CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Case Study for Progressive Algorithms

An Investigation into Progressive Extraction of Intermediate Solutions for The Weighted Interval Scheduling Problem

Master's thesis in Computer Science and Engineering

Johannes Ringmark

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

A Case Study for Progressive Algorithms

An Investigation into Progressive Extraction of Intermediate Solutions for The Weighted Interval Scheduling Problem

Johannes Ringmark

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

A Case Study for Progressive Algorithms
An Investigation into Progressive Extraction of Intermediate Solutions for The Weighted Interval Scheduling Problem
Johannes Ringmark

Supervisor: Peter Damaschke
Examiner: Richard Johansson

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

A Case Study for Progressive Algorithms
An Investigation into Progressive Extraction of Intermediate Solutions for The Weighted Interval Scheduling Problem
Johannes Ringmark
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Asserted convergence characteristics shared by all serial algorithms have traditionally not been as prominent a measure of quality as time complexity has been. In instances relying on the exploration of large datasets, a convergence bound on interactively guided exploration approaches could act complementary to traditional time complexity, and constitute a alternative starting point for algorithm design. In an attempt to further explore the plausibility of this conjecture, a theoretical framework (Alewijnse, 2014) for convergence analysis through decomposition into consecutive intermediate computations is adopted, and its resulting intermediate solutions are used as a mean of empirical algorithm convergence analysis and categorization. Different scenarios related to the weighted interval scheduling problem are explored in this light, which is chosen primarily based on its documented compatibility with dynamic programming approaches. The result is presented as asymptotic upper bound functions on the convergence along with conjectures on upper bound hardness with respect to two different error functions, one adapted for stochastically and one for deterministically rooted algorithms.

Keywords: *Weighted Interval Scheduling*, *Progressive Algorithms*, *Convergence Analysis*, *Worst-Case Analysis*

Acknowledgements

First and foremost, more than two hundred emails later, I have without a doubt profited greatly from the criticism and suggestions of my supervisor, Professor Peter Damaschke, without whom the completion of this thesis would not have been possible. I would also like to thank my examiner, Richard Johansson, who, together with Peter, showed great patience and repeatedly granted me more time. Finally, I also want to thank all of the folks at the Department of Computer Science and Engineering, who have allowed me to resume writing after months of inactivity.

Johannes Ringmark, Gothenburg, October 2020

Contents

List of Figures **ix**

1 Introduction **5**
1.1 Purpose . . . 5
1.2 Objective . . . 6
1.3 Method . . . 7
1.4 Limitations . . . 7
1.5 Background . . . 7
1.6 Overview . . . 7

2 The Framework **9**

3 The Weighted Interval Scheduling Problem **11**
3.1 Definition . . . 11
3.2 Introducing the Base Algorithm . . . 11
3.3 Designing a Stochastic Algorithm . . . 14
3.4 Sorting and Representation . . . 15
3.5 Taking an Adversary's View on the Problem . . . 18
3.6 Designing a Deterministic Algorithm . . . 22

4 Concluding Remarks **31**

List of Figures

2.1 The internal relations of the framework's concept. The x-axis illustrates the *error*. The y-axis shows the n rounds denoted by r. The line is an example of a possibly desired monotone convergence profile. 10

3.1 The convergence, obtained by an consideration of the heaviest interval already during the first round. The solid line is a convergence asymptote that is continuous up until the last round n. 21

List of Algorithms

1 WIS_1 computes a non-overlapping interval set with the highest profit from progressively bigger subsets of input interval. First, it sorts the interval and fills a memoisation array, then it infers an optimal solution through backtracking. 13

2 WIS_2 computes a non-overlapping interval set with the highest profit from progressively bigger subsets of input interval. It sorts the intervals progressively by using a heap. 19

3 WIS_3 computes a non-overlapping interval set with the highest profit from progressively bigger subsets of input interval. It considers the heaviest interval during its first round. 20

4 WIS_4 computes a non-overlapping interval set with the highest profit. The subroutine h decides the enumeration order. 23

Abbreviations

DP	Dynamic Programming.
FPTAS	Fully Polynomial-Time Approximation Schemes.
WIS	Weighted Interval Scheduling.

Glossary

$\lceil a \rceil$	largest integer not larger than a.
$\lfloor a \rfloor$	smallest integer not smaller than a.
$o(P)$	optimal solution value of problem P.
$\|A\|$	cardinality of set A.

1

Introduction

Precisely at which shape or size large data sets are a computational challenge is ambiguous, since it is in general highly dependent on which operations are to be applied to the data set. Conventionally, the total time of an operation (the so-called competitive *complexity analysis*[1]) has traditionally received the most attention when determining the cumbersomeness of data sets. While this may be fair, it also comes at the expense of other alternatives, such as the structure of the computation process, which consequently have not been prioritised. It is, naturally, essential for the algorithm to arrive at a desirable conclusion and to do so as fast as possible. Still, if the problem instance is sufficiently large, a decision not to consider intermediate results is a decision to postpone the extraction of some knowledge.

Designing for parallel execution causes its own problems. The design and implementation of *concurrent algorithms*, when practised, is widely viewed by the community as an intrinsically difficult and intellectually demanding undertaking. Progressive *combinatorial optimisation* algorithms can help bridge this gap in difficulty.

We will demonstrate that this issue is new and cannot be replaced with any forebearer (i.e. neither with *approximation algorithms*, *heuristics*, nor with *anytime algorithms*), and it can potentially give a clearer picture of how to transition between *approximate solutions*. We conjecture that more control of the execution can be achieved through the extraction of intermediate results of progressively higher quality.

1.1 Purpose

This thesis serves a dual purpose: as a more extensive overview of the subject of *progressive* algorithms and as a source for discussion. Historically, as complexity has come into focus, natural but abstract algorithm-related concepts like the *error* of a solution have not been introduced to the operator in a systematic way during execution, but rather after termination. The handful of contemporary techniques wherein the run-time dynamics have been in focus, such as *meta-heuristics*, *contract algorithms*, and *anytime algorithms*, all fail to guarantee the eventual output of an

[1] Through the entire thesis, italics will be used when introducing context-specific technical terms, and occasionally to resolve any ambiguity. Terms not introduced in this way should consequently henceforth be interpreted as general or more abstract.

optimal solution. This can be seen as a direct result of the fact that they traditionally have received the most attention as pragmatic ways to guarantee arbitrarily good solutions, for particularly complex *differentiable functions* that may not have a *global maximum*.

Through further analysis of this new area, we wish to advance this theoretical direction and extend the present reach through the deduction of solid, relevant holds from which provable results can be reached, thereby benefiting multiple disciplines. We are interested in results which can be deduced through rigorous mathematical reasoning. The problems wherein such results can be helpful presumably require early access and particularly time-consuming solution methods. They are problems such as the construction of large infrastructure, the exhaustive analysis of biological material and organisms, or, given a large enough data set, any *NP-hard* problem. If such an algorithm with several *intermediate* solutions were to exist, for example, for the problem of designing infrastructure, the construction could start once the first solution has been computed. Then depending on unpredictable outer parameters, it could be steered towards the areas with the greatest prosperity. Such iterative methodologies have been used, through interactive identification of desirable parts, to steer indexing and to query [1] [2]. By allowing for quickly retrieval of information, they can significantly increase the performance of applications within economics as demonstrated in [3], and thus substantially increase profit [4].

While the previously-mentioned problems presuppose some foreknowledge and a natural connection between the data at hand and some specific problem, in the opposite case, when the problem is to interpret the data, the *convergence profile* of an *exploring heuristic* can, by comparing it to a larger set, be used to make estimates regarding properties. At present, the *complexity class* of a problem can be looked up for hundreds of common algorithm problems [5], but inferring a time-profit convergence classification of any of those problems requires considerably more effort. Such a categorisation would be useful only if all problems were to use similar definitions of error and profit. A categorisation could also interactively guide heuristics or constitute a starting point for future algorithm design. It may become apparent that no such conclusions can be drawn. The purpose of this thesis is primarily to be exploratory.

1.2 Objective

The overall objective of this exploratory thesis is to improve understanding of how the characteristics of intermediate solutions relate to and impact each other and the final solution. This is done through the computation of provably tight *bounds* for the convergence of over-time change of carefully defined error functions for a textbook problem—the *Weighted Interval Scheduling Problem*—without exceeding the best asymptotic time of its traditional counterparts.

1.3 Method

Through the use of a framework, we apply rigorous mathematical argumentation to analyse the convergence and time complexity of new progressive algorithms that we design by providing alternative enumeration orders to a slightly modified traditional non-progressive base algorithm. We will build the framework on the notations introduced in [6] (intermediate and complete solutions, convergence function, error function), and adopt the framework's definition of progressive algorithms. We also include a brief summary later for the reader's convenience. Furthermore, decisions and the resulting bounds are analysed to identify where further challenges may be sought in the future, and whether any coherent results can be derived.

1.4 Limitations

The scope of this thesis is limited to the study of only one problem. We acknowledge that this problem cannot be thought to represent more than a fraction of all **Dynamic Programming**-compatible problems. The algorithm design focuses on altering the enumeration order of pre-selected existing well-known algorithms which are considered efficient with regards to time complexity. Another algorithm, which might be slightly slower, could have a better convergence profile and dominate ours. A limitation, connected to our definitions, is that our convergence bounds will depend on our definition of profit, simply because profit, although clear in an informal sense, can be defined in multiple ways.

1.5 Background

Recent works on progressive algorithms include the analysis and design of *progressive geometric algorithms* [6] and of a progressive algorithm computing indexes [7]. The progressive geometric algorithms presented include the analysis of array sorting, finding popular places in a set of trajectories, and the convex hull of a planar point set. Presented works on graph-related progressive algorithms include an algorithm for the *Group Steiner Tree* [8]. As shown in [6], progressive algorithms can be derived from sequential repetitive use of approximation algorithms; especially suitable are Fully Polynomial-Time Approximation Schemes, abbreviated FPTAS. Alewijnse has further explored this relationship in [6] and with the result of a theoretical upper bound. Thus, similarities can also be found between progressive algorithms and anytime algorithms, which become especially notable if a progressive algorithm is designed in a *stochastic context*.

1.6 Overview

Initially, early sections 2-3.2 introduces the framework, notations and a traditional algorithm. Section 2 further elaborates on the concept of convergence and error.

Section 3.1 describes the problem, and section 3.2 outlines the base Dynamic Programming algorithm that is a starting point for the later design.

Later chapters present a stochastic enumeration strategy 3.3 and a deterministic enumeration order 3.6. Section 3.4 discusses different aspects of sorting in regards to the combination DP and progressive algorithms, and section 3.5 presents a way of viewing the challenges in the design process as the challenge of a competitor when facing an adversary.

2

The Framework

The framework that will be used for our analysis of progressive algorithms is as follows: we define a problem

$$P := (\mathcal{D},\ error,\ S(\mathcal{D})) \tag{2.1}$$

to be the entity to which the following three concepts relate: the data set $\mathcal{D} = \{d_1 \dots d_n\}$ that forms the problem input, the error function:

$$error : S \in [0, 1]$$

and the solution set $S(\mathcal{D})$ which is the set of all valid solutions $S \subseteq \mathcal{D}$. First and foremost, a solution is a set of elements whereto the error function constitutes a non-negative and problem-specific grading system, and wherein a valid solution is a solution for which a error value exists. The value $error(S)$ is referred to as the error of the solution S. A complete solution is a solution without error: $error(S) = 0$, and an intermediate solution is a solution which an algorithm outputs prior to its complete solution.
An algorithm is a progressive algorithm if the following holds: its error is bounded from above by a convergence function, $f_{conv}(r) \geq error(S_r)$ where

$$f_{conv} : r \in \mathbb{R}^+$$

is defined for *rounds* $r \in \{1, \dots, n\}$, and it has a set of $n-1$ consecutively numbered intermediate solutions $S_1, \dots, S_{n-1}$ and a complete solution S_n.

Intuitively, the relations can be summarised as follows: if an algorithm outputs $n - 1$ valid intermediate solutions, all of which are bounded by its convergence function, before finally producing a solution with zero error, then the algorithm is progressive; the solutions without an error are called complete solutions, and the rest intermediate solutions. Rounds constitute the intervals in between solution outputs. While the rounds are allowed to differ in the amount of time they require, when we discuss the *round time* of a progressive algorithm, we refer to the time of the round requiring the longest time. For a convergence function to be valid in this context, it must be *monotone*, namely of decreasing error; ergo,

$$f_{conv}(r) \geq f_{conv}(r + 1)$$

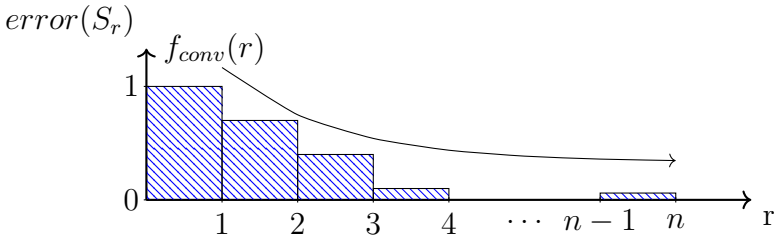

Figure 2.1: The internal relations of the framework's concept. The x-axis illustrates the *error*. The y-axis shows the n rounds denoted by r. The line is an example of a possibly desired monotone convergence profile.

for all rounds $r \in \{1, \ldots, n\}$.

The time complexity of a progressive algorithm is conventionally computed asymptotically and represented by $T(n)$ where n denotes the input size.

Approximation algorithms are algorithms that, instead of computing the optimal solution, calculate an approximate solution. In this context, we define an *α-approximation algorithm* as an algorithm which given a data set $\mathcal{D}$ computes a solution S_a to a problem P such that S_a is *compatible* with an error function *error* such that

$$\alpha \geq error(S_a)$$

where α is a non-negative number.

3

The Weighted Interval Scheduling Problem

With the framework defined, we turn our attention to the second part wherein our problems are tested for progressive features. We achieve this by choosing an algorithm as a starting point and then analysing its different enumeration strategies by adopting a competitive view of convergence.

The WIS problem is a promising candidate for two reasons. It can arguably be regarded as efficiently solved to optimality, and its total time requirement is naturally strongly limited, as is the time that a heuristic can be spent on each round. This is a clear limitation on creativity, because all techniques that require time approaching that of the best-known traditional solution must also be partitionable into smaller problems, each of which contributes to the convergence.

3.1 Definition

Weighted interval scheduling, the problem P at hand, can be loosely formulated as the problem of picking the subset of non-overlapping weighted intervals $S \subseteq \mathcal{D}$ from a predefined input set $\mathcal{D} = \{I_1 \dots I_n\}$ of n intervals $I_i = (s_i, f_i, p_i > 0)$ that maximises $\sum_{I_j \in \mathcal{D}} p(I_j)$ where $1 \leq i \leq n$ and $p : I_i \in \mathbb{R}^+$. By overlap-free we mean that for any two intervals $I_j := (s_j, f_j)$ and $I_{j+\alpha} := (s_{j+\alpha}, f_{j+\alpha})$, such that $f_j \leq f_{j+\alpha}$ it holds that $s_{j+\alpha} \geq f_j$. We define, in accordance with definition (2.1), the problem P by defining its remaining constituent, the error function, as follows,

$$error_{WIS}(S) := 1 - \frac{\sum_{\hat{I} \in S} p(\hat{I})}{\sum_{I \in WIS(\mathcal{D})} p(I)}$$

where $WIS(\mathcal{D})$ is an arbitrary optimal solution from $S(\mathcal{D})$ and where S must be overlap-free. We have chosen the name "profit" over weight as affording a reoccurring reminder that it is the profit that is maximised.

3.2 Introducing the Base Algorithm

The algorithm we will use when designing our own requires $\mathcal{O}(n \log n)$ time and is based on that in [9]. The algorithm consists of two steps: firstly, all intervals are

sorted on increasing right endpoints, using $\mathcal{O}(n \log n)$ time; secondly, an iterative backtracking routine is used to compute the solution in $\mathcal{O}(n)$ time.

We previously defined our optimal solution to be the set $WIS(\mathcal{D})$. We now continue and let $OPT(j)$ denote the optimal profit of the subset from $\mathcal{D}$ containing the j first intervals enumerated on increasing right endpoints where $0 \leq j \leq n$. Using this definition of $OPT(j)$ and a renowned *DP* recurrence [9], the definition of the iterative routine OPT can be formulated as follows,

$$OPT(j) = \max(OPT(j-1),\ OPT(f_{next}(j-1)) + p(i_j)) \tag{3.1}$$

$$OPT(0) = 0$$

where $j \geq 1$, and $f_{next}(j)$ is the index of I_j's nearest preceding non-overlapping interval.

By using the recurrences as a mean of decomposition, the problem can be viewed recursively as sets of subproblems whereto *DP* can be applied to compute the optimum sum, the optimal set of intervals, or both–depending on one's objective. In general, in order for *DP* to be applicable, these *subproblems* must be such that they build up to larger and larger subproblems and finally to the optimal solution. In our case, this property follows directly from the recurrence (3.1).

The optimum sum can be computed by applying addition and the max function to the recurrence accordingly; thus, the problem is compatible with the *DP* approach. The correctness of the solution follows directly from the recurrence, but the effectiveness and complexity remain dependent on the implementation and the computation model.

The algorithm utilises two *memoisation* arrays, in order to prevent redundant computations. After firstly sorting the intervals, it uses array M to store the solution corresponding to each subproblem $OPT(j)$ for $j < n$, and array N to store the values of f_{next} such that when the arrays have been filled

$$OPT(j) := M[j]$$

and

$$f_{next}(j) := N[j]$$

for $j \in \{1, \ldots, n-1\}$. Once the arrays are filled, the algorithm computes the optimal solution $WIS(D)$ by tracking the recursion's choices backwards through array M starting from $M[n]$. A rough implementation describing this is defined in pseudocode in *Algorithm 1*. Because the proposed algorithm already uses sorting and thus $\mathcal{O}(n\ \log n)$ time, we settle with binary search as the procedure for computing all previous non-overlapping intervals in $\mathcal{O}(n\ \log n)$, while allowing that there are better and faster approaches.

Algorithm 1 WIS_1 computes a non-overlapping interval set with the highest profit from progressively bigger subsets of input interval. First, it sorts the interval and fills a memoisation array, then it infers an optimal solution through backtracking.

```
1: procedure WIS_1(Array intervals)
2:     int n = size(intervals)
3:     int m = ⌊n / log n⌋
4:     Array M = instantiateMemoisationArray()
5:     intervals = sortOnIncEnd(intervals);
6:     Array copiedIntervals = newArray();
7:     for r := 1, 2, ..., ⌈log(n)⌉ do
8:         for p := m(r − 1), ..., m(r − 1) + m do
9:             copiedIntervals = addTo(intervals[p], copiedIntervals);
10:        end for
11:        for p := m(r − 1), ..., size(copiedIntervals) do
12:            Integer next = f_next(p, copiedIntervals);
13:            Integer pick = profit(copiedIntervals[p]) + M[next]
14:            Integer ignore = M[p]
15:            M = addTo(max(pick, ignore), M)
16:        end for
17:        S = backtrack(copiedIntervals, M)
18:        Output S
19:    end for
20: end procedure
21:
22: function instantiateMemoisationArray
23:     Array temp = newArray()
24:     temp = addTo(0, temp)
25:     return temp
26: end function
27:
28: function backtrack(Array copiedIntervals, Array M)
29:     Integer sum = 0
30:     Array solution = newArray()
31:     Integer i = size(M) − 1
32:     while i > 0 do
33:         if m[i] > m[i − 1] then
34:             sum = sum + profit(copiedIntervals[i − 1])
35:             solution = addTo(copiedIntervals[i − 1], solution)
36:             i = f_next(i − 1, copiedIntervals) + 1;
37:         else
38:             i = i − 1
39:         end if
40:     end while
41:     return solution
42: end function
```

Compared to [9], our minor modifications alter the execution slightly, but they do not alter the underlying approach. The differences in this algorithm's execution compared to [9] are in the way the elements are selected and how sets of elements are processed. The algorithm that we propose selects a subset of elements (see Algorithm 1, rows 8-10) in rounds (rows 7) and runs the DP procedure (rows 11-16) and the backtracking (rows 17, 28-42) on these elements exclusively, before it addresses the other elements. Instead of filling all the tables completely with all the elements, executing the entire recurrence once as in [9], it computes the tables by considering more and more intervals gradually in iterations. For each iteration, a new subset of intervals is added to the working set, which is then sorted and processed. If the considered subsets are externally sorted, the memoisation arrays can be continuously extended, without any re-computations. Through a worst-case analysis of *enumeration strategies*—ways to construct these interval subsets— we determine if the convergence can be improved.

3.3 Designing a Stochastic Algorithm

By combining the previously-explained algorithm routine with a randomised enumeration strategy, we obtain a randomised algorithm that achieves an expected linear growth of profit and thus also an expected linear decay of error. For this stochastic approach, we choose to call an algorithm *pseudo-progressive* if it is a progressive algorithm with respect to the error function (3.2)

$$error_{PWIS}(S) := 1 - \frac{\mathbb{E}[\mathbf{Z}_m]}{\sum_{i \in I} p(i)} \tag{3.2}$$

where $\mathbf{Z}_m$ is a *discrete random variable* denoting the sum of profits from selecting a number $m = |S|$ of intervals from the interval population of size n using some predefined strategy, and where $\mathbb{E}[\mathbf{Z}_m]$ denotes the expected profit-sum.

Theorem 1. *There is a stochastic pseudo-progressive algorithm for the Weighted Interval Scheduling Problem that runs in $\lceil n \log n \rceil$ rounds, denoted by indices r, in $\mathcal{O}(n \log n)$ time with a round time of $\mathcal{O}(n)$. This algorithm obtains an expected linear decay in the error computed through $error_{PWIS}$.*

Proof. We pick m intervals ($0 \leq m \leq n$) at random, uniformly, without replacement. Let $C_j \in \{0, 1\}$ be an indicator random variable that takes the value 1 if the interval with index j was selected and 0 if that same interval was not selected. The random variable we are investigating is

$$\mathbf{Y} = \frac{\sum_{j=1}^{n} C_j x_j p(I_j)}{\sum_{j=1}^{n} x_j p(I_j)} \tag{3.3}$$

where $x_j \in \{0, 1\}$ is 1 if the interval is in some optimal solution $S = WIS(\mathcal{D})$ and 0 otherwise. Here $\mathbf{Y}$ represents the ratio of the profit sum of the selected intervals that are also in the optimal solution S, and all the intervals in the optimal solution

S. The probability $P[C_i = 1]$ is always m/n. Consequently, we have the relation in (3.4).

$$\mathbb{E}[C_i] = \frac{m}{n} \tag{3.4}$$

The expected value of $\mathbