrees with same profile length and same number of nodes aside this longest chain, but having different run times. Figure **??** shows two such intrees (namely $(0, 0, 0, 0, 1)$ and $(0, 0, 0, 1, 1)$).

(a) Snapshot DAG for HLF on intree $(0, 0, 0, 0, 1)$ resulting in an overal run time of 4.0625.



(b) DAG containing only the length of the longest chain and the number of tasks *not* on this chain for HLF on intree $(0, 0, 0, 0, 1)$ resulting in an overal run time of 4.0625.

(c) Snapshot DAG for HLF on intree $(0, 0, 0, 1, 1)$ resulting in an overal run time of 4.125.



(d) DAG containing only the length of the longest chain and the number of tasks *not* on this chain for HLF on intree $(0, 0, 0, 1, 1)$ resulting in an overal run time of 4.0625.

Figure 4.5: Comparing snapshot DAGs to DAGs that contain only the length of the longest chain and the number of tasks *not* on this chain. This example shows that there are distinct intrees both having depth three and three tasks not on the longest chain, but having different optimal expected run times.

Chapter 5

Implementation details

While chapter **??** focussed more on theoretical aspects, this chapter sums up some information about the implementation. We explain why we chose C++ and Python to implement the project and present some central algorithms that were developed during this project. In addition, we provide asymptotic run times for the single algorithms.

5.1 Choice or programming language(s)

The first question we faced was the programming language to use. We first tried to develop a prototype with Maple, because Maple already supports many mathematical concepts (in particular graph theory and probability theory) that might later be useful.

However, we quickly realized that probably Maple has some downsides for this use case: We needed something that was able to cope with quite large amount of data. While Maple might be able to do so if you are experienced enough, we went for C++, sacrificing the elegant notation of Maple, but gaining speed and control over how data are organized in memory. We also considered simple C, but quickly discarded it, because we did not want to implement some basic data structures ourselves, but instead relied on well-tested implementations (especially of Maps and Vectors). As a bonus, there are many good C++ libraries out there, some of which we made use of (Boost and GMP).

Additionally, we looked a simple language to solve several non-critical one-time tasks, such as e.g. generating a database containing all (distinct) intrees or some data analysis. We chose Python.

5.2 General structure

The algorithmic core works in two main steps:

- Computing *all* schedules for an intree allowed by a specific scheduling strategy.
- Optimizing the generated schedules.

Before starting, the usual amount of command-line parsing and initialization is done, and afterwards, we can generate output files and clean up. We will now describe in short the two most important steps when generating an optimal schedule.

5.2.1 Generating all schedules according to a specific strategy

Our program basically starts off given a single intree, a certain number of processors and a scheduling strategy. According to this strategy, we then compute all initial combinations of scheduled tasks, i.e. all initial snapshots (up to equivalent snapshots).

Starting at an initial snapshot, we then recursively compute all its successors, i.e. we simulate which task finishes first and wich task shall be chosen next according to the given scheduling strategy. All of

these events (task finishes, next task is chosen) are associated with a certain probability. We do this until we reach the intree containing only 1 task, which then denotes the end of the scheduling process. We thereby eliminate equivalent snapshots in order to save memory.[1]

After the whole snapshot DAG has been computed we can compute the expected run times of the initial snapshots (recursively, as explained in section **??**).

5.2.2 Optimizing schedules

Moreover, the program is capable of "excluding the scheduler's bad choices", i.e. it determines where the scheduler has chosen a good and where it has chosen a bad task. Then, the program discards the bad choices and leaves only the good choices, yielding a (possibly) new schedule that is optimal for a given intree.

In order to keep the program fast, we thereby had to cache quite a lot of results (since the recursions through the snapshot DAG would otherwise requiring computing the same things over and over). This, of course, leads to increased memory consumption.

We implemented support for various representation of probabilities and expected run times, most notably floating point numbers. Other possible representation include doubles or rationals (through the use of e.g. the GNU Multiple Precision library).

5.3 Representation of intrees and snapshots

For the representation of intrees and snapshots, we tried to find a good balance between ease of use, memory consumption and speed. We thereby rely on well-tested data structures from the C++ standard library. We now describe on a very high level how we implemented the needed data structures.

Intrees are basically maintained as a collection of edges that – implicitly – also store the vertices resp. tasks. We store the edges in an array such that an edge (i, j) is stored as $array[i] = j$, i.e. the start of the edge is used as the array index while the target of the edge is used as the element at that particular position in the array. This is possible since we are dealing with intrees only and enables a quick lookup on which task is a successor of a given task.

Snapshots store the following things:

- The corresponding intree (in normalized form – see section **??**).
- The set of currently scheduled tasks.
- The set of its successors in the snapshot DAG and the corresponding transition probability.

Additionally, we cache many results that are frequently needed and expensive to recompute, such as the expected run time. In order to be able to reuse snapshots so that we do not recompute results for equivalent snapshots, we need some data structure that enables us to retrieve an equivalent snapshot if it already has been constructed. Therefore we introduced a pool that manages all snapshots and can be used to avoid duplicates of (equivalent) snapshots.

5.4 Computing equivalent snapshots

As mentioned in section **??**, it is possible to combine certain snapshots into one single snapshot, thereby avoiding redundant computations. The requirements for two snapshots being equivalent have been discussed there.

It is a notable fact that we can determine in polynomial time (w.r.t. the number of nodes) whether two snapshots are equivalent. This computation involves foremost a check whether the two corresponding intrees are isomorphic.

[1] We also implemented a variant that does not eliminate equivalent snapshots. However, this variant quickly became too slow as the number of tasks increased. This is why we focussed on the variant that eliminates equivalent snapshots.

While there currently is no known algorithm that checks isomorphism in polynomial time *for general graphs* (see [**arora2009computational**]), there is a simple algorithm for isomorphism of intrees. The idea behind this algorithm is to recursively sort the predecessors of a task according to the number of their respective predecessors. An early description of a possible algorithm can be found e.g. in [**aho1974design**]. We can easily adapt this isomorphism check to our needs in the sense that we can construct an algorithm that constructs a *canonical snapshot*, and all equivalent snapshots are converted to exactly this canonical snapshot.

One thing to keep in mind is that we have to explicitly take care of the currently scheduled tasks, i.e. we have to adopt the algorithm to distinguish between scheduled and unscheduled tasks.

5.4.1 The algorithm

The algorithm basically relies on an ordering of intrees (there are of course many possible, but we use a simple one). To be able to compute equivalent snapshots, this ordering has to incorporate which tasks are currently scheduled. Thus, we use an ordering that uses a set of tasks (the scheduled tasks), and does not solely rely on the structure of the intree.

Definition 5.1 (Ordering of intrees with preferred tasks)**.** *We introduce an ordering denoted by* $\overset{X}{\geq}$*. For two intrees* I_1 *and* I_2 *and a set* X *of tasks, we define* $I_1 \overset{X}{\geq} I_2$ *inductively.*

- *If both* I_1 *and* I_2 *consist only of a root, we have* $I_1 \overset{X}{\geq} I_2$ *if and only if the root of* I_2 *is not in* X *or the root of* I_1 *is in* X*.*
- *If the root of* I_1 *has more predecessors than the root of* I_2*, then* $I_1 \overset{X}{\geq} I_2$*.*
- *If the root of* I_1 *has the same number* r *of predecessors as the root of* I_2*, we sort the corresponding predecessors* $p_1, \ldots, p_r$ *resp.* $q_1, \ldots, q_r$ *(in* I_1 *resp.* I_2*) according to* $\overset{X}{\geq}$*. Then, if* $p_i \overset{X}{\geq} q_i \forall i \in \{1, 2, \ldots, r\}$*, we have* $I_1 \overset{X}{\geq} I_2$*.*

TODO: Examples!

An algorithm to compute a *canonical form* of a snapshot, can recursively sort the tasks in the intree according to the ordering $\overset{X}{\geq}$. It is shown in algorithm **??**.

Algorithm 5.1 Computing canonical snapshots for a snapshot s containing the corresponding intree and the tasks that are currently scheduled (as defined in section **??**).TODO: Algorithmus verbessern.

```
procedure CanonicalSnapshot(s)                ▷ Returns the canonical snapshot for snapshot s
    t ← s.intree                                                    ▷ Retrieve root of intree
    X ← s.scheduled                                                 ▷ Retrieve scheduled tasks
    return CanonicalIntree(t, X)
end procedure

procedure CanonicalIntree(t, X)               ▷ t: a (sub)tree, X: set of scheduled tasks
    r ← t.root                                                      ▷ Retrieve root of subtree
    CanonicalPredecessors ← {CanonicalIntree(c, X) | c ∈ r.predecessors}
    return root with predecessors in CanonicalPredecessors in sorted order according to ≥^X
end procedure
```

Algorithm **??** is good for visualizing how the algorithm works. One disadvantage, however, is that recursively comparing two subtrees can be expensive. Nonetheless, there is a modification of this algorithm, that is more along the lines of [**aho1974design**] and has a slightly better asymptotic run time. It is shown in algorithm **??**.

Algorithm **??** is a very high level description that requires some explanations:

Algorithm 5.2 Computing a canonical intree with a better asymptotic run time

```
procedure CanonicalIntree(I, X)
    L ← number of levels in I
    Label all leaves with "0" if they are not in X, otherwise with "1"
    for i = L − 2 ... 0 do
5:      for Each non-leaf t in level i with predecessors p_1, ..., p_r do
            Label t by a (label(p_1), ..., label(p_r))        ▷ Concatenate sorted labels of predecessors
        end for
        Sort tasks on level i according to their label
        Replace labels for tasks on level i by integers geq1 such that same labels obtain same numbers

10:  end for
end procedure
```

- For this algorithm, it is convenient to have a canonical intree structure in which each task directly knows its predecessors. We can transform our representation (as explained in section **??**) into this one in $O(n)$.
- Concatenating two labels can be done in $O(1)$ (either amortized by e.g. using dynamic arrays or simply using linked lists with some additional information such e.g. a pointer to the last node in the list). Thus, the overal costs arising from line **??** (resp. for the whole inner loop) are $O(n)$ because the label of each task is subject to concatenation at most once. Moreover, note that the predecessors $p_1, \ldots, p_r$ are sorted (they have been sorted in the previous iteration of the loop).
- Line **??** ensures that the labels of tasks do not become very long. To be precise, if level i has exactly l_i tasks, the labels for level i are at most the numbers 0 to l_1.
- Line **??** from algorithm **??** shall sort in the following way: Shorter labels come first, and if two labels have the same length, they are compared lexicographically. This sorting can be implemented in $O(l_{i+1} \cdot \log l_{i+1})$, where i denotes the index from the algorithm (i.e. the current level number) and l_i denotes the number of tasks of I on level i. This can be explained as follows:
  - We use radix sort to sort the tasks according to their labels. The labels for the l_i tasks on level i are composed of the labels on level $i+1$.
  - Inductively, the labels on level $i+1$ are at most the numbers 0 to l_{i+1} (because there are l_{i+1} tasks on level $i+1$ and there might be leaves on that level), thus requiring $O(\log l_{i+1})$ bits per label.
  - Since we concatenate labels from level $i+1$ (or simply assign them label 0) to obtain the new labels on level i, we have l_i labels of total length at most $l_{i+1} \cdot \log(l_{i+1})$.
  - Now, we use a modified radix sort: We first distribute the labels according to their labels into different buckets (can be done in $O(1)$ per label if we e.g. store its length appropriately). We can now radix-sort each bucket. Since each digit of each label of a label is examined *exactly once*, we can sort all labels in $O(l_{i+1} \cdot \log l_{i+1})$ (since this is the total length of all labels on level i).
  - The tree can be sorted top-down in $\sum_{i=0}^{L-2} O(l_{i+1} \cdot \log l_{i+1}) = O(n \cdot \log n)$.
- Replacing labels can be done in $O(n)$ in total since each tasks gets assigned a label only once.

Theorem 5.2. *The canonical intree of an intree with n tasks can be constructed in $O(n \cdot \log n)$.*

Proof. See considerations above. □

Remark: This algorithm is sometimes said to have runtime $O(n)$ – assuming that we can express labels as integers and ignoring the fact that the total length of a label is $\log(l_{i+1})$ bits. This is, especially

in the case we are considering, a very reasonable assumption because an integer nowadays has at least 32 bit, meaning that the length of a label (i.e. an integer) is only exceeded if a level of the intree contains more than 2^{32} tasks.

5.4.2 Additional approaches

We experienced that computing canonical snapshots has a major impact on the overal performance of our program, which is why we tried other approaches to tackle this problem. For completeness, we will explain them shortly. Unfortunately, they did not work out as good as expected, but maybe they are helpful for future work. In this section, we focus on the part concerning the computation of *canonical intrees* (i.e. we do not distinguish between scheduled and non-scheuled leaves).

Tree sequence As we have seen in previous chapters, we can describe intrees as a sequence of tasks $(x_1, \ldots, x_n)$ such that task i is a requirement for task x_i (see section **??**). It is clear that we can assume certain facts about these sequences: We could e.g. assume that w.l.o.g. the sequence $(x_1, \ldots, x_n)$ is non-decreasing (simply assign number by a breadth first search started by task 0). We tried to exploit the knowledge about tree sequences.

Most of the time, we are generating a certain subtree from a tree whose tree sequence is already known. We then want to know the canonical intree of this subtree. We tried exploiting the fact that we already could *start* with a canonical intree. That is, we tried to construct the tree sequence of a canonical subtree by just examining the tree sequence of the original intree.

It seems that we can compute the tree sequence of an intree $I \setminus \{t\}$ (where t is a leaf of the intree I) in some cases by simply removing the corresponding entry in the tree sequence. But iterating this (i.e. removing two or more tasks one after another) often leads to problems.

We could not devise a simple pattern that works as a general rule. The problem for this approach is that – even if it might work for pure intrees – it seems to be very complex to incorporate the fact that isomorphic intrees with a different set of scheduled tasks might or might not yield equivalent snapshots. Consider e.g. the intree $(0, 0, 0, 1, 1, 2, 2, 3, 3)$ with tasks 4, 6 and 8 scheduled (shown in figure **??**). For the situation shown, it does not matter which task is the first to finish — each resulting snapshot is equivalent to all others. However, the tree sequences obtained by removing the respective tasks are $(0, 0, 0, 1, 2, 2, 3, 3)$, $(0, 0, 0, 1, 1, 2, 3, 3)$ and $(0, 0, 0, 1, 1, 2, 2, 3)$.

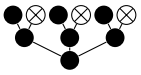


Figure 5.1: The tree sequences for the snapshots resulting when one task finishes are different, but the snapshots are still equivalent.

Matula numbers Intrees and their encodings have of course already been subject to research. One example is [**matula1968natural**], where a (more or less) natural bijection between intrees and natural numbers, the so-called *Matula numbers*, is shown.

These numbering is inductively defined as follows: A single leaf (resp. a tree consisting of a single root) obtains the number 1. A root with predecessors $t_1, \ldots, t_r$ with corresponding numbers $n_1, \ldots, n_r$ obtains the number $\prod_{i=1}^{r} p(n_i) = p(n_1) \cdot p(n_2) \cdot \cdots \cdot p(n_r)$, where $p(x)$ denotes the x-th prime number. See figure **??** for an example.

While this description could lead to very elegant algorithms in a theoretical setting, it does not come in handy in practice because the matula numbers can be very big (more than 32 bit needed for matula numbers of 15-node intrees): As shown in [**onmatulanumbers**], the largest Matula number for an intree with $n \geq 5$ tasks is $p^{(n-4)}(8)$, where

$$\begin{aligned} p^{(0)}(x) &= x, \\ p^{i}(x) &= p\left(p^{i-1}(x)\right). \end{aligned}$$

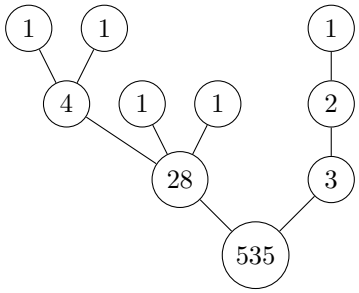


Figure 5.2: A tree whose vertices are labelled by the corresponding matula numbers. Consider e.g. the node 4, whose predecessors are two nodes labelled 1. Thus, we can compute $4 = p(1) \cdot p(1)$. The node labelled 28 can be computed by $28 = p(4) \cdot p(1) \cdot p(1) = 2 \cdot 2 \cdot 7$. The node 535 obtains its label by $535 = p(28) \cdot p(3) = 107 \cdot 5$.

The growth of $p^{n-4}(3)$ is immense as n increases. We can see from table **??** that for intrees with 14 tasks, we exceed the size of a 32-bit unsigned integer, and for intrees with 20 tasks, the largest corresponding matula number even exceeds a 64-bit integer (table **??** shows the values of $p^{(n-4)}(8)$ for n up to 20 — if these values were too large, we restricted ourselves to the logarithm (base 2) of the respective values to determine how many bits are needed).

n	$p^{(n-4)}(8)$	n	$p^{(n-4)}(8)$	$\log_2\left(p^{(n-4)}(8)\right)$
5	19	13	997525853	< 30
6	67	14	22742734291	> 34
7	331	15	$\approx 5.92852 \cdot 10^{11}$	> 39
8	2221	16	n.a.	> 43
9	19577	17	n.a.	> 49
10	219613	18	n.a.	> 54
11	3042161	19	n.a.	> 59
12	50729129	20	n.a.	> 65

Table 5.1: Values of $p^{(n-2)}(3)$ for certain values of n.

Note that even intrees with fewer than 20 tasks can have matula numbers greater than $p^{(20-4)}(8)$, thus even worsening the scenario.

In addition to the huge numbers that would need to be multiplied, computing matula numbers efficiently would require us to quickly compute $p(n)$. This would probably best acchieved using a lookup table, but this would require a huge amount of memory.

5.5 Computing all schedules according to a specific scheduling strategy

We have seen how we can compute equivalent snapshots resp. a canonical representation of a snapshot. We now describe in short how this can be used to generate all schedules according to a specific scheduling policy.

The program starts out with a given intree and the scheduler generates a list of possible initial settings describing which tasks shall be chosen. This gives us a list of *initial snapshots* where each snapshot contains the original intree and a set of tasks that shall be scheduled initially. We then process one initial snapshot after another. We describe now what we have to do for each initial snapshot.

When we have a snapshot with a certain set of scheduled tasks, the program computes the intrees that occur if one of the tasks finishes. This has to be done for every currently scheduled task. That means, that for every currently scheduled task that could finish, we obtain a new intree *and* also a set of

tasks that are scheduled (since we are considering non-preemtive scheduling, we have to keep the other tasks scheduled).

For each of the newly generated intrees, the scheduler determines which task shall be chosen with which probability in the next step, and generates new snapshots this way. The original snapshot stores pointers to the newly generated snapshots, associated with the respective probabilities of reaching them.

However, as aforementioned, it is useful to compute a canonical version of each snapshot, thereby eliminating quite a large amount of snapshots for which we would – essentially – compute the same things twice. This requires us to manage a *snapshot pool* where we store all snapshots that we have encountered up to now. If a newly generated snapshot is already is in this pool, we do not continue computations on this newly generated snapshot but instead re-use the original one.

This, on the other hand, requires us to quickly find out whether a snapshot has already been generated or not (and also to retrieve its address in memory). There are several ways to acchieve this. We are using a technique that maps snapshots onto the respective addresses. We therefore basically encode the snapshot's intree and the scheduled tasks as a sequence of bits. We could do this using ordinary `int`s or `long`s (or their `unsigned` equivalents in C++), but we decided to encode them using C++'s `vector<unsigned char>`s in connection with a list of the currently scheduled tasks.

We went for this solution for several reasons:

- The number of intrees grows rapidly, thus quickly exceeding the range of a (32-bit) integer [**oeisrootedtrees**].
- An encoding of intrees as integers in a way such that we do *not* "waste" some encodings may be very cumbersome to implement (we could use the aforementioned Matula numbers, but as said, these are not very handy in practice — see table **??**).

Using a `vector` of course increases memory consumption compared to e.g. using an ordinary `long` but is much more convenient when it comes to implementation details and – at least in theory – provides a much larger range that a single `int` or `long`.

We basically describe the current intree storing the same notation as shown in this work (just stored in a `vector` of `unsigned char`s). Of course, there are more memory-efficient solutions, but since we did never experience any problems regarding memory, we went for this solution.

As mentioned in section **??** we have $O\left(|\{T \mid T \subseteq I\}| \cdot n^p\right)$ snapshots for an intree with n tasks scheduled on p processors. We can simply call this $O\left(|D|\right)$ where D denotes the snapshot DAG for a certain scheduler. Since we compute the *canonical snapshot* for each of those snapshots and the run time to compute the canonical snapshot is $O(n \log n)$, we can deduce that the overall run time to compute all schedules for a given scheduler is bounded by $O(|D| \cdot n \log n)$.

Remark: The run time for the version *without* canonical snapshots could be denoted by $O(|D_{\text{non-canonical}}|)$, i.e. without the term $n \log n$ introduced by computing the canonical snapshots. Note that in this case, the size of the snapshot DAG (denoted by $|D_{\text{non-canonical}}|$) is remarkably bigger than the size of the snapshot DAG after merging equivalent snapshots. In practice, this behaviour became evident quite quickly.TODO: Benchmarks: Canonical vs. non-canonical!

5.6 Extensibility of the program

We now describe in short how the program can be extended. We describe the possible adjustments and how complex they are.

5.6.1 Easy adjustments

Number representations As mentioned in section **??**, we already support different possibilities to represent probabilities and expected values, namely `float`s, `double`s, and rational numbers using Boost Rational Number Library [**boostrational**] or the GNU Multiple Precision Arithmetic Library [**gnumultiprecision**]. These can easily be extended to other data types if they have overloaded operators and can be used like ordinary *float*s.

Scheduling strategies We provide an interface to implement new scheduling strategies. New scheduling strategies have to provide (most important) two methods: One to generate the initial chioces of tasks to be chosen and one method to determine which task should be the next to be chosen. We thereby focus on non-preemtive schedulers.

Exporters An important part of the program consists of code for exporters to various formats (most notably to LaTeX (in connection with TikZ)) and the program provides a straightforward interface for new exporting methods.

5.6.2 Complex adjustments

Other distributions While our program is developed around tasks having exponentially distributed processing times, we offer an interface that could – in principle – deal with other distributions and other probabilities. While this adjustment is not particularly difficult to implement, one should keep the following in mind:

- Having other distributions probably destroys memorylessness, and, thus, probably prevents us from excluding equivalent snapshots. However, excluding equivalent snapshots is one of the main features of the program.
- While it is not particularly difficult to *implement* other distributions, it might still be difficult to *calculate* the corresponding expected values and probabilities which task finishes first.

Preemtive scheduling While our schedulers focus on preemtive scheduling, it is possible to adapt the program for non-preemtive scheduling. However, this would require to rewrite some interfaces and to adjust the schedulers.

General DAGs instead of intrees It should be possible to adapt the program to work with arbitrary DAGs instead of simple intrees. However, isomorphism for directed acyclic graphs is GI-complete, meaning that it is at least as hard as the general graph isomorphism problem [**graphisomorphismproblem**]. While it *might* still be possible to exclude equivalent snapshots (because we have some information about the isomorphism we are looking for by the currently scheduled DAGs), it would probably require some tuning.

5.7 Enumerating all intrees with a certain number of nodes

It is clear that the number of intrees (more precisely, the number of unlabelled rooted trees) with exactly n nodes is exponential in n $(1, 1, 2, 4, 9, 20, 48, 115, \dots$ — see e.g. [**flajolet2009analytic**] for a derivation of the sequence or refer to [**oeisrootedtrees**]). However, for experimental purposes, it is convenient to have an algorithm that is capable of enumerating all these intrees. The main thing that should be kept in mind is that we do *not* generate isomorphic intrees over and over again.

We now show an algorithm to generate *all* intrees with a certain number of nodes (called n) up to isomorphism. This algorithm is based on the following two facts:

- The overall root can have any amount of children between 1 and $n-1$. If it has only 1 child, the corresponding predecessor intree must contain exactly $n-1$ vertices. If it has exactly $n-1$ children, each predecessor intree contains exactly 1 vertex. All the cases in between admit several possibilities.
- If the overall root of the intree with n verices has exactly r predecessors (with $r \in \{1, 2, \dots, n-1\}$, as stated before), then the sum of the vertices with in the predecessor intrees is exactly $n-1$. Moreover, let us denote the predecessor intrees by $T_1, T_2, \dots, T_r$ and call n_i the number of vertices in predecessor intree T_i for all $i \in \{1, 2, \dots, r\}$. This implies that $n_1 + n_2 + \dots + n_r = n-1$. Without loss of generality, we can assume $1 \leq n_1 \leq n_2 \leq n_3 \leq \dots \leq n_r$ (we sort the root's predecessors by the number of tasks).

Remark: It is clear that the number of possibilities P_n to decompose the value $n-1$ into r summands (also called *partition*) grows very fast. While – according to [**concretemathematics**] – there is no closed form for P_n, the first values for P_n ($n = 0, 1, 2, \dots$) are $1, 1, 2, 3, 5, 7, 11, 15, 22, 30, 42, 56, 77$ (see [**oeispartitionnumbers**]).

We can exploit these two facts to construct a recursive algorithm which is described in algorithm **??**. This algorithm enumerates all intrees with exactly n vertices. It does so by traversing all tuples $(n_1, \dots, n_r)$ fulfilling

$$n_1 + n_2 + \cdots + n_r = n - 1 \quad \text{and} \quad 1 \leq n_1 \leq n_2 \leq \cdots \leq n_r.$$

It then (recursively) generates all combinations of predecessor intrees $(p_1, \dots, p_r)$ whose respective number of nodes are $n_1, \dots, n_r$. The algorithm thereby omits duplicate combinations. This can easily be acchieved by defining an order $\left(\overset{\text{X}}{\geq}\right)$ on intrees as follows (t_1 and t_2 being two intrees with roots r_1 resp. r_2 and roots' predecessors $p_{1,1} \dots, p_{1,r}$ resp. $p_{2,1}, \dots, p_{2,r}$):

$$t_1 \overset{\text{X}}{\geq} t_2 \equiv (t_1 \text{ has more vertices than } t_2) \vee \exists k \in \{1, 2, \dots, r\} \,.\, \left(p_{1,k} \overset{\text{X}}{\geq} p_{2,k} \wedge \forall i < k . p_{1,i} = p_{2,i}\right) \tag{5.1}$$

Algorithm 5.3 Generating all intrees up to isomorphism

```
    procedure GenerateIntrees(n)          ▷ Returns the set of all intrees with exactly n vertices
        if n = 1 then
            return {●}                    ▷ Base case: Intree with just 1 vertex
        end if
 5:     R ← {}                            ▷ Variable for result
        for (n_1, ..., n_r) ∈ {(n_1, ..., n_r) ∈ ℕ^r | 1 ≤ r < n ∧ 1 ≤ n_1 ≤ n_2 ≤ ··· ≤ n_r} do
            P ← (GenerateIntrees(n_1),...,GenerateIntrees(n_r))   ▷ Predecessor intrees
            for (p_1, ..., p_r) ∈ P[1] × P[2] × ··· × P[r] do
                if p_1 ≥X p_2 ≥X ... ≥X p_r then                  ▷ No duplicates
10:                 R ← R ∪ CombinePredecessorIntrees(p_1, ..., p_r)
                end if
            end for
        end for
        return R
15: end procedure

    procedure CombinePredecessorIntrees(p_1, ..., p_r)
        return { p_1  p_2  ···  p_r }     ▷ New root with predecessor intrees p_1, ..., p_r
                  \    |    /
                   \   |   /
                    \  |  /
                     \ | /
                      \|/
                       ●
    end procedure
```

Please note that algorithm **??** can be easily adjusted to generate only non-trivial intrees (i.e. intrees whose root has a degree greater than 1) by adding a simple check that shall occur *only in the initial call* of GenerateIntrees: We then simply have to assure that r is greater than 1 in line **??**.

Moreover, even if the algorithm is described here in a quite mathematical way, it can be faily easy implemented in e.g. Python. Especially, the ordering of intrees – that might look complicated at first sight – can be acchieved with very simplistic methods.

Chapter 6

Benchmarks

While chapter **??** will focus on more theoretical aspects of our problem, we first summarize in short how the program performs in practice. We therefore ran tests that considered intrees with a certain number of tasks.

Remark: It may be the case that the program (and in particular, the data shown in the tables) changed since this thesis was finished.

6.1 Canonical vs. non-canonical variant

First we compare how the program performs with resp. without the elimination of equivalent snapshots (see section **??**). We therefore computed the LEAF schedules for all non-trivial intrees with a certain number of tasks and measured the run times. Table **??** shows the results (we did not measure up to milliseconds but only seconds since the results speak for themselves and need not be more accurate). This table was created using a Intel Core2 with 2.13GHz and 2Gb of RAM. For 13 tasks, we had to stop the variant that did not exclude equivalent snapshots because it needed too much memory.

Tasks	≤ 8	9	10	11	12	13
Equivalent snapshots included	$\leq 0.2s$	$\approx 0.5s$	$3s$	$15s$	$80s$	$> 360s$
Equivalent snapshots excluded	$\leq 0.2s$	$\approx 0.3s$	$2s$	$6s$	$20s$	$75s$

Table 6.1: Run time comparison with vs. without elimination of equivalent snapshots.

Even more important is the memory consumption: While the variant *with* canonical snapshots used roughly 30mb of memory to process all non-trivial intrees with 12 tasks, we needed over 550mb when we did *not* exclude equivalent snapshots.

We observe that excluding equivalent snapshots results both in a remarkable speedup and in a considerably smaller memory footprint. For this reason, we quite quickly decided to focus on the variant that excludes equivalent snapshots. All following benchmarks are done with equivalent snapshots excluded.

6.2 Keeping all intrees in memory

We now research how much time it takes to compute optimal schedules for intrees with a certain size.

As a first benchmark, we computed optimal schedules for all non-trivial intrees (with a certain amount of tasks) "in one run", i.e. we generated all non-trivial intrees, kept them in main memory and computed the optimal schedule for each of these intrees. This technique has the advantage that intermediate results can be re-used and do not need to be recomputed over and over again. On the other hand, keeping that many intrees in RAM leads to enormous memory consumption and, thus, was only done for up to 15 intrees.

Table **??** shows the results. This table was created on a reasonably modern machine (with an Intel Core i7). Peak memory consumption was measured with Valgrind for up to 13 tasks. According to the valgrind manual [**massifmanual**], these measurements are accurate within 1%. For more than 13 tasks, Valgrind was too slow, so we went for a more simplistic approach: We simply observed the stats shown by htop. This means, that the memory consumption for 14 and 15 tasks is not *that* accurate, but probably still accurate enough to get an impression of how many memory was needed.

Moreover, we carried out these tests with simple floating point numbers to represent probabilities and expected run times. We also implemented another feature to support fractions — see section **??** for more details.

Tasks	Intrees	Snapshots	Time	Memory
≤ 9	≤ 171	≤ 891	≤ 0.5s	≤ 1451040b
10	433	3004	1s	4941576b
11	1123	10143	4s	16847304b
12	2924	34065	6s	57016592b
13	7720	113492	26s	193781984b
14	20487	375088	2m10s	$\approx$ 410mb
15	54838	1230391	7m	$\approx$ 1.5Gb

Table 6.2: Time needed to compute optimal schedules for all non-trivial intrees with a certain number of tasks using a floating-point representation for probabilities and run times.

The memory consumption might look quite high at first glance, but if we look closer, it becomes clear that quite a lot of things have to be stored. We have to store each snapshot containing the current intree, the scheduled tasks, and its successors. Moreover, we need a "pool" of snapshots that is used to avoid over and over recomputing equivalent snapshots. Moreover, the statistics in table **??** of course include the memory that is consumed by e.g. auxiliary data structures from the C++ container classes.

The main problem is – of course – that the number of non-trivial intrees drastically grows as the number of tasks increases (1,1,2,2,5,11,28,67,171,433,1123,... — see [**oeisnumbernontrivialintrees**]). For that reason, we continue benchmarking by computing optimal schedules for single intrees (or a small set of intrees).

6.3 Grouping intrees and computing them one after another

The next benchmark we conducted was done for intrees with 15 or more tasks. First, we generated all (nontrivial) intrees with a certain number of tasks. Afterwards, we split the whole collection of intrees into smaller chunks containing a certain amount of intrees. We then processed these chunks one after another.

The results for this setting are shown in table **??**.

In this scenario, the measurements become more difficult to interpret. Especially, the number of snapshots and the amount of memory can not be directly compared to the values shown in section **??**. For the time, it is also non-trivial. This is because we carry out some computations twice. While we can measure the time quite accurately, we can not do this for the memory consumption because Valgrind slowed down the program too much to finish in reasonable time. This is why we used `htop` to estimate the amount of memory used for 15 and 16 tasks. We did not exactly measure the memory consumption for 17 or more tasks, but it can be said that we experienced no problems on a machine with 8Gb RAM.

One interesting fact that can be observed from table **??** is that for 15 tasks, we needed roughly 11 minutes, while in the non-grouped case, it took only about the half of this time (about 6 minutes — see table **??**). That is, as expected, grouping the intrees into smaller chunks considerably increases the run time, but – on the other hand – seems to be the only real alternative since the number of trees grows that fast so they do not fit into main memory anymore.

As you can see, the time needed to compute optimal schedules for all non-trivial intrees increases very fast with growing number of tasks. Of course, the run time can be reduced by parallelizing the

Tasks	Intrees	Chunk size	Time	Memory per chunk
15	54838	5000	11m	$<$ 500mb
16	147570	5000	1h	$<$ 800mb
17	399466	5000	3h	n.a.
18	1086312	6000	13.3h	n.a.
19	2967517	6000	24h	n.a.

Table 6.3: Time needed to compute optimal schedules for all non-trivial intrees with a certain number using sets of subtrees of tasks using floating point numbers for probabilities and run times.

computations for example in a way that computes different chunks in parallel.

6.4 Other representations for probabilities and expected values

As said before, the user can choose between different representations for probailities and expected values. By default, we simply use C++'s `float`s but for some scenarios it might be useful to have precise fractions. We implemented those using Boost Rational Number library or GNU Multiple Precision Arithmetic Library (GMP).

We compare the performance of the program with different representations, namely the default `float` representation, the representation by Boost's Rational Number library with `unsigned long long`s as denominators and numerators, and representationn by GMP. The results are shown in table **??** (generated on an Intel Core2 with 2.13GHz).

Tasks	9	10	11	12	13
`float`	$\leq 0.3s$	$2s$	$6s$	$20s$	$75s$
Boost Raionals	≤ 0.3	$2s$	$6s$	$21s$	$78s$
GMP	$\leq 0.6s$	$3s$	$10s$	$38s$	$140s$

Table 6.4: Comparison of different representations for probabilities and expected values.

We see that Boost's Rational Number library does not considerably increase run time, while GMP almost doubles it. This may come from the fact that GMP uses "arbitrarily large" integers as denominators and numerators to represent fractions, thus allowing arbitrary precision (within memory limits), while we can adjust which underlying representation for numerators and denominators Boost should use. Since we are using common `unsigned long long`s, the operations on these are no very expensive, while the GMP counterparts might be due to overflow checks, dynamic memory allocation for larger numbers, etc. However, *only* GMP offers – out of the box – arbitrary precision, while Boost's Rational Number library might yield wrong results if the denominators and numerators get too big for `unsigned long long`s.

Surprisingly, using Boost's Rationals did not increase memory consumption considerably. On the other hand, GMP also increased the memory consumption by roughly 40%. This again comes possibly from the fact that GMP offers arbitrary precision, thereby requiring to store arbitrarily large numbers and requiring auxiliary data structures.

Finally, we observer that using rational numbers does not increase run time as much as not eliminating equivalent snapshots, but the additional amount of time with GMP is considerably large – as a tradeoff we get arbitrary precision.

Remark: We almost never ran into problems when we used simple `float`s to represent probabilities. Only in rare cases we had to rely on GMP because of rounding errors that possibly gave us incorrect results. For this reason, we decided that the default representation shall use `float`s.

Chapter 7

Suboptimal strategies for P3

In this section, we take a look at some strategies and use them for scheduling with three processors. We will do so very exemplarily and inspect counterexamples for the respective strategies.

Finally, we conclude some properties that *optimal* scheduling strategies seem to fulfill.

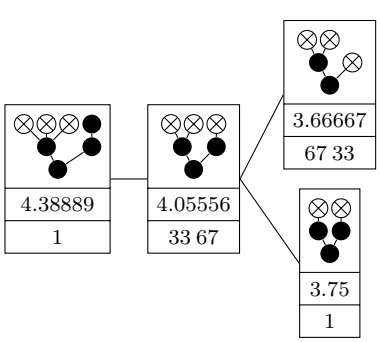
TODO: Introductory text

7.1 HLF

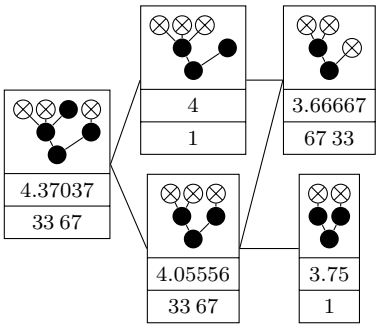
First of all, we take look at some examples that show that HLF is not optimal for the three processor case. We will consider several phenomena that can occur if we use HLF with three processors.

7.1.1 HLF does not behave the same for intrees with same profile

In the two-processor case it is known that trees with the same level profile (see section **??**) have the same run time. This is not the case for three processors TODO: Erklären, warum der Beweis nicht mehr hinhaut! Figure **??** shows an intree, where HLF can choose at some points, and different choices result in different runtimes.



(a) Suboptimal HLF run



(b) Optimal HLF run. This is also the overal optimal schedule.

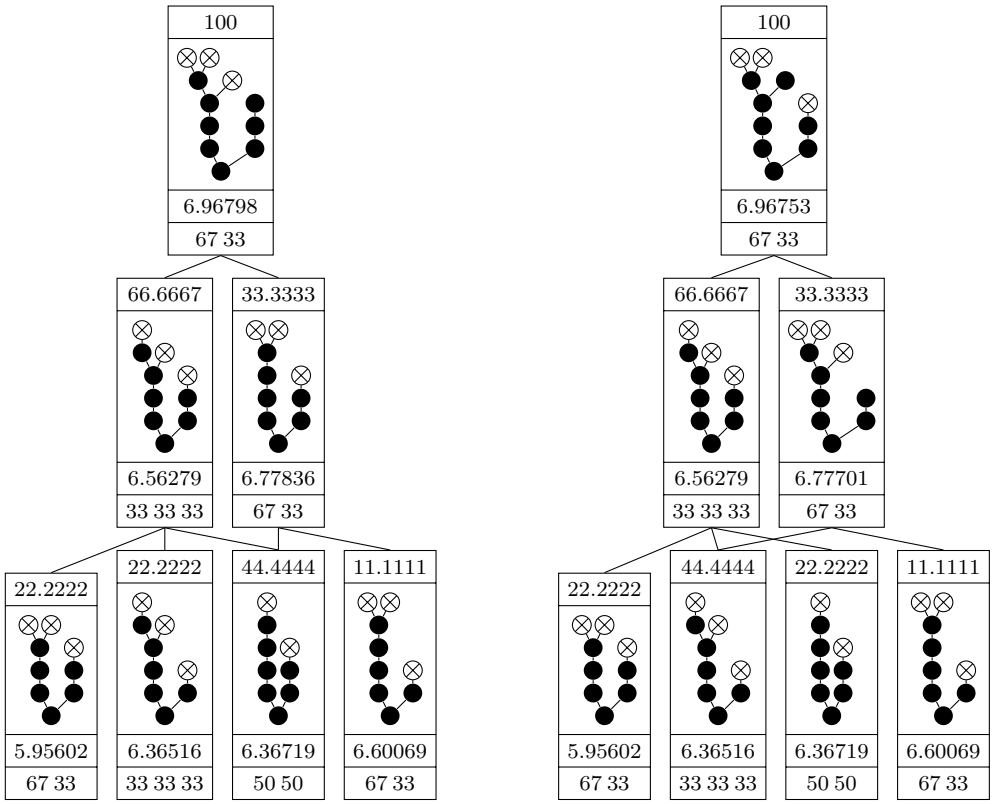
Figure 7.1: HLF on $(0, 0, 1, 1, 1, 2)$. Different runs of HLF do not necessarily produce the same result.

Because HLF can produce different run times depending which task it has chosen, it is clear that HLF in its raw form can not be optimal. The following section reveals even more.

7.1.2 Examples where HLF is strictly suboptimal

The example from figure **??** shows the optimal run. We observe that this run is a specific instance of HLF, because at each point of time, always tasks with the highest level numbers are chosen.

However, there are intrees, where *no* HLF-run is optimal. Figures **??**, **??** and **??** show some examples for which this is exactly the case.



(a) HLF – suboptimal

(b) Optimal run is non-HLF

Figure 7.2: HLF vs. optimal solution for $(0, 0, 1, 2, 3, 4, 6, 6, 8, 8)$

7.1.3 Quality of HLF

We have seen that HLF is suboptimal in some cases, but experience shows that HLF is quite good in many cases. In fact [**journals/siamcomp/PapadimitriouT87**] have shown that the following theorem holds:

Theorem 7.1. *There is a function $\beta : \mathbb{N} \mapsto \mathbb{R}_0^+$ with $\lim_{n\to\infty} \beta(n) = 0$ such that for each intree I and an arbitrary HLF strategy HLF we have*

$$T_{HLF}(I) \leq \inf_{\pi} T_\pi(I) \cdot (1 + \beta(N)),$$

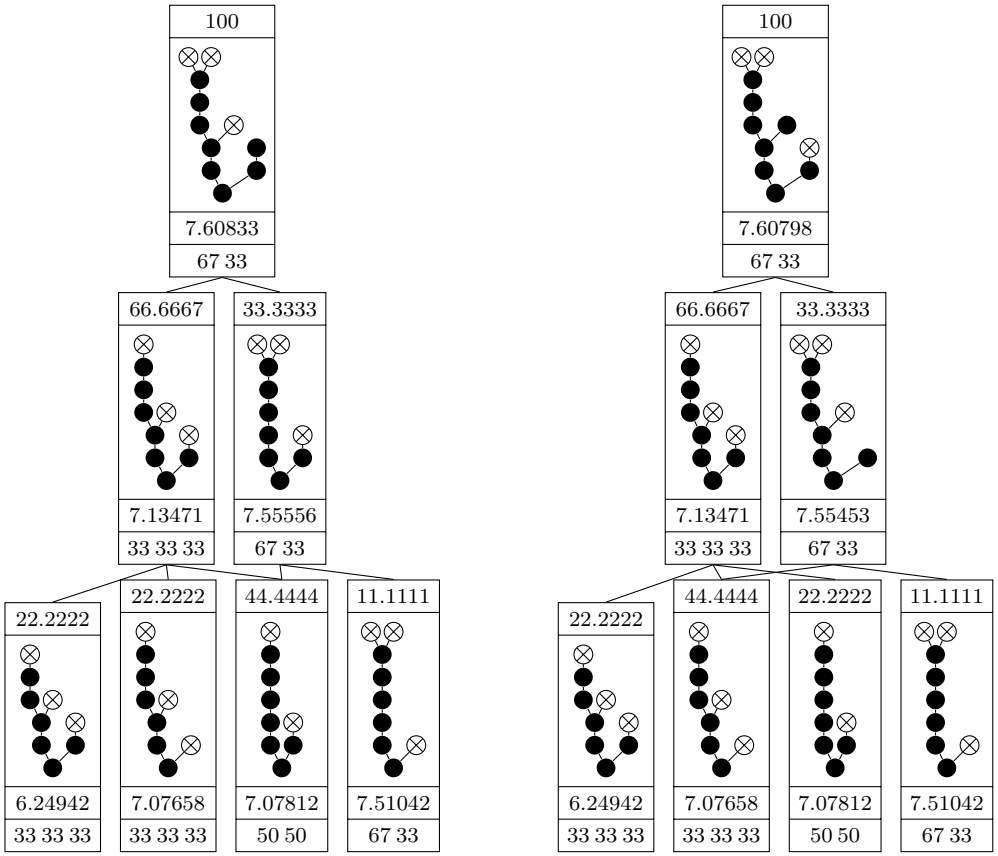
where $T_P(I)$ denotes the expected run time for intree I if it is scheduled by policy P, N is the number of tasks in I and the infimum is taken over all scheduling strategies π.

Proof. See [**journals/siamcomp/PapadimitriouT87**]. □

This result is quite useful because it shows that HLF is – even if not optimal – quite close as the number of tasks grows.

7.1.4 Different categories of sub-optimality

As we saw, there are two possibilities for the sub-optimality of HLF:



(a) HLF – suboptimal

(b) Optimal run is non-HLF

Figure 7.3: HLF vs. optimal solution for $(0, 0, 1, 2, 4, 4, 6, 7, 8, 8)$ (taken from Ernst Mayr)

- *Not each possible run* of HLF yields the same run time.
- The optimal run has to choose a strict non-HLF task.

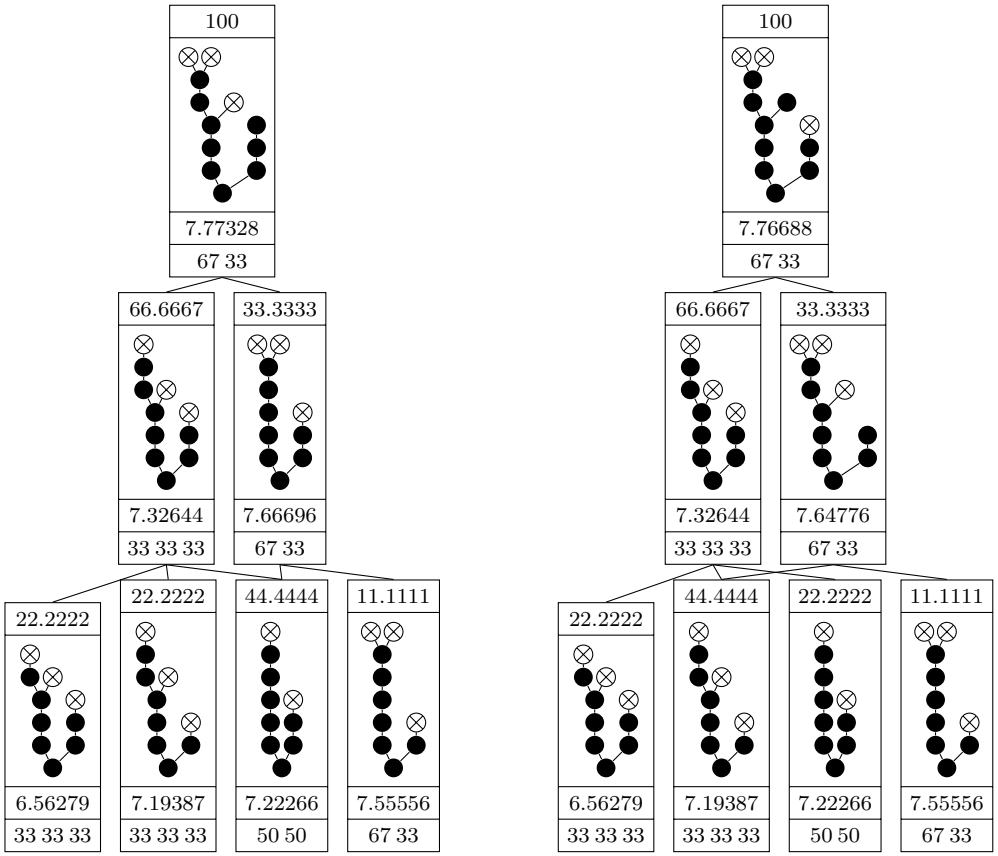
We use the following nomenclature: A strategy is called *can-optimal* (a term already used in [**MoritzMaasDiploma**]), if it *might* result in an optimal schedule. A strategy that *can not* produce an optimal solution is called *strictly suboptimal* (or simply suboptimmal if it is clear from the context).

It is a notable fact that there are many cases, where HLF is can-optimal.

We use this distinction to (roughly) differentiate the following strategies into two categories:

- The cases where HLF is strictly suboptimal lead us to several strategies that are presented in section **??**. These strategies are strict counterparts to HLF.
- For the cases where HLF is can-optimal, we examined several strategies that try to determine which tasks should be chosen to minimize the expected run time. These strategies are presented in section **??**. These strategies can be seen as "refinements" of HLF such that HLF behaves better.

We will not only focus on particular strategies, but we will focus on which snapshots can be excluded or which particular structure snapshots might have. That is, the strategies we consider are in many cases ambiguous because they admit several possible choices. However, they do *not* allow *all* possible choices, thereby possibly reducing the amount of snapshots to examine. Not all strategies fall strictly into one of the groups – we then put it into the category we found more insightful.



(a) HLF – suboptimal

(b) Optimal run is non-HLF

Figure 7.4: HLF vs. optimal solution for $(0, 0, 1, 2, 3, 4, 5, 5, 7, 9, 9)$ (taken from Chandy/Reynolds)

7.2 (Dynamic) list scheduling

List scheduling considers the current intree and generates a list of tasks sorted in such a way that the tasks that come first in the list shall be priorized and scheduled first, if possible. As shown by [**MoritzMaasDiploma**], dynamic list scheduling strategies can not be optimal for non-preemtive scheduling of intrees (whose tasks' run times are exponentially distributed) with three processors.

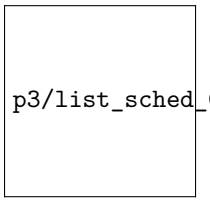
This can be seen by examining the intree $(0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 6)$ as shown in figure **??**. Because the subtree shown in figure **??** is optimally scheduled using other tasks than the one that have to be used when it is reached in the original schedule, there can not exist a dynamic list scheduling strategy that works for *all* intrees.

Note that this also excludes HLF from being optimal.

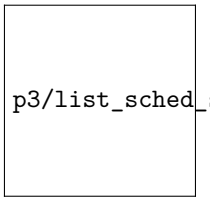
That is, an optimal strategy has to consider which tasks are already scheduled from previous steps and can not rely *only* on the current intree's structure. We experience this phenomenon in many intrees shown in this chapter. However, for the initial snapshot, we – of course – must rely on the structure of the intree. Thus, we will examine some strategies and inspect if the first snapshot of an optimal schedule adheres specific rules.

7.3 Non-HLF strategies

We saw in section **??** that there are examples where HLF is strictly suboptimal. We now present some strategies that we took into consideration and that we examined w.r.t. their optimality.



(a) Optimal schedule for intree $(0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 6)$.



(b) This subtree of $(0, 0, 1, 1, 1, 2, 2, 2, 3, 6)$ is optimally scheduled as shown, but in the optimal schedule for the original intree, it has to be scheduled in another way.

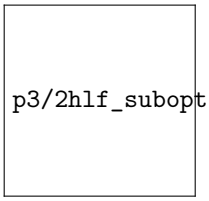
Figure 7.5: A counterexample for list scheduling

We therefore considered optimal schedules and examined whether the initial snapshot (i.e. the initial choice of tasks) are formed upon a certain pattern.

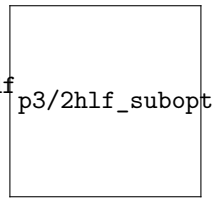
These strategies are strongly different from HLF and rely upon the structure of the intree. None of the strategies considered lead us to strictly optimal results in all cases.

7.3.1 "2-HLF plus 1"

We examined all intrees with up to 13 tasks, especially the cases where HLF is not optimal. Thereby, we obsered that in all cases where three tasks could be scheduled, the optimal solution scheduled two tasks, that could be chosen by HLF for two processors and only the third task *might* be a task that would not have been chosen by HLF (see figures **??**, **??** and **??** as particular instances of those). Thus, we examined whether an optimal scheduling strategy for three processors has always *at most one* task that is non-HLF. Interestingly, there is an intree with 14 tasks, whose optimal schedule starts out by choosing the single topmost task and *two* non-HLF tasks. This intree $((0, 0, 1, 2, 2, 3, 3, 6, 8, 9, 10, 11, 12))$ is shown in figure **??**. We can generalize this intree to a whole family of intrees where the optimal strategy initially chooses the single topmost task, and two lowest-level leaves by adding leaves along the longest chain. This results in intrees of the form $(0, 0, 1, 2, 3, 4, 4, 5, 5, 8, 10, 11, 12, 13, 14, 15, \dots)$.



(a) Optimal schedule picking *two* non-HLF tasks.

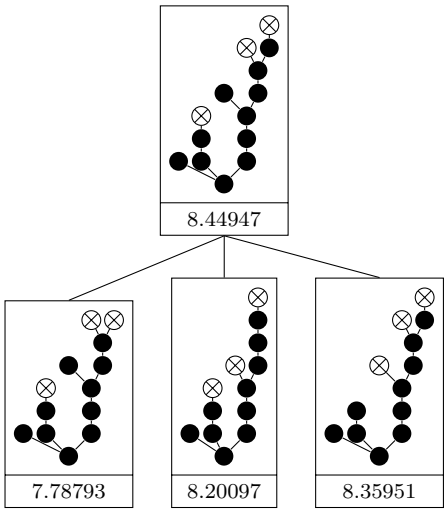


(b) HLF-Schedule

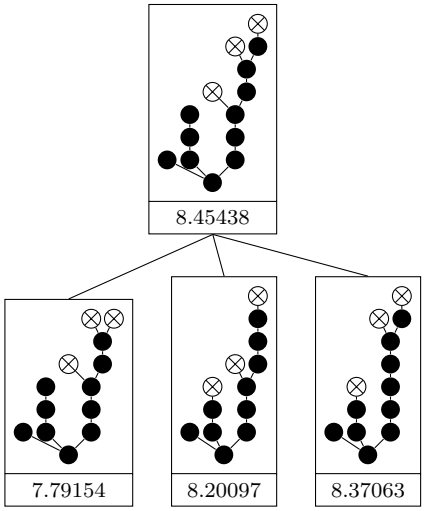
Figure 7.6: Intree $(0, 0, 1, 2, 2, 3, 3, 6, 8, 9, 10, 11, 12)$ requires the optimal schedule to start out by choosing two non-HLF tasks.

7.3.2 Only highest or lowest leaves

The trees we examined so far resulted in schedules that picked only combinations *highest leaves and lowest leaves* possible. Thus, we were tempted to think that an optimal schedule chooses only topmost tasks or leaves whose level is minimal (among all leaves). However, this is not a criterion for an optimal schedule, as we can observe by scheduling the 14-tasks-intree $(0, 0, 0, 2, 3, 4, 5, 7, 7, 9, 10, 10, 12)$, which is shown in figure **??**. For this intree, we have to schedule two topmost tasks and one task on level 3, but there is one unscheduled task remaining on level 1.



(a) Optimal schedule picking a non-HLF task that is also not the lowest possible.



(b) Suboptimal HLF-Schedule.

Figure 7.7: Intree $(0, 0, 0, 2, 3, 4, 5, 7, 7, 9, 10, 10, 12)$ shows that there are intrees where an optimal schedule has to choose a non-HLF task that has a higher level than some non-chosen task.

7.4 Refining can-optimal HLF

As mentioned in section **??**, there are many cases where HLF is *can-optimal*, i.e. where the optimal schedule always has tasks of the highest levels scheduled, but not each HLF schedule is optimal. This results from situations where HLF can choose one from several task as the next task to be scheduled. We describe some strategies that try to eliminate these ambiguities and give counterexamples that show that these strategies are not optimal.

7.4.1 "As few free paths as possible"

(For now,) we call a path from the root to a leaf (i.e. a ready task) t *free* if t is not scheduled.

One might be tempted to think that it should be the foremost goal to exploit parallelism as good as possible and that this might be acchieved by choosing the currently scheduled tasks in a manner such that as few free paths as possible in an optimal schedule. That is, we choose the leaves in a way so that the ends of as many different paths as possible are scheduled. This strategy was the first that came to our mind and was inspired by looking at the counterexamples against HLF depicted in figures **??**, **??**, **??** and **??**. We observe for these intrees that the optimal schedules has no as few free chains as possible.

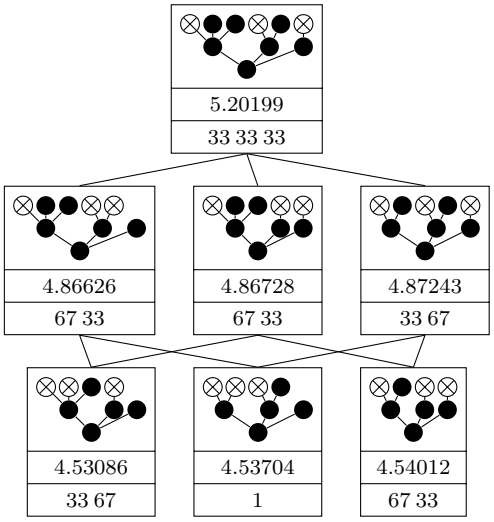
However, there are examples where the optimal contains snapshots that do not adhere this property. Consider e.g. the tree $(0, 0, 0, 1, 1, 1, 2, 2, 3)$ (figure **??** compares the optimal schedule to the no-free-paths schedule).

7.4.2 Subtree with minimum number of topmost tasks

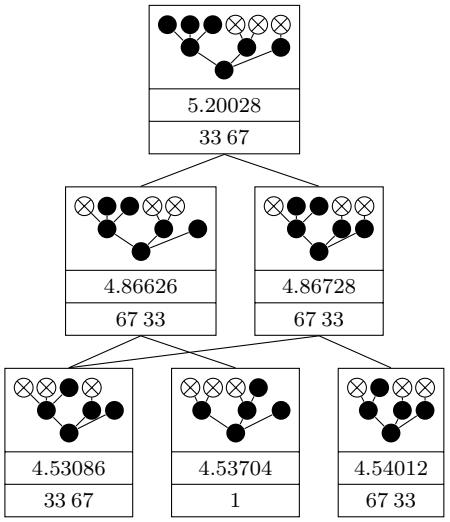
If we consider the intree $(0, 0, 0, 1, 1, 1, 2, 2, 3)$, we have seen that an optimal schedule picks 7,8 and 9 as initially scheduled tasks (see figure **??**). Moreover, in many cases, topmost-tasks that are the *single direct predecessor* of their respective direct successor, are chosen by an optimal schedule.

These facts lead us to the suspicion that – if we have an intree for which HLF is can-optimal – (informally) we should pick the subtrees with the lowest number of topmost tasks.

In this context, it is important to exactly describe which subtree we are talking about. Therefore, we employ the following definition:



(a) HLF schedule while choosing tasks such that there are as few free paths as possible – overall run time of 5.20199.



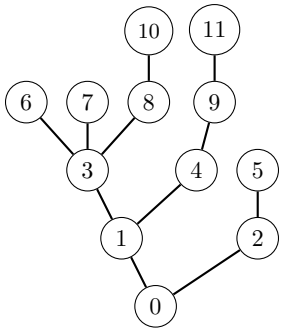
(b) Optimal schedule (run time 5.20028) has three free paths at the beginning.

Figure 7.8: HLF with as few free paths as possible is not necessarily optimal.

Definition 7.2 (Toptask-maximal subtree for a leaf)**.** *Let t be a leaf of an intree I and let $p = (t, t_1, t_2, t_3, \ldots, r)$ be the path from t to the root r.*

The toptask-maximal subtree for a leaf *t is the subtree rooted at the* lowest *task t^* within p that is not t and that does* not *contain more topmost tasks than the subtree rooted at the predecessor of t^* within p.*

As an example, consider the following intree (remember that topmost tasks are defined to be the tasks whose levels are at least as large as the level of *any* task in the intree — see section **??**):



The maximal subtree for leaf 10 is the subtree rooted at node 3, which can be derived as follows:

- The path from 10 to the root 0 is given by $p = (10, 8, 3, 1, 0)$.
- We consider the subtrees rooted at the tasks along this path, and denote the subtree rooted at node x by I_x:
  - The subtree rooted at 10 (called I_{10}) contains only the topmost task 10.
  - Subtree I_8 contains only topmost task 10.
  - Subtree I_3 still contains only 10 as the topmost task (it introduces only a new leaf, namely 7).
  - Subtree I_1 contains 10 *and* 11 as topmost tasks.

- As seen, task 3 is the lowest task within p that does not contain more topmost tasks than its predecessor (in the path from the leaf to the root).

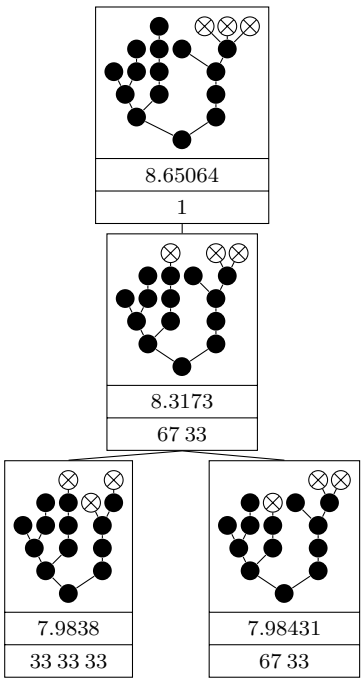
A strategy for cases where HLF is can-optimal now might be to resolve HLF-ambiguities as follows:

- Generate all possible choices that could result from HLF.
- For each topmost task, compute the topmost-maximal subtree.
- Prefer topmost tasks whose topmost-maximal subtrees contain fewer topmost tasks.

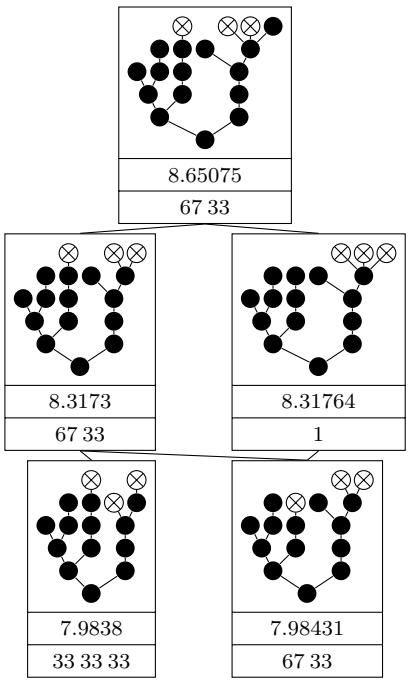
The last step in the above explanation can be viewed as follows: We first create all possible choices of topmost tasks and then only pursue those, where there is no topmost-maximal subtree that has unscheduled topmost tasks.

This strategy seems to do a good job in many cases, but can be seen to be false by examining the optimal schedule for the following intree with 18 tasks: $(0, 0, 1, 1, 2, 3, 3, 4, 5, 7, 8, 9, 9, 11, 13, 13, 13)$. It is shown in figure **??**.

Remark: We did not specify what should be done if there are several maximal subtrees with the same number of topmost tasks, but our counterexample suffices that this strategy does not work optimally even if there are no maximal subtrees with the same number of nodes.



(a) Optimal schedule picking a subtree with three topmost tasks.



(b) If we initially start with tasks as shown, this is the best schedule that can be obtained.

Figure 7.9: Intree $(0, 0, 1, 1, 2, 3, 3, 4, 5, 7, 8, 9, 9, 11, 13, 13, 13)$: For this intree, the optimal schedule chooses all tasks from a subtree with three topmost tasks and chooses none of the subtree with only one topmost tasks.

7.4.3 Subtree with maximum or minimum number of leaves

It can be quickly seen that slightly altering the strategy given in section **??** in the sense that we do not concentrate on the number of *topmost tasks* in a maximal subtree, but more generraly on the number of *leaves* in a maximal subtree, does not yield a successful strategy. We adapt the notion of topmost-maximal subtrees in a straightforward manner:

Definition 7.3 (Leaf-maximal subtree for a leaf)**.** *Let t be a leaf of an intree I and let $p = (t, t_1, t_2, t_3, \ldots, r)$ be the path from t to the root r.*

The leaf-maximal subtree for a leaf *t is the subtree rooted at the* lowest *task t^* within p that is not t and that does* not *contain more leaves than the predecessor of t^* within p.*

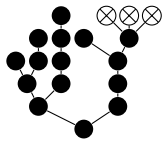
Preferring leaf-maximal subtrees with fewer leaves Figure **??** shows that this strategy is not optimal, since the optimal solution prefers a subtree with four leaves over one with only three leaves. *Remark:* The tree $(0, 0, 1, 1, 2, 3, 3, 4, 5, 7, 8, 9, 9, 11, 13, 13, 13)$ in particular shows that there are situations where a topmost task that is the *only* requirement for its predecessor is initially *not* scheduled in the optimal case. During our research we have experienced that this is a very rare situation.

Preferring leaf-maximal subtrees with more leaves One of our first examples, the intree $(0, 0, 0, 1, 1, 1, 2, 2, 3)$ (see figure **??**) already shows that this strategy is not optimal in general.

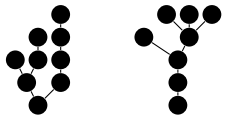
7.4.4 Preferring root's predecessors with longest processing time

We also tried a recursive approach that decomposed an intree as follows: We separate the intrees rooted at the predecessors of the root. This way, we get a whole set of intrees. For each subtree, we now compute the optimal schedule and, moreover, the expected processing time – assuming three processors in each individual subtree. The schedule for the whole intree then shall prefer subtrees whose expected processing time is the longest.

We were tempted to conjecture this because of the intree $(0, 0, 1, 1, 2, 3, 3, 4, 5, 7, 8, 9, 9, 11, 13, 13, 13)$ whose optimal schedule starts shown in figure **??**. If we decompose this intree into its parts, we see that the subtree whose expected run time is maximal is the one whose tasks are initially scheduled in the optimal schedule (see figure **??**).



(a) Intree with initially scheduled tasks (for optimal schedule).



(b) Removing the root yields two subtrees with optimal expected runtimes (for three processors) as noted.

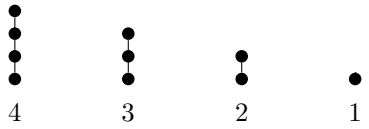
Figure 7.10: Intree $(0, 0, 1, 1, 2, 3, 3, 4, 5, 7, 8, 9, 9, 11, 13, 13, 13)$ and its corresponding subtrees rooted at the root's predecessors

Once again, this strategy can be shown to be suboptimal by considering $(0, 0, 0, 1, 1, 1, 2, 2, 3)$ (as depicted in figure **??**) whose root has the following three predecessor intrees.

For $(0, 0, 0, 1, 1, 1, 2, 2, 3)$ the optimal schedule initially chooses the tasks 7,8 and 9 (the respective subtrees have run times 2.5 and 2).

7.4.5 Preferring root's predecessors with shortest processing time

It is clear that the opposite of the strategy from section **??**, namely preferring those subtrees whose processing time is shortest, does also not yield correct results. This can be easily seen by considering the intree $(0, 1, 2, 3, 0, 5, 6, 0, 8, 0)$ that is optimally scheduled by HLF (see section **??** for a proof) and whose root has the three following predecessors:



7.4.6 "Filling up subtrees"

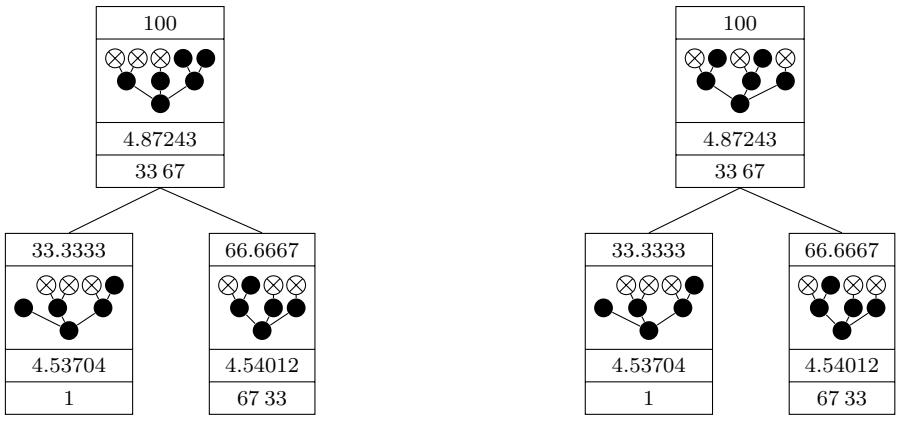
We observed that (for three processors) an optimal schedule looks as if it chose certain subtrees and "filled them up one after another". This is probably most concise formalized as the following pattern:

- Identify disjoint leaf-maximal (or topmost-maximal) subtrees of the whole intree.
- Assign priorities to the subtrees so they are sorted according to this priority (thus, we have a sequence of subtrees $S_1, \ldots, S_r$).
- Schedule as many tasks from S_1 as possible. If all ready tasks in S_1 are scheduled, schedule as many tasks as possible in S_2. If all tasks in S_2 are scheduled, schedule as many tasks as possible in S_3.

An alternative formulation of the above strategy states that there is at most one task having both scheduled and non-scheduled predecessors that are leaves.

We have already seen that there are optimal schedules violating this property, e.g. the intree $(0, 0, 0, 1, 1, 2, 2, 3)$. But for those intrees, there is another schedule having *exact the same* run time but fulfilling the property.

The example intree $(0, 0, 0, 1, 1, 2, 2, 3)$ admits two different schedules (one beginning with tasks 4,5,8 and the other beginning with 4,6,8) that have exactly the same run time (see figure **??**). The reason is that both schedules result in equivalent snapshots with the same probabilities after the first task finishes.



(a) Optimal schedule starting with tasks 4,5 and 8. (b) Optimal schedule starting with tasks 4,6 and 8.

Figure 7.11: The intree $(0, 0, 0, 1, 1, 2, 2, 3)$ has two different schedules reaching the optimum expected run time.

For trees with fewer than 12 tasks, we could not find any tree that violated our conjecture, but the intree $(0, 0, 0, 2, 2, 3, 5, 5, 6, 6, 6)$ has the interesting property that the optimal schedule for this intree has to schedule tasks such that the intree contains two tasks that have as well scheduled as non-scheduled but ready predecessors. Figure **??** shows the first steps of an optimal schedule for this intree.

Remark: Surprisingly, every subtree of $(0, 0, 0, 2, 2, 3, 5, 5, 6, 6, 6)$ fulfills the conjecture that there is at most one task that has both scheduled and non-scheduled leaves as predecessors (during the whole schedule). This shows that even task that *might seem* of minor impact for the first choices, might be relevant to the question which tasks are chosen in the optimal schedule.

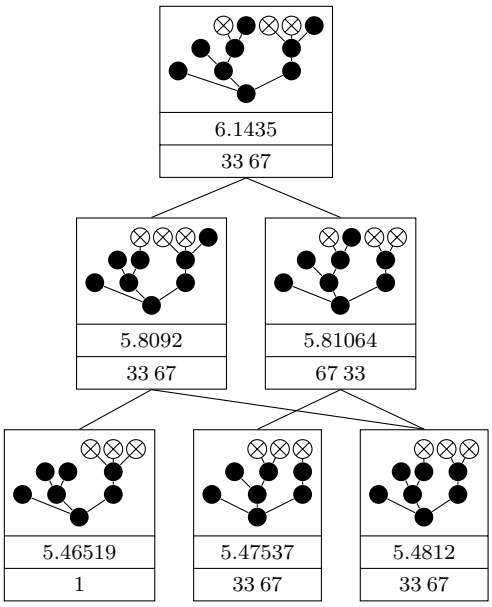


Figure 7.12: The optimal schedule for $(0, 0, 0, 2, 2, 3, 5, 5, 6, 6, 6)$ has two tasks (5 and 6) that have both scheduled and non-scheduled leaves as predecessors.

7.5 Maximizing 3-processor-time, minimizing 1-processor time

Up to now, we maily focused on the structure of the current intree to derive strategies — which all turned out to be (not strictly, but still) suboptimal. We now inspect another, more involved approach.

If we have three processors in total, we can split the total run time into three parts: The time where all three processors are processing tasks, the time where one processor is idle and two are working, and the time where only one processor is working.

Definition 7.4 (Run time and its variants)**.** *We denote by T the expected run time for a schedule associated with an intree. Moreover, we denote the time where exactly p taks are scheduled by T_p.*

Note that T actually describes an *expected value*. Because of the linearity of expectation, we have that – for three processors – $T = T_1 + T_2 + T_3$. If we want to construct an optimal schedule for three processors, we might be tempted to think that (at least) one of the two following criteria should be fulfilled for the optimal schedule:

P3L For the optimal schedule, T_3 should be maximal (over all schedules), i.e. we should exploit three processors as long as possible (in the expectation).

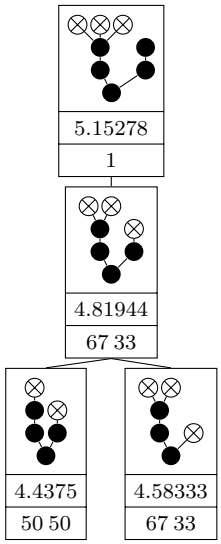
P1S For the optimal schedule, T_1 should be minimal (over all schedules), i.e. we should try to keep the expected time for which only one processor is working as short as possible.

Surprisingly, *both* of them are wrong (at least if considered separately).

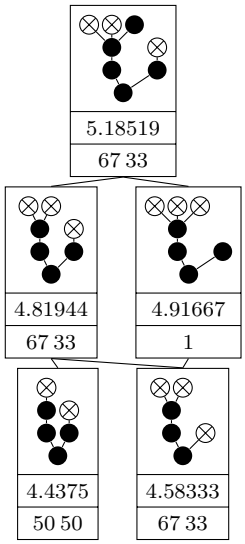
7.5.1 Maximizing T_3

Figure **??** shows an example, where the optimal schedule keeps three processors busy for expected 0.77777 time steps, while a suboptimal schedule keeps three processors busy for a longer expected time, namely about 0.851852 time steps.

From this we can conclude that it may be advantageous in some cases to accept a shorter time with three busy processors, thereby possibly also decreasing the time where only one processor is busy.



(a) Optimal schedule. Keeps three processors busy for $7/9 \approx 0.78$ time steps $((T_3, T_2, T_1) = (7/9, 31/24, 37/12))$.



(b) This suboptimal schedule keeps three processors busy for expectedly 0.851852 time steps $((T_3, T_2, T_1) = (23/27, 10/9, 29/9))$.

Figure 7.13: An intree that shows that an optimal P3 schedule needs not keep busy three processors as long as possible. Snapshots with fewer than 6 tasks omitted since they have at most two tasks to be schedlued can be (optimally) processed via ordinary HLF.

7.5.2 Minimizing T_1

The "other direction", i.e. minimizing the time where only one processor is busy, still is suboptimal. Figure **??** shows an intree with the property that the optimal schedule has an expected timespan of roughly 2.59259, within which only one processor is busy. On the other hand, a suboptimal schedule has a timespan of roughly 2.55555 within which only one processor is busy.

This shows that it can be useful to accept a longer time with only one processor busy, probably acchieving a longer time span where three processors are busy.

7.5.3 Maximizing T_3 *or* minimizing T_1

It can also be shown that even combining the two arguments – in the sense that P3L *or* P1S should be fulfilled for the optimal schedule – is not correct. This can be observed by examining the intree $(0, 0, 1, 1, 2, 3, 3, 3)$. Figure **??** shows this example.

Corollary 7.5. *Let T^s denote the overall run time of a schedule s and T_1^s, T_2^s and T_3^s be the times where exactly three, two and one tasks are scheduled within this schedule, respectively.*

Let I be an intree and S be the set of all schedules. Let s^* be the optimal schedule, which has associated the optimal run time T^*, with T_1^*, T_2^*, T_3^* being its parts.

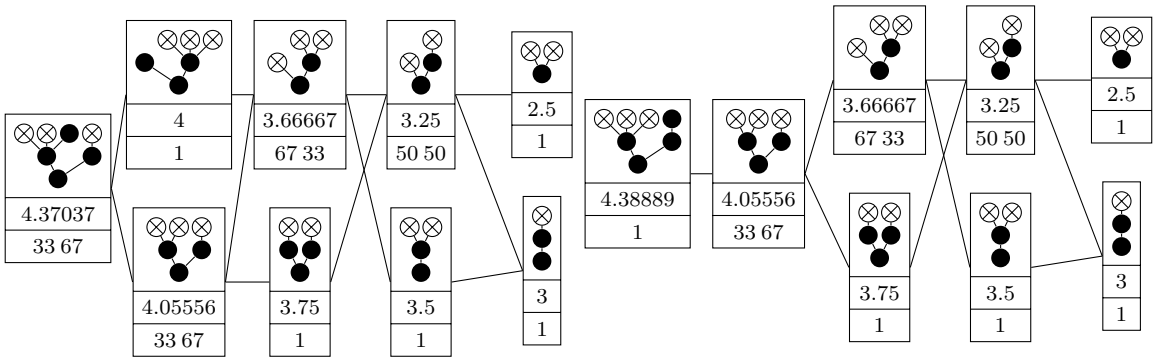
- *It may be the case that there is a schedule $s \in S$ such that $T_3^s \geq T_3^*$.*
- *It may be the case that there is a schedule $s \in S$ such that $T_1^s \leq T_1^*$.*

That is, it is not necessarily the case that T_3 is maximal for the optimal schedule, nor is it necessarily the case that T_1 is minimal for the optimal schedule.

However, after some investigation, we are tempted to conjecture the following.

Conjecture 7.6. *Let I, T^s, T_1^s, T_2^s, T_3^s and S be as defined above. Let s^* be the optimal schedule for I associated with the respective times T_1^*, T_2^*, T_3^*. Then, there is no schedule $s \in S$ such that*

$$T^s > T^* \wedge T_1^s \leq T_1^* \wedge T_3^s \geq T_3^*.$$



(a) Optimal schedule. For expectedly $70/27 \approx 2.59$ time steps, only one processor is busy $(T_3, T_2, T_1) = (23/27, 25/27, 70/27)$.

(b) This suboptimal schedule has an approximated timespan of $23/9 \approx 2.55$ time steps, where only one processor is working $((T_3, T_2, T_1) = (7/9, 19/18, 23/9))$.

Figure 7.14: An intree where the expected time with only one processor being busy is longer within the optimal schedule (≈ 2.59259) than within a suboptimal schedule (≈ 2.555555).

Even if this conjecture turns out to be true, it seems complex to transform this knowledge into a scheduling strategy that does something more significantly efficient than "explore everything, and choose the best", because T_3, T_2 and T_1 are not that easy to compute.

7.6 Conclusion

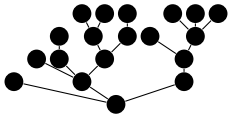
Unfortunately, we did not find any strategy that always yields an optimal schedule. Of course, it is still possible to compute the optimal schedule by an exhaustive search.

During our research, we recognized some patterns that we are tempted to transform into conjectures. We were, however, not yet able to prove or disprove them. These conjectures might be used to reduce the number of snapshots that need to be examined by an exhaustive search used to compute the optimal snapshot.

This section shows the most important conjectures we found.

Conjecture 7.7. *An optimal schedule always schedules as many topmost tasks as possible.*

Please note that the above conjecture does not state anything about *which* topmost tasks should be chosen in order to generate a schedule that is as good as possible. It can – however – drastically reduce the number of choices for the tasks to be scheduled. As an example, consider the following tree:

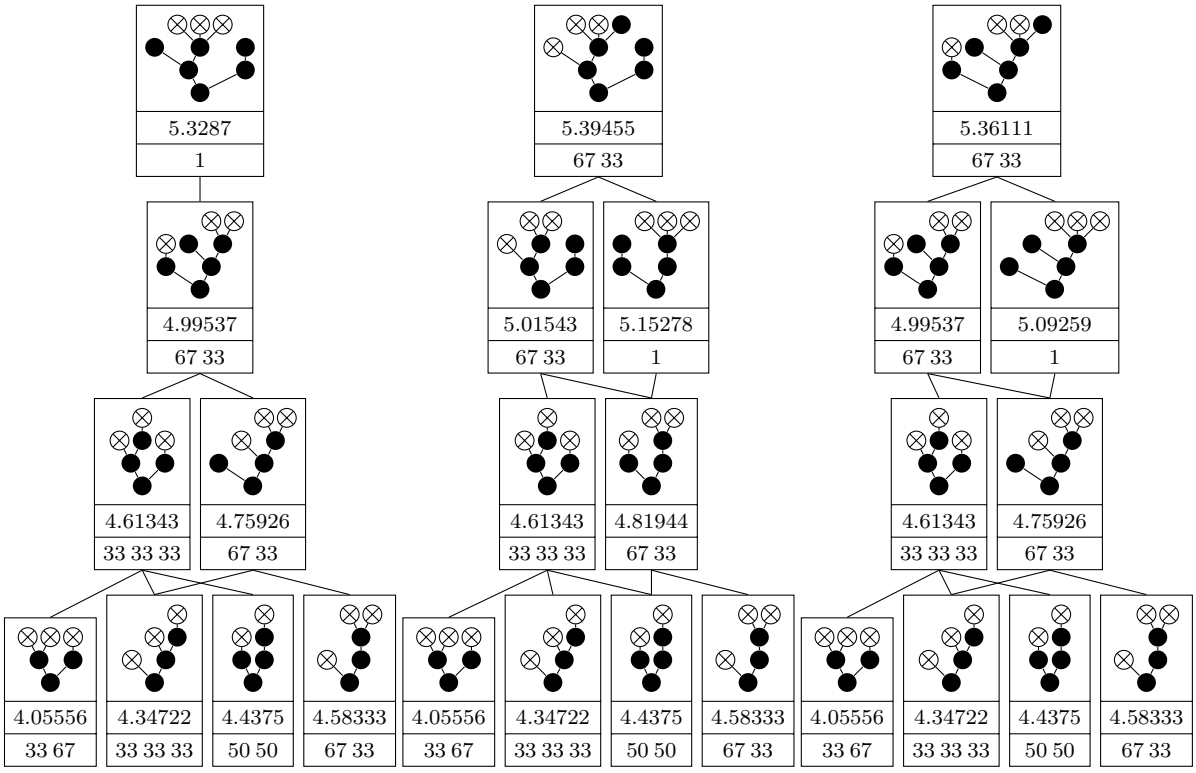


This tree has 6 topmost tasks, but 10 leaves in total. If conjecture **??** is correct, then we can restrict ourselves to combinations of 6 topmost tasks – being at most $\binom{6}{3} = 20$ possible choices, in this particular case even only 6 due to equivalence of snapshots. In contrast, considering all 10 leaves, we have 48 possible choices in the above example.

Note that conjecture **??** also helps us to restrict the number of snapshots in another way: If we have to keep two tasks scheduled (because they were already scheduled in the previous step), we possibly do not need to examine all other tasks to be scheduled. If there are topmost tasks remaining, we can focus on them and do not need to examine non-topmost tasks.

Moreover, we are tempted to say the following:

Conjecture 7.8. *If for an intree only non-top tasks are scheduled, you can schedule any top-task instead of one non-top scheduled task to obtain a better run time.*



(a) Optimal schedule. $(T_3^*, T_2^*, T_1^*) = (\frac{4}{3}, \frac{217}{216}, \frac{323}{108})$.

(b) Schedule with minimal T_1. $(T_3, T_2, T_1) = (\frac{290}{243}, \frac{2369}{1944}, \frac{2899}{972})$.

(c) Schedule with maximal T_3. $(T_3, T_2, T_1) = (\frac{110}{81}, \frac{299}{324}, \frac{499}{162})$.

Figure 7.15: A combination of P3L and P1S is not a criterion for an optimal schedule. The optimal schedule has $T_3^* = \frac{4}{3} \approx 1.333$ and $T_1^* = \frac{323}{108} \approx 2.99074$. One other (suboptimal) schedule has $T_1 = \frac{2899}{972} \approx 2.98251 < T_1^*$, while still another schedule has $T_3 = \frac{110}{81} \approx 1.358025 > T_3^*$.

Conjecture **??** is not that useful in a direct application to reduce the number of snapshots needed to examine if we do an exhaustive search. However, it might be useful for proofs.

The main problems we faced when we tried to prove the above conjectures can be summarized as follows:

- If working with particular cases of intrees, the structure is not necessarily maintained over the induction step — and if so, many case distinctions may be required.
- Comparing different intrees seems to be quite cumbersome, especially if we do not know which tasks are scheduled.

Chapter 8

Properties of schedule DAGs and optimal schedules

We now researching some properties of snapshot DAGs and optimal schedules. In particular, we will look at a particular non-trivial class of intrees, for which HLF is optimal.

8.1 Properties of optimal schedules

8.1.1 Idle processors

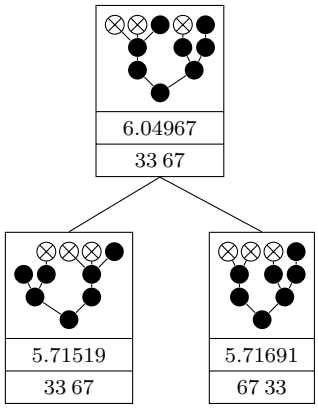
As shown in [**chandyreynoldslargepaper1979**], it is known that an optimal scheduling strategy does not keep a processor idle if it could do some work. An intuitive explanation of this fact is as follows: Assume that there is a strategy that keeps a processor idle at some point even if there is a ready task t that could be processed. Then, we construct a new strategy, that schedules t (using the idle processor) and behaves like the original strategy. Then, it can be shown that this new strategy yields a smaller overall expected run time.

8.1.2 Preemtive vs. non-preemtive scheduling

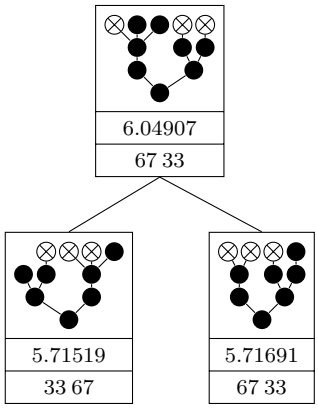
Nonpreemtive scheduling is worse than preemtive scheduling In section **??**, we mentioned that preemptive scheduling might yield better results than non-preemtive scheduling. As an example, [**MoritzMaasDiploma**] shows the intree $(0, 0, 1, 2, 2, 3, 3, 3, 4, 5)$. This intree is – non-preemtively – optimally processed by starting out with tasks 7, 8 and 9. However, we can acchieve a better overall runtime by initially scheduling 8, 9 and 10. Consider figure **??** to compare both schedules.

Another example that shows nicely that preemtive scheduling yields better results in some cases is the intree $(0, 0, 1, 2, 2, 3, 3, 6, 8, 9, 10, 11, 12, 13)$ shown in figure **??**. It is clear that – if the first 6 steps, always the topmost task is the first task to finish, we at some point reach the intree shown in figure **??**. If we allowed to reschedule (by possibly interrupting the execution of some tasks) and chose a schedule as shown in figure **??**, we would obviously acchieve a better overal run time.

Rescheduling only at certain points in time We have seen that preemtive scheduling might result in lower expected run times than non-preemtive scheduling. However, it has been proven in [**chandyreynoldslargepaper1979**] that – without loss of generality – that rescheduling is only necessary – if at all – when one task finishes. I.e. as long as no task on any processor has finished, we can leave them working until the first task finishes.



(a) Optimal non-preemtive schedule.



(b) A better preemtive schedule.

Figure 8.1: Preemtive vs. non-preemtive scheduling for $(0, 0, 1, 1, 1, 2, 2, 2)$. Note that the snapshots in the second line are the same, but they are reached with different probabilities due to the fact that we can reschedule in the preemtive case.

8.1.3 Considerations about subtrees

It can be easily seen by examining any of the intrees for which HLF is strictly suboptimal (described in chapter **??**) that the following theorem must hold:

Theorem 8.1. *There are intrees T and S such that T is a subtree of S and both T and S have the same root such that the optimal (non-preemtive) schedule for S is strictly non-HLF, but the optimal schedule for T is HLF.*

Proof. See considerations and examples in chapter **??**. □

The other way around can be examined by looking at the intree $(0, 0, 1, 2, 2, 3, 3, 6, 8, 9, 10, 11, 12, 13)$ that we have considered before. From another point of view, the intree $(0, 0, 1, 2, 2, 3, 3, 6, 8, 9, 10, 11, 12, 13)$ shows that an optimal schedule may be forced at some time to process a subtree in a way that it would not process it if the subtree was processed for itself. This means that we can formulate the following:

Theorem 8.2. *There are intrees T and S such that T is a subtree of S and both T and S have the same root such that the optimal (non-preemtive) schedule for S is HLF, but the optimal (non-preemtive) schedule for T is non-HLF.*

Proof. See figure **??** and the corresponding considerations above. □

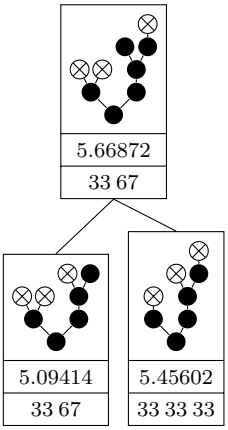
That is, the fact that we consider non-preemtive scheduling only results in the fact that we can not conclude anything about the (non-)optimality of HLF within subtrees even if we know the optimal schedule for a whole intree.

8.1.4 More processors do not worsen the run time

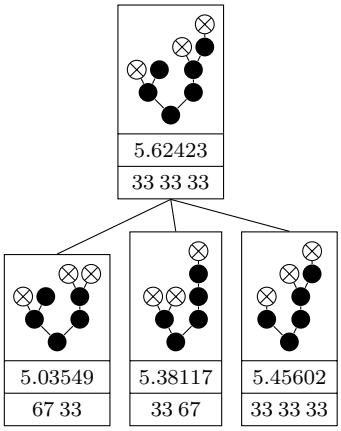
It is intuitively clear that more processors should be able to process an intree faster (if appropriately scheduled). We show a short proof why this is the case.

Theorem 8.3. *Let I be an intree that is processed by p processors according to a specific scheduling strategy. Then we can construct a scheduling strategy for $p + 1$ processors whose expected run time is at least as good as the run time for p processors.*

Proof. We consider a schedule S_p that works for p processors and construct a new schedule S_{p+1} working for $p + 1$ processors. We show that the expected run time for S_{p+1} is not longer than the expected run time for S_p:



(a) Suboptimal schedule. These snapshots are part of the optimal schedule for the intree $(0, 0, 0, 2, 3, 4, 5, 7, 7, 9, 10, 10, 12)$, shown in figure **??**.



(b) Optimal schedule for $(0, 0, 1, 1, 2, 5, 5, 6)$. This schedule is HLF.

Figure 8.2: Intree $(0, 0, 1, 1, 2, 5, 5, 6)$. This intree might result at some point from the intree $(0, 0, 1, 2, 2, 3, 3, 6, 8, 9, 10, 11, 12, 13)$ with the two lowest tasks scheduled (see figure **??**). If we allowed for preemting tasks we could improve the overal run time.

If the intree I has at the beginning at most p ready tasks, then a new processor can not help any further because we then can at most keep p processors busy. In this case S_{p+1} shall work exactly like S_p, implying that the run times for S_p and S_{p+1} are the same.

However, if I has more than p ready tasks, then S_{p+1} shall initially schedule exactly the same tasks as S_p would do (we call those $x_1, \ldots, x_p$) and one *additional* task y.

We can now compare the run times of S_p and S_{p+1}. First of all, note that the tasks $x_1, \ldots, x_p$ are independent (one of our basic assumptions) and do not know whether they are currently scheduled by S_p or by S_{p+1}. This means, they all finish at the same respective times in S_p and S_{p+1}.

Of course, the task y finishes at some point in S_p, but it finishes *earlier* in S_{p+1} because it has been scheduled from the beginning. We now use induction to prove the claim: For small intrees (with, say, fewer than six tasks) we can easily test that the claim is true.

For S_{p+1}, we can argue as follows: If y is the first task to finish, we basically have the same situation as for S_p, but with the modification that we do not have to process the whole intree I but instead only a subtree, namely $I \setminus \{y\}$. It is clear that processing this subtree requires less time than processing the original intree.

If, on the other hand, one of the tasks $x_1, \ldots, x_p$ finishes first, we continue like S_p would have continued. If S_p chose y as next task, we can argue that the resulting run time for S_{p+1} is at most the run time for S_p because then y finishes earlier in S_{p+1} than in S_p. Otherwise, the expected run time for S_{p+1} still is smaller than the expected run time for S_{p+1} by induction.TODO: Improve this □

8.2 Size of the snapshot DAG

Similar to the reasoning in section **??**, we can research the size of a snapshot DAG for the P3 case. We conducted an experiment and examined the size for snapshot DAGs of intrees containing up to 14 tasks. Therefore, we generated all intrees (up to isomorphism) with a certain number of tasks (see section **??** for an algorithm). Then we computed the following for each intree:

- Number of distinct (i.e. non-isomorphic) subtrees.
- Number of snapshots that are constructed using the LEAF scheduling strategy (i.e. exhaustive search).

- The size of the *optimal* snapshot DAG. This DAG is constructed by excluding bad choices of the LEAF scheduler.

We do so because of the following: It is easily possible to construct an optimal schedule if we take the possible snapshot DAGs of the LEAF scheduler and only leave the choices that yield the best expected run time.

It is clear that the size of the snapshot DAG for an intree with n tasks is at most n^3 times as large as the number of subtrees of that particular intree. This can be easily seen by an argument similar to the one in section **??**, where we said that each subtree has three scheduled tasks and we have at most $\binom{n}{3} \leq n^3$ possibilities to choose three tasks from each intree.

We examined all intrees up to 15 tasks and computed for each of them the number of subtrees, the number of snapshots to be examined by an exhaustive search and the number of snapshots in the snapshot DAG for the optimal schedule.

The results are summed up in table **??**.

Tasks	Subtrees		Snapshots		"Optimal DAG"	
	Max	Avg	Max	Avg	Max	Avg
3	3	3.00	3	3.00	3	3.00
4	5	4.25	5	4.25	5	4.25
5	7	5.89	7	5.89	7	5.89
6	11	8.10	11	8.25	11	8.05
7	16	11.04	19	11.75	16	10.81
8	24	15.10	34	17.39	22	14.37
9	34	20.57	63	26.53	31	18.76
10	54	28.08	119	41.85	41	24.16
11	79	38.33	230	67.48	55	30.67
12	119	52.41	437	112.68	71	38.41
13	169	71.69	812	184.95	89	47.49
14	269	98.19	1510	304.41	113	58.05

Table 8.1: Number of subtrees, size of the optimized snapshot DAG depending on the number of tasks. "Subtrees" denotes the number of distinct subtrees. "Snapshots" shows the number of distinct snapshots that have to be examined if we try all possible schedules. The column "Optimal DAG" shows the size of the snapshot DAG describing the optimal schedule.

As we can see in table **??**, the number of subtrees is (at least for $n \geq 9$) significantly larger than the number of snapshots in the snapshot DAG for the optimal schedule. However, the overall number of snapshots is strictly higher than the number of subtrees.

Another interesting fact is that there is no "strict correlation" between the number of subtrees and the number of snapshots in the optimal snapshot DAG. That is, there are certain DAGs that have more non-isomorphic subtrees than another DAG, yet – on the other hand – more snapshots in the optimal snapshot DAG. As an example, consider the intrees T_1 described by 00011111 and T_2 described by 00001234: T_1 has 19 subtrees and its optimal snapshot DAG contains 13 snapshots, while T_2 has only 15 subtrees, but an optimal snapshot DAG containing 14 snapshots.

Moreover, intrees containing n tasks and having the maximal number of subtrees are (at least for $8 \leq n \leq 17$) are not the ones having the largest optimal snapshot DAG.

To determine the maximum size of the optimal snapshot DAG for the P3 case, it might be useful to investigate whether the trees that have a large snapshot DAG can be constructed according to a specific pattern. The intrees resulting in snapshot DAGs of maximum size are depicted in figure **??**. The intrees in this figure seem to behave quite chaotic and we were not able to deduce any pattern according to which they could be generated for general n.

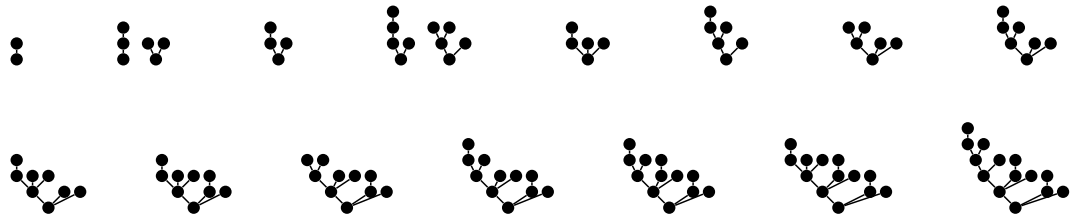


Figure 8.3: These are the intrees (containing between 2 and 14 tasks) for which the a brute-force algorithm has to generate maximally many snapshots to generate the optimal schedule (maximal compared to all other intrees with the same number of vertices).

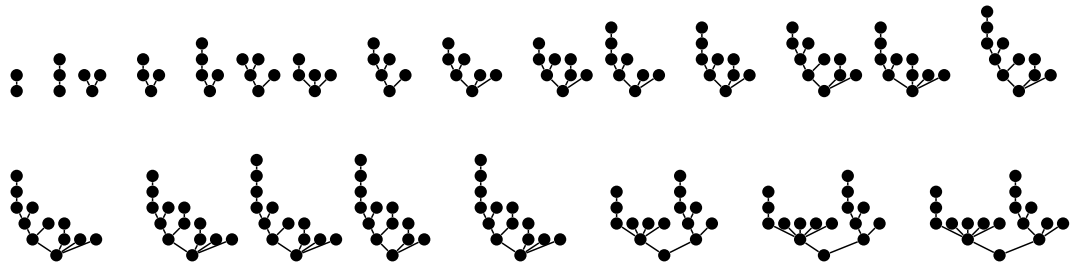


Figure 8.4: These intrees result in *optimal* snapshot DAGs that are larger than all other optimal snapshot DAGs resulting from intrees having the same number of tasks n ($2 \leq n \leq 17$ shown). There seems to be no simple pattern according to which these trees are constructed.

8.3 Compressing the snapshot DAG

When we considered the two processor case (see chapter **??**), we mentioned that all intrees with the same profile (see section **??**) have – scheduled on two processors – the same run time. We then were able to compress the snapshot DAG and to merge certain snapshots into one single snapshot.

Unfortunately, this behaviour does not transfer to the three processor case. The core requirement for two snapshots that could be merged was that they had the same run time. For three processors, we have seen that intrees with the same profile do not necessarily have the same (optimal) run time. As an example, consider the intrees $(0, 0, 1, 1, 3, 4)$ with optimal run time 4.81482 and $(0, 0, 1, 2, 4, 4)$ with optimal run time 4.81944 (but both having profile $[\![2, 2, 2, 1]\!]$. Additionally, we saw that HLF might even yield different run times depending on its choices of tasks (see section **??**).

However, we still can merge certain snapshots in some rare cases. Consider e.g. the intree $(0, 0, 0, 1, 1, 2, 2, 3)$ shown in figure **??**. This intree has the property that it is irrelevant whether we initially schedule tasks $4, 5, 8$ or tasks $4, 6, 8$: The optimal run time is in both cases 4.87243. This comes from the fact that, after the first task finishes, the resulting snapshots are the same and they are reached with the same probability.

That means: If two snapshots reach the same successors with the same probabilities, we can easily merge these snapshots. But this is not the only possibility. Consider the intree $(0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 10, 10)$ whose optimal schedule is shown in figure **??**. For this schedule, the respective snapshots with run times 6.02418 can be merged because they reach the same successor snapshots with the same probability. *After* merging these two snapshots, the snapshots whose run time is 6.45792 can be merged because *after* the first merge step these two snapshots reach the same snapshots with same probabilities.

That means, that we have to consider not only whether the direct successors are the same and reached with same probabilities, but we have to recursively examine whether we can merge some successors and whether after merging successors we can finally merge the respective snapshots.

Figure **??** shows the optimal schedule for the given intree, while figure **??** shows the snapshot DAG where snapshots are recursively have been merged.

It is a notable fact that merging snapshots can not be done easily. Unfortunately, it can not be

(a) Original schedule with unmerged snapshots.

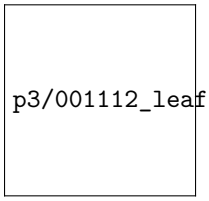
(b) First three levels of the snapshot DAG with merged snapshots.

Figure 8.5: The optimal schedule for $(0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 10, 10)$ allows certain snapshots to be merged.

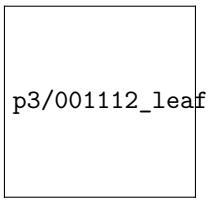
applied to snapshot DAGs arising from the LEAF scheduler. If we consider the first three levels of the LEAF schedule on the intree $(0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 10, 10)$ we observe that the corresponding snapshots that could have merged in the optimal schedule can *not* be merged directly in the LEAF snapshot DAG. Figure **??** explains why this is the case: If we use use exhaustive search (i.e. a LEAF scheduler), the corresponding snapshots that could be merged in the optimal schedule can not be merged because they have different expected run times.

Figure 8.6: LEAF schedule for the intree $(0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 10, 10)$. No snapshots can be merged because they all have different run times. Only important snapshots are shown in detail.

Moreover, we are not always allowed to merge certain snapshots just if their respective expected run times are equal. Consider e.g. the intree $(0, 0, 1, 1, 1, 2)$ whose LEAF schedule beginning with tasks $4, 5, 6$ contains two snapshots whose run time is 4.05556.



(a) Two snapshots have run time 4.05556, but they can *not* be merged.



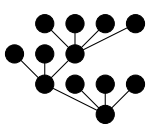
(b) The reason that the two snapshots shall not be merged is that they result from different tasks being finished.

Figure 8.7: A LEAF schedule for $(0, 0, 1, 1, 1, 2)$ contains two snapshots with the same run time that can not be merged.

8.4 Degenerate intrees

We now focus one one particular class on intrees, namely *degenerate intrees*. A degenerate intree is an intree that consists of one longest chain from the bottom to one leaf, and all other tasks are direct predecessors to one of the tasks within this longest chain. Another characterization is the following: On each level, *at most one task* has predecessors.

Here is an example for a degenerate intree with four levels and profile $[\![5, 3, 4, 2]\!]$:



Note that *all* degenerate intree with the same profile are isomorphic intrees.

8.4.1 HLF is optimal for degenerate intrees

We prove that HLF is optimal for degenerate intrees (and three processors). Therefore, we introduce some convenient notation that will later be useful.

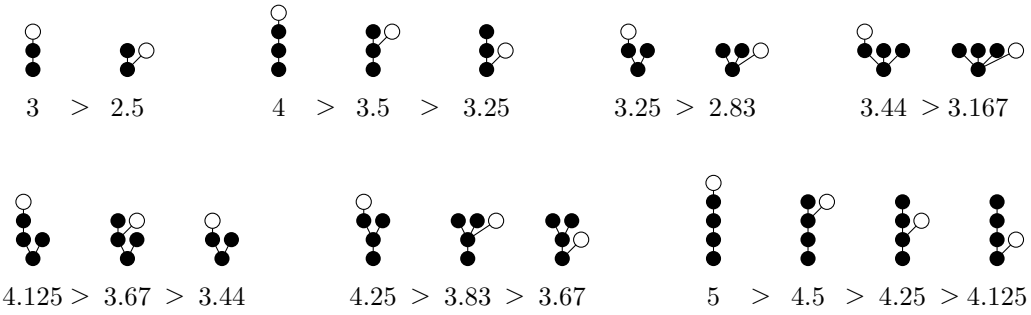


Figure 8.8: Adding new tasks to degenerate intrees with less than four nodes such that the resulting intrees are still degenerate. The original (2- resp. 3-node) intrees are drawn black, the newly added tasks are drawn white. Below each intree, we see the corresponding optimal expected run time. The lower the level of the newly added task, the lower the expected run time. This serves as basis for the induction proof for lemma **??**.

Definition 8.4. *Let I be an intree. By $T^*_{t_1,t_2,t_3}(I)$ we denote the optimal expected run time that can be acchieved if we* initially *schedule all tasks from the set $\{t_1, t_2, t_3\}$.*

Note that the notation from definition **??** does not necessarily require that we actually have three tasks (e.g. if $t_1 = t_2$).

Lemma 8.5. *Let I be a degenerate intree and x, y two (not necessarily distinct) ready tasks within this intree. Let z_1, z_2 be two new tasks that will be added to this intree with $level(z_1) > level(z_2)$ in a manner such that $I_1 := I \cup \{z_1\}$ and $I_2 := I \cup \{z_2\}$ are still degenerate intrees. Moreover, the tasks z_1 and z_2 shall be added in such a way that neither x nor y is a successor of z_1 or z_2 (i.e. x, y stay ready in I_1 resp. I_2).*

Then, if x, y and z_1 resp. z_2 are used as initial tasks (for I_1 resp. I_2), we have the following for the best acchievable expected run times (for respective initial tasks):

$$T^*_{x,y,z_1}(I \cup \{z_1\}) > T^*_{x,y,z_2}(I \cup \{z_2\}) \tag{8.1}$$

If we loosen the level condition to $level(z_1) \geq level(z_2)$, we obtain

$$T^*_{x,y,z_1}(I \cup \{z_1\}) \geq T^*_{x,y,z_2}(I \cup \{z_2\}).$$

Proof. We focus first on the case where $level(z_1) > level(z_2)$ and prove the claim by induction over the number of nodes:

The induction basis is the case where we have degenerate intrees with 3 tasks[1] (all of them are depicted in figure **??** (only the black nodes)).

If we add two tasks z_1 and z_2 with $level(z_1) > level(z_2)$ in a way such that the original ready tasks stay ready and the resulting intrees stay degenerate, we obtain the intrees depicted in figure **??** (trees *including* the white nodes). By simply computing the expected optimal run times, we can confirm our claim for intrees with 3 nodes.

We now do the induction step by considering an intree with n tasks. Let x, y be ready tasks and z_1 and z_2 to be added with $level(z_1) > level(z_2)$ such that the resulting intrees are degenerate. We can now compare the two runs that can occur if x, y, z_1 resp. x, y, z_2 are initially scheduled. Therefore, we consider what happens in $I_1 = I \cup \{z_1\}$ resp. $I_2 = I \cup \{z_2\}$ if either x, y or z_1/z_2 finishes first:

- If z_1 resp. z_2 is the first task to finish, the resulting intree is exactly I. Thus, the remaining run times for these cases are identical if the next task chosen is the same in both trees. We denote the

[1] We start with 3 tasks since these trees are the only ones that allow adding z_1 and z_2 on different levels such that both x and y stay ready. For an intree with two tasks (i.e. for the intree denoted by (0)), the claim can be seen by simply examining that $2.5 = T((0,0)) < T((0,1)) = 3$.

task that may be chosen additionally to x and y by z'. If it is the case that only x and y can be scheduled, we set $z' = x$ to simplify notation. The corresponding run time for the resulting intree is then $T^*_{x,y,z'}(I)$.

- If x is the first task to finish, then the resulting intrees are

$$I_1^x = I_1 \setminus \{x\} \quad \text{resp.} \quad I_2^x = I_2 \setminus \{x\}\,.$$

By x' we denote the task that is scheduled next in the optimal schedule for intree I_1^x. The expected optimal runtime for I_1^x in this situaion is then given by $T^*_{x',y,z_1}(I_1^x)$.

If there are only two ready tasks left in I_1^x (which then must be y and z_1), we set $x' = y$.

We now examine whether x' is also ready in the intree I_2^x:

  - If there are only two ready tasks left in I_1 (namely x and y), we set – as mentioned before – $x' = y$. Thus, in I_2^x, x' is still ready[2].
  - If x' is the direct successor of x, then x must have been the *single topmost task* and the *single predecessor* of x' (since I_1^x is a degenerate tree). However, since we assumed that $level(z_2) < level(z_1)$ and z_1 can not be a predecessor of x' (since x' is ready in I_1), it can not be the case that z_2 blocks x' in I_2^x. We conclude that in this case x' is ready in I_2^x.
  - If x' is not the direct successor of x, we recognize the fact that x' must reside on a certain level within the degenerate intree.

    If x' is *not* in the topmost level, it can not be blocked by z_2 because we assumed that z_2 is added in a way such that $I \cup \{z_2\}$ is still a degenerate intree.

    Otherwise (if x' is a topmost task), z_2 can not be added *above* (i.e. as a predecessor of) x' because we assumed $level(z_1) > level(z_2)$.

    Again, x' is ready in I_2^x.

We observed that for I_2^x, the task x' must be ready.

The intrees I_1^x and I_2^x have exactly n tasks – and they have a common subtree, namely

$$I^x := I_1^x \setminus \{z_1\} = I_2^x \setminus \{z_2\}\,.$$

We can now apply the induction hypothesis for the intree I^x, since this intree contains only $n-1$ tasks: We have an intree with $n-1$ tasks (namely I^x) and two tasks z_1 and z_2 with $level(z_1) > level(z_2)$, $I_1^x = I^x \cup \{z_1\}$ and $I_2^x = I^x \cup \{z_2\}$, implying that $T^*_{x',y,z_1}(I_1^x) > T^*_{x',y,z_2}(I_2^x)$.

- If y is the first task to finish, we argue similar to the x case, thereby considering I_1^y, I_2^y and I^y which are all defined analogously. This finally yields the following inequality: $T^*_{x,y',z_1}(I_1^y) > T^*_{x,y',z_2}(I_2^y)$ (where y' denotes the task that is scheduled next after y finishes).

The above considerations are illustrated in figure **??**.
Now we argue that the (optimal) run times for I_1 and I_2 can be computed as follows:

$$T^*_{x,y,z_1}(I_1) = \frac{1}{3} + \frac{1}{3} \cdot \left(T^*_{x,y,z'}(I) + T^*_{x',y,z_1}(I_1^x) + T^*_{x,y',z_1}(I_1^y)\right)$$
$$T^*_{x,y,z_2}(I_2) = \frac{1}{3} + \frac{1}{3} \cdot \left(T^*_{x,y,z'}(I) + T^*_{x',y,z_2}(I_2^x) + T^*_{x,y',z_2}(I_2^y)\right)$$

We will now use the aforementioned inequalities (i.e. $T^*_{x',y,z_1}(I_1^x) > T^*_{x',y,z_2}(I_2^x)$ and $T^*_{x,y',z_1}(I_1^y) > T^*_{x,y',z_2}(I_2^y)$):

$$\begin{aligned} T^*_{x,y,z_1}(I_1) &= \frac{1}{3} + \frac{1}{3} \cdot \left(T^*_{x,y,z'}(I) + T^*_{x',y,z_1}(I_1^x) + T^*_{x,y',z_1}(I_1^y)\right) > \\ &> \frac{1}{3} + \frac{1}{3} \cdot \left(T^*_{x,y,z'}(I) + T^*_{x',y,z_2}(I_2^x) + T^*_{x,y',z_2}(I_2^y)\right) = T^*_{x,y,z_2}(I_2). \end{aligned}$$

[2] It may even be the case that I_2^x contains some additional ready tasks that are not ready in I_1^x.

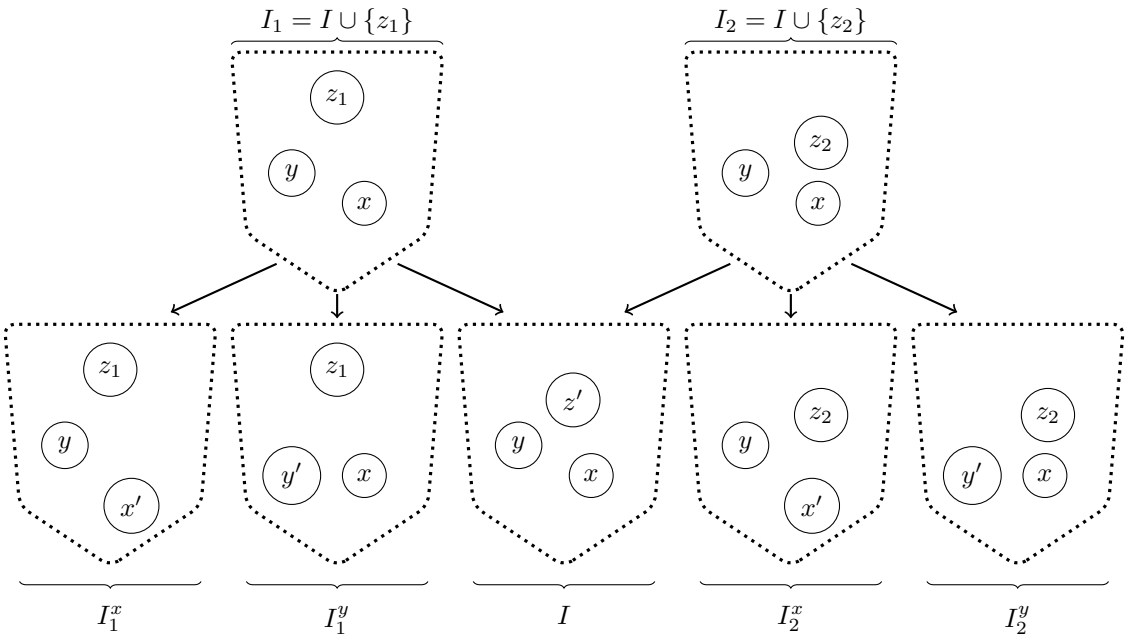


Figure 8.9: Proof sketch for lemma **??**. By induction hypothesis, we have that $T^*_{x',y,z_1}(I_1^x) > T^*_{x',y,z_2}(I_2^x)$ and $T^*_{x,y',z_1}(I_1^y) > T^*_{x,y',z_2}(I_2^y)$, from which we deduce $T^*_{x,y,z_1}(I_1) > T^*_{x,y,z_2}(I_2)$. Note that z_2 can not block x' or y' since z_1 didn't block any of the two and we required that adding z_1 resp. z_2 still results a degenerate tree.

This proves our claim for $level(z_1) > level(z_2)$. It is simple to obtain the claim for the version where $level(z_1) \geq level(z_2)$: If the levels are the same, the resulting degenerate intrees I_1 and I_2 are isomorphic (and there is an isomorphism that maps z_1 onto z_2), thus the optimal run time is the same. If the levels are different, we argument as in the beginning of the proof. □

Application of lemma ?? We presented lemma **??** in a way that involved adding tasks to a certain intree, because this related well to our proof technique. In practice, we can use it to compare two degenerate intrees with n nodes and that have a common subtree containing $n-1$ nodes.

We can now derive the following theorem.

Theorem 8.6. *Degenerate intrees are optimally scheduled by HLF.*

Proof. Consider a degenerate intree with $n = 3$ tasks. It is trivially clear that for P3, a HLF schedule is optimal for this intree (see figure **??** to see what these intrees look like – we simply can conclude examine all possible schedules and see that the optimal ones is exactly HLF).

Consider now a degenerate tree I with n nodes and assume that we know that for all degenerate trees with $n-1$ nodes HLF is optimal for three processors. If I has two or less topmost tasks, it is obvious that we have to use HLF (since we can use at most two processors and HLF is known to be optimal for two processors — see chapter **??** or refer to [**chandyreynoldsshortpaper1975**]).

Thus, we only have to focus on the case where I has at least three ready tasks.

If we choose three HLF-conform tasks x, y, z of I, we can argue as follows: If x is the task that finishes first, for the resulting subtree $I \setminus \{x\}$, we can be sure that we can choose the next task such that we adhere to HLF, thus choosing the optimal solution for $I \setminus \{x\}$ (by induction hypothesis). The same holds if y or z finishes first.

We now consider a (possibly) non-HLF schedule for I and compare it to the HLF schedule described before. Let x', y', z' be the tasks to be chosen such that at least $x \neq x'$ or $y \neq y'$ or $z \neq z'$. We can – without loss of generality – assume $level(x') \leq level(x)$, $level(y') \leq level(y')$, $level(z') \leq level(z)$. If x' is the first task to finish, we consider the degenerate intree $I \setminus \{x'\}$. Since $I \setminus \{x'\}$ is a degenerate

intree with $n-1$ tasks, it would be optimal to use HLF. However, using HLF may or may not be possible depending on our previous choices of y' and z'. That is, the optimum for $I \setminus \{x'\}$ *might* be acchieved if we chose y' and z' accordingly. From this we can conclude that the optimal expected run time – if we start with x', y' and z' – is at least $T_{HLF}(I \setminus \{x'\})$, where T_{HLF} denotes the run time for HLF (which we know, by induction, is optimal).

Now we compare the run time for $I \setminus \{x'\}$ to the optimal run time for $I \setminus \{x\}$, which is exactly given by $T_{HLF}(I \setminus \{x\})$. We recognize that $I \setminus \{x'\}$ and $I \setminus \{x\}$ have a common subtree, namely $I \setminus \{x, x'\}$ with $n-2$ tasks.

Moreover, we know that for both $I \setminus \{x\}$ and $I \setminus \{x'\}$ HLF is optimal (because of our induction hypothesis) and that for $I \setminus \{x\}$ and $I \setminus \{x'\}$ both y and z must be in the optimal (i.e. HLF) schedule since y and z are two of the three HLF-conform tasks. At last, we have $level(x) \geq level(x')$. Thus, we can apply lemma **??** for the degenerate tree $I \setminus \{x, x'\}$ with tasks y and z and deduce that

$$T^*_{x',y,z}(I \setminus \{x\}) \leq \underbrace{T^*_{x,y,z}(I \setminus \{x'\})}_{\text{Optimal for } I \setminus \{x'\}}.$$

Equality holds if x and x' are on the same level.

As stated before, for the intree $I \setminus \{x'\}$, the schedule chosen *might* be a non-HLF schedule. In this case, the schedule performs even worse than $T_{x,y,z}(I \setminus \{x'\})$, since the optimal schedule would choose x, y, z as scheduled tasks. This means that

$$T^*_{x,y,z}(I \setminus \{x'\}) \leq T_{x'',y',z'}(I \setminus \{x'\}),$$

where x'' is the task chosen by the non-HLF schedule, finally implying (by $T_{HLF}(I \setminus \{x\}) \leq T_{x',y,z}(I \setminus \{x\})$) that

$$T_{HLF}(I \setminus \{x\}) \leq T_{x'',y',z'}(I \setminus \{x'\}) \quad \text{and } T_{x',y,z}(I \setminus \{x\}) \leq T_{x'',y',z'}(I \setminus \{x'\}).$$

We argue similar for tasks y and y' resp. z and z' and finally obtain the following:

$$\begin{aligned} T_{x',y,z}(I \setminus \{x\}) &\leq T_{x'',y',z'}(I \setminus \{x'\}) \\ T_{x,y',z}(I \setminus \{y\}) &\leq T_{x',y'',z'}(I \setminus \{y'\}) \\ T_{x,y,z'}(I \setminus \{z\}) &\leq T_{x',y',z''}(I \setminus \{z'\}) \end{aligned}$$

Note that if e.g. $x = x'$, the above three equations are still satisfied. Combining the above inequalities yields that

$$\begin{aligned} T_{x,y,z}(I) &\leq \frac{1}{3} + \frac{1}{3} \cdot (T_{x',y,z}(I \setminus \{x\}) + T_{x,y',z}(I \setminus \{y\}) + T_{x,y,z'}(I \setminus \{z\})) \leq \\ &\leq \frac{1}{3} + \frac{1}{3} \cdot (T_{x'',y',z'}(I \setminus \{x'\}) + T_{x',y'',z'}(I \setminus \{y'\}) + T_{x',y',z''}(I \setminus \{z'\})) \leq \\ &\leq T_{x',y',z'}(I) \end{aligned}$$

This finally shows that a HLF schedule is at least as good as an arbitrary other task, meaning that HLF is optimal for three processors on degenerate trees. □

8.4.2 Comparison of exhaustive search vs. HLF on degenerate binary trees

We researched degenerate binary trees, i.e. trees whose sequence has the structure

$$(0, 0, a_0, a_1, a_2, a_3, a_4, \ldots, a_n),$$

for $n+3$ being the total number of tasks within the intree. The values a_i can be recursively defined as follows:

$$a_k = \begin{cases} 1, & \text{if } k \leq 1 \\ a_{k-1}, & \text{if } k > 1 \text{ is odd} \\ a_{k-1} + 2, & \text{if } k > 1 \text{ is even} \end{cases}$$

Tasks	Snapshots		Tasks	Snapshots		Tasks	Snapshots	
	Overall	HLF		Overall	HLF		Overall	HLF
3	3	3	11	150	41	19	12600	175
4	5	5	12	276	57	20	22594	221
5	7	7	13	477	63	21	34883	231
6	11	11	14	884	85	22	61774	287
7	17	14	15	1477	92	23	93775	298
8	28	21	16	2717	121	24	164187	365
9	48	25	17	4398	129	25	245852	377
10	85	36	18	7991	166	26	426089	456

Table 8.2: Number of snapshots for degenerate binary trees in the P3 case. The first column shows the number of tasks. "Overall" denotes the number of distinct snapshots that are explored if an optimal schedule is constructed by examining all schedulings. "HLF" denotes the number of distinct snapshots for HLF.

That is, degenerate binary trees have sequences of the form $(0, 0, 1, 1, 3, 3, 5, 5, 7, 7, 9, \dots)$.

We now examine how many snapshots are considered if we compute the optimal P3 schedule by considering *all* possibilities and afterwards discarding the bad ones. The results are summed up in table **??**. We clearly observe that the number of snapshots grows exponentially (at least within the range for n under consideration). A simple pattern that can be observed from table **??** is that (at least for $n \leq 26$) that the number of snapshots for a degenerate binary tree with n tasks is greater than twice the number of snapshots for a degenerate binary tree with $n-2$ tasks. If $S(n)$ denotes the number of snapshots for a degenerate binary tree, we can formulate $S(n) > S(n-2)$, which we can (by induction) convert to $S(n) > \sqrt{2}^n$.

This can be illustrated by the fact that degenerate binary intrees are fully determined by their profile (please see section **??** for the definition of profiles). A degenerate binary tree has a profile of the form

$$[\![a, 2, 2, 2, 2, \dots, 2, 1]\!],$$

where a is either 1 or 2. Assume the length of the profile (i.e. the height of the degenerate binary tree) is exactly l. Then, we have $2 \cdot (l-2) + 1 + a = 2l - 1 + a$ tasks in total. Assume for now that $a = 2$ (i.e. we are dealing with a complete degenerate binary tree) and $l > 2$.

A subtree of a this degenerate binary intree having height l' has a profile of the form

$$[\![a_0, a_1, a_2, \dots, a_{l'-2}, 1]\!],$$

where $1 \leq a_0 \leq a$ and $1 \leq a_i \leq 2$ for all $i \in \{1, 2, \dots, l-2\}$. Using basic combinatorics, we can tell that there must be

$$\sum_{l'=0}^{l-1} 2^{l'} = 2^l - 1$$

distinct subtrees if $a = 2$ for profile length (resp. intree depth) l.

If $a = 1$ and we have a profile length of l, we cann argue that there must be as many subtrees as for the profile without the first item (then of length $l-1$) plus the number of profiles of length exactly $l-1$ with one additional 1 prepended. These are exactly 2^{l-2}. This, in total, leads to our desired bound for $S(n)$.

On the other hand, the number of snapshots needed by HLF seems to concide with the sequence [**oeisA055803**].

We have seen that degenerate binary intrees, while having a possibly huge amount of snapshots, can be examined by considering much less snapshots because we know we can process them optimally by HLF. You can compare the number of HLF snapshots to the number of overal snapshots by looking at table **??**.

8.5 Parallel chains

We will now consider another class of intrees that can be shown to have HLF as optimal schedules.

Definition 8.7 (Parallel chain)**.** *Let I be an intree. We call I a* parallel chain*, if each task except the root has at most one predecessor. The root may have arbitrarily many predecessors.*

We show an example of a parallel chain in figure **??**.

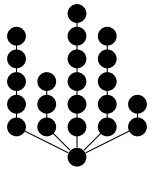


Figure 8.10: Parallel chains have the property that – except for the root – all tasks have at most one direct predecessor. Note that for parallel chains, HLF admits no ambiguities in the sense that all different possible choices of HLF result in equivalent snapshots.

8.5.1 Parallel chains are optimally scheduled by HLF

We start out by a lemma that we will use later. This lemma is an analogue variant of a lemma described in [**chandyreynoldsshortpaper1975**]. While their lemma works for two processors and general intrees, our variant is for three processors, but restricted to parallel chains.

Lemma 8.8. *Let I be an intree and x and y two ready tasks with $level(x) > level(y)$. Then, $T_{HLF}(I \setminus \{x\}) < T_{HLF}(I \setminus \{y\})$, where $T_{HLF}(I)$ denotes the run time that is acchieved if intree I is scheduled according to HLF.*

Proof. We prove the claim by induction. For parallel chains with fewer than 6 tasks (all parallel chains with fewer than 5 tasks are trivially optimally scheduled by HLF), the fact can be easily proven. Note that it suffices to examine all parallel chains with at least 3 ready tasks, since intrees with two or less tasks are optimally scheduled by HLF (according to the fact that only two processors can be used for those intrees and the fact that for two processors HLF is optimal). The only parallel chain with fewer than 6 tasks with more than 2 ready tasks is the intree $(0, 0, 0, 1)$. This parallel chain (and its resulting parallel chains after removing HLF-conform tasks) are shown in figure **??**.

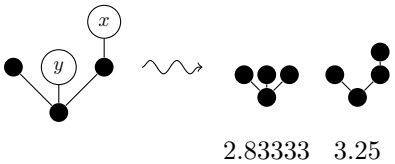


Figure 8.11: Removing tasks on lower levels from parallel chains results in higher run times than removing tasks on higher levels. Original parallel chain $(0, 0, 0, 1)$, and the parallel chains resulting from removing x resp. y with their respective expected HLF run times. This serves as base case for the induction in the proof of lemma **??**.

We consider the HLF schedules on the intrees $I \setminus \{x\}$ and $I \setminus \{y\}$, with variable names as aforementioned. Note that the intrees $I \setminus \{x\}$ and $\setminus\{y\}$ have at least two common HLF-tasks (i.e. tasks that are scheduled by HLF). We call these two tasks x_1 and x_2. For the intree $I \setminus \{x\}$, we call the third HLF-task x_3 and can compute its expected run time (for HLF) by

$$T_{HLF}(I \setminus \{x\}) = \frac{1}{3} + \frac{1}{3} \cdot T(I \setminus \{x, x_1\}) + \frac{1}{3} \cdot T(I \setminus \{x, x_2\}) + \frac{1}{3} \cdot T(I \setminus \{x, x_3\}).$$

On the other hand, for the intree $I \setminus \{y\}$, we call the third HLF-task x' and can compute its expected run time analogously by

$$T_{HLF}(I \setminus \{y\}) = \frac{1}{3} + \frac{1}{3} \cdot T(I \setminus \{y, x_1\}) + \frac{1}{3} \cdot T(I \setminus \{y, x_2\}) + \frac{1}{3} \cdot T(I \setminus \{y, x'\}).$$

We can now compare $T(I \setminus \{x, x_1\})$ with $T(I \setminus \{y, x_1\})$, $T(I \setminus \{x, x_2\})$ with $T(I \setminus \{x, x_2\})$ and $T(I \setminus \{x, x_3\})$ with $T(I \setminus \{y, x'\})$.

The intrees $I \setminus \{x, x_1\}$ and $I \setminus \{y, x_1\}$ have a common subtree, namely $I \setminus \{x_1\}$. By induction ($I \setminus \{x_1\}$ has $n - 1$ tasks, $level(x) > level(y)$), we know that the expected HLF run time for $I \setminus \{x, x_1\}$ is less than the expected run time for $I \setminus \{y, x_1\}$. Note that $I \setminus \{x, x_1\}$ has at least as many ready tasks as $I \setminus \{y, x_1\}$. So, if $I \setminus \{y, x_1\}$ has two or less ready tasks, the claim is true, because it has been shown for two processors in [**chandyreynoldsshortpaper1975**] and we – moreover – can use a third processor for $I \setminus \{x, x_1\}$. This consideration can be done for x_2 in the same manner.

It remains to compare $T(I \setminus \{x, x_3\})$ and $T(I \setminus \{y, x'\})$. If $I \setminus \{y, x'\}$ has fewer ready tasks than $T(I \setminus \{x, x_3\})$, then we trivially have $T(I \setminus \{x, x_3\}) \leq T(I \setminus \{y, x'\})$ (similar to the case before, it has been shown for two processors, and we additionally can use a third processor for $I \setminus \{x, x_3\}$). We, thus, focus on other cases.

- If $level(x') = level(x_3)$, the intrees $I \setminus \{x'\}$ and $I \setminus \{x_3\}$ are isomorphic (we are dealing with parallel chains only), and can – thus – be considered equal in our computations (we can do so because HLF can be seen as deterministic for parallel chains). That is, we can w.l.o.g. assume $x' = x_3$ if the levels of x' and x_3 are equal. In this case, we have that $T(I \setminus \{x, x_3\})$ and $T(I \setminus \{y, x'\})$ share a common supertree (namely $I \setminus \{x_3\} = I \setminus \{x'\}$), and we know – by induction since $level(x) > level(y)$ that $T(I \setminus \{x, x_3\}) < T(I \setminus \{y, x'\})$.
- If $level(x') \neq level(x_3)$, we know that x_3 is *not* a HLF task in $I \setminus \{y\}$, but x' is. This means that x' can only be a HLF task since it is not part of $I \setminus \{y\}$. Therefore, we conclude that y must be the same as x_3 (implying w.l.o.g. $I \setminus \{x, y\} = I \setminus \{y, x'\}$). This means that $level(x) > level(y) = level(x_3)$. Moreover, because x' is only a HLF task because x_3 is not existent in $I \setminus \{y\}$, we know that $level(x') < level(x_3)$. Combining the inequalities yields $level(x) > level(x')$. We can now argue that $I \setminus \{y\}$ is (w.l.o.g.) a common supertree of $I \setminus \{x, y\} = I \setminus \{x, x_3\}$ and $I \setminus \{y, x'\}$ with $n - 1$ tasks. Since – as explained before – $level(x) > level(x_3)$, we can apply the induction hypothesis and conclude that $T(I \setminus \{x, x_3\}) \leq T(I \setminus \{y, x'\})$.

We have now shown, that in any case the following hold:

$$T(I \setminus \{x, x_1\}) \leq T(I \setminus \{y, x_1\})$$
$$T(I \setminus \{x, x_2\}) \leq T(I \setminus \{x, x_2\})$$
$$T(I \setminus \{x, x_3\}) \leq T(I \setminus \{y, x'\})$$

These inequalities can be used to derive the desired inequality. □

Remark: In the above proof, we used the fact that, for parallel chains, HLF does not admit any ambiguities (i.e. all tasks that can be chosen by HLF as the next task to be scheduled result in equivalent snapshots). We used this fact in the case distinction where we said that two intrees are isomorphic and we can consider them equal for our computations. This, however, is something that can not be extended to arbitrary intrees for three processors.

We will now use lemma **??** to derive the following theorem.

Theorem 8.9. *Parallel chains are optimally scheduled by HLF.*

Proof. We prove the claim by induction and can clearly observe that for any parallel chain with fewer than 5 tasks, HLF is optimal (because there is no other possibility to schedule the tasks).

Consider the HLF run time for a parallel chain I with n tasks. Let x_1, x_2, x_3 denote the HLF tasks of I. We can compute the expected run time by

$$T_{HLF}(I) = \frac{1}{3} \cdot T(I \setminus \{x_1\}) + \frac{1}{3} \cdot T(I \setminus \{x_2\}) + \frac{1}{3} \cdot T(I \setminus \{x_3\}).$$

For the parallel chains $I \setminus \{x_1\}, I \setminus \{x_2\}$ and $I \setminus \{x_3\}$ (each having $n-1$ tasks) we know by induction that they are optimally scheduled by HLF. Note that – if we start in an HLF manner – we can choose the next tasks in a way such that for each of these intrees we behave according to HLF (because x_1, x_2 and x_3 are HLF tasks of I). This means, we can acchieve optimal run times for each of the three intrees.

Now consider any other initial choice of tasks y_1, y_2, y_3. We can (w.l.o.g.) assume that $level(y_i) \leq level(x_i)$ for $i = 1, 2, 3$ and we can assume (w.l.o.g.) $level(y_1) < level(x_1)$. The optimal expected run time for this initial choice of tasks is given by

$$T_{\{y_1,y_2,y_3\}}(I) = \frac{1}{3} \cdot T_{\{y_2,y_3,z_1\}}(I \setminus \{y_1\}) + \frac{1}{3} \cdot T_{\{y_1,y_3,z_2\}}(I \setminus \{y_2\}) + \frac{1}{3} \cdot T_{\{y_1,y_2,z_3\}}(I \setminus \{y_3\}),$$

where z_i denotes the optimal task to be chosen if y_i is the first task to finish and the expected run times noted above are the optimal ones that can be acchieved. We set z_i to $y_{i+1(\mod 3)}$ if there are only two ready tasks in a subtree.

We now apply lemma **??**: The parallel chains $I \setminus \{y_1\}$ and $I \setminus \{x_1\}$ share I as common supertree and $level(x_1) > level(y_1)$. Applying the lemma shows that $T_{HLF}(I \setminus \{y_1\}) > T_{HLF}(I \setminus \{x_1\})$. We know by induction that $I \setminus \{y_1\}$ is optimally scheduled by HLF, implying that $T_{y_2,y_3,z_1}(I \setminus \{y_1\}) \geq T_{HLF}(I \setminus \{y_1\})$. Combining the two inequalities gives $T_{y_2,y_3,z_1}(I \setminus \{y_1\}) > t_{HLF}(I \setminus \{x_1\})$.

Similar to the proof of lemma **??**, we can argue that the parallel chain $I \setminus \{x_1\}$ has at least as many ready task as $I \setminus \{y_1\}$ and, thus, that $I \setminus \{x_1\}$ has a lower expected HLF run time than $I \setminus \{y_1\}$.

The arguments work similar for x_2 resp. x_3 with y_2 resp. y_3 (where we can even allow the run times to be equal), resulting in our claim.

□

8.5.2 Comparison of exhaustive search and HLF

Similarily to the comparison done in section **??**, we are interested in what we gain by the fact that parallel chains are optimally scheduled by HLF and, thus, need not be examined by an exhaustive search.

To to this, we introduce a shorthand notation for parallel chains for this section: Instead of writing the whole intree in the usual manner, we focus on the lengths of the parallel chains. The parallel chain in figure **??** consists of a root with five chains of lengths 5, 3, 6, 5 and 2. We simply write down the chain lengths as a tuple $(5, 4, 6, 5, 2)$.

The results are summed up in table **??**. This table compares the scheduling approaches for parallel chains with a given amount of chains, where each chain has the same length. Other comparisons for selected parallel chains can be found in table **??**.

L \ N	3	4	5	6
2	10	18	30	46
3	20	50	110	210
4	35	115	315	715
5	56	231	756	1981
6	84	420	1596	4732

(a) Number of snapshots with examined by exhaustive search.

L \ N	3	4	5	6
2	10	14	18	22
3	20	30	40	50
4	35	55	75	95
5	56	91	126	161
6	84	140	196	252

(b) Number of snapshots examined by HLF.

Table 8.3: Parallel chains: Comparison of the number of snapshots needed by exhaustive search in contrast to HLF. Each table denotes the number of chains (N) to the right, and the length of each chain (L) downwards.

We can see that (deterministic) HLF can drastically reduce the number of snapshots that have to be examined. This, of course, reduces computation times for parallel chains.

Chain lengths	Exhaustive search	HLF
$(4, 3, 6, 3)$	233	99
$(3, 2, 2, 10)$	247	126
$(1, 2, 3, 4)$	63	37
$(1, 2, 3, 4, 5)$	356	85
$(1, 2, 3, 4, 5, 6)$	2007	172
$(1, 2, 3, 4, 5, 6, 7)$	10760	315

Table 8.4: Comparing exhaustive search to HLF for selected parallel chains.

8.6 Improving exhaustive search

We have seen that degenerate intrees and parallel chains are optimally scheduled by HLF. We can now try to improve exhaustive search by proceeding according to (deterministic) HLF from the point on where we have a degenerate intree or a parallel chain. When we do so, we assume the following:

Assumption: We assume that for degenerate intrees and parallel chains, that, if already two tasks are scheduled, the optimal task to be scheduled as third task should be chosen according to HLF.

Using this assumption, we can exclude snapshots that are non-HLF from the point on where the current intree is a degenerate trees or a parallel chain.

Please note that this assumption is not proven. Our experiments suggest that it is true. Also note that we possibly exclude too many snapshots, if the assumption turns out to be false. Nevertheless, we will use the assumption to research the number of snapshots that have to be examined by an improved exhaustive search.

If we recognize that this improved exhaustive search performs significantly better than an ordinary exhaustive search, it might be worth proving (or possibly even disproving) the assumption. If not, it is probably not rewarding to examine whether the assumption is true or not (because our assumption (possibly) allows us to exclude more snapshots).

Tasks	LEAF		SCLEAF		Ratio	
	Max	Avg	Max	Avg	Max	Avg
5	7	5.89	7	5.89	1	1
6	11	8.25	11	8.25	1	1
7	19	11.75	19	11.60	1	0.99
8	34	17.39	32	16.74	0.94	0.96
9	63	26.53	52	24.65	0.82	0.93
10	119	41.85	96	37.56	0.81	0.90
11	230	67.48	195	58.96	0.85	0.87
12	437	112.68	405	95.18	0.93	0.84
13	812	184.95	738	156.44	0.91	0.85
14	1510	304.41	1393	259.95	0.92	0.85

Table 8.5: "Improved" LEAF scheduling with HLF when encountering a degenerate intree or a parallel chain

Table **??** compares the number of snapshots that are needed by exhaustive search (LEAF) to the number of snapshots needed by improved (HLF for degenerate intrees and parallel chains) exhaustive search. It shows that we could not reduce the number of snapshots to under 80% in the average which is somewhat dissappointing because the tables **??**, **??** and **??** suggested a more remarkable performance gain.

We have seen that working with HLF for degenerate intrees and parallel chains has – in contrast to the benchmarks described in tables **??** and **??** – only a minor impact on the total number of snapshots needed. It is an interesting question why we do not observe an improvement as strong as it was when the *original* intree was already a degenerate intree or a parallel chain. The reason for this behaviour is

that, if we reach a degenerate intree or a parallel chain, we can possibly reach it in *many different ways*, requiring the algorithm to examine HLF for each of the reached intrees.

As an example, consider the intree $(0, 0, 1, 1, 3, 3, 5, 1, 2)$. Note that this intree is neither a degenerate intree nor a parallel chain. However, it is quite close to being a degenerate intree (by simply removing one task 9). However, since an exhaustive search (even the improved one) has to start out by ordinarily examining all combinations of settings for the original intree. Only under certain circumstances, it reaches a situation where it can apply HLF for a degenerate intree. But, as you can see from figure **??**, it is clear from the beginning which degenerate intrees we reach.

Each degenerate intree that is reached still has to be examined seperately by HLF. This explains why this improvement, while still reducing the number of required snapshots, is not as useful if only a subtree of the original intree is a degenerate intree (or a parallel chain). Note that in figure **??**, there are certain intrees that occur in both schedules, but *with different sets of scheduled tasks* in each schedule (see figure **??** for the respective intree). This requires us to examine both of these snapshots seperately by HLF to compute the optimal run time.

Remark: Even if – at the point of computation – there are different snapshots containing the same (or an isomorphic) intree, we did not observe one single instance where the *optimal schedule* contained two distinct snapshots with two isomorphic intrees. This lead us to the following conjecture:

Conjecture 8.10. *Let x and y be two distinct snapshots in an optimal snapshot DAG. Then, their respective intrees are not isomorphic.*

8.7 Improving exhaustive search with conjectures

As we have seen in conjecture **??**, we suspect that if there are more topmost tasks than processors, then we have to schedule as many topmost tasks as possible.

Even if we were not able to prove this conjecture, we still wrote a scheduler that exploits it, because it can drastically reduce the number of snapshots that have to be considered by an exhaustive search.

That is, we now consider a LEAF scheduler that

- uses HLF for degenerate intrees and parallel chains ...
- ...and discards snapshots that have unscheduled topmost tasks and scheduled non-topmost tasks.

We do an analogue comparison to the one done in section **??** and show the results in table **??**.

Tasks	LEAF		CSCLEAF		Ratio	
	Max	Avg	Max	Avg	Max	Avg
5	7	5.89	7	5.89	1	1
6	11	8.25	11	8.05	1	0.98
7	19	11.75	16	10.83	0.84	0.92
8	34	17.39	28	14.66	0.82	0.84
9	63	26.53	46	20.19	0.73	0.76
10	119	41.85	81	28.83	0.68	0.69
11	230	67.48	148	42.92	0.64	0.64
12	437	112.68	327	66.52	0.75	0.59
13	812	184.95	564	106.12	0.69	0.57
14	1510	304.41	1090	172.44	0.72	0.57
15			1959	282.36		
16			3637	462.81		
17			6399	755.86		

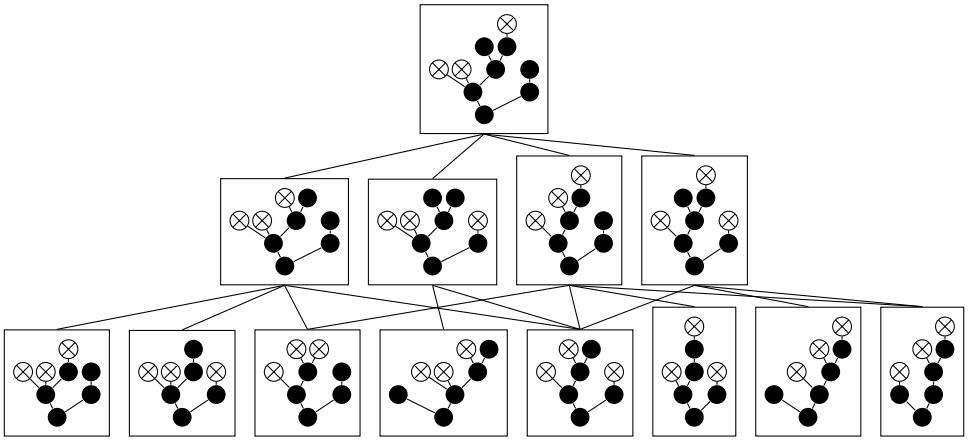
Table 8.6: TODO: Fill table!Conjecture-improved LEAF scheduling with HLF when encountering a degenerate intree or a parallel chain and scheduling as much topmost tasks as possible

We can see that the results are better than the ones we acchieved by only working with HLF from the point on where we come to degenerate intrees or parallel chains. Most notably, we gain over 30% on average for intrees with more than 9 tasks.

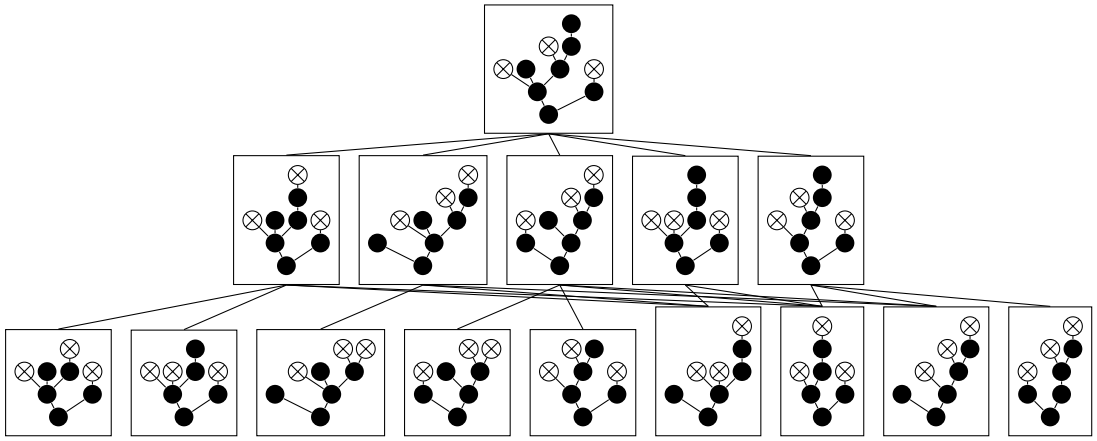
Still, these conjectures and improvements might serve as a starting point for further optimizations and exclusion rules for snapshots that can be omitted by an exhaustive search.

8.8 Conclusion

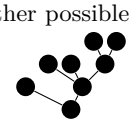
We have seen two very regular classes of intrees that are optimally scheduled by HLF. Unfortunately, even simplest combinations of the two classes are not guaranteed to be optimally scheduled by HLF. As an example, consider the intree $(0, 0, 1, 2, 3, 4, 6, 6, 8, 8)$ (depicted in figure **??**) whose components attached to its root are a degenerate intree with 7 tasks and a simple chain with 3 tasks. As shown in section **??**, for this intree, HLF is strictly suboptimal.



(a) One possible schedule.



(b) Another possible schedule.



(c) This intree occurs in both schedules, but with different sets of scheduled tasks, requiring to HLF to examine both variants if we compute the optimal run time.

Figure 8.12: Two possible schedules for intree $(0, 0, 1, 1, 3, 3, 5, 1, 2)$ that must be examined by exhaustive search to compute the optimal schedule. These can be used to illustrate why improving exhaustive search for degenerate intrees and parallel chains is not that useful in some cases. There are several screenshots containing a degenerate intree, each of whom must be processed by HLF seperately.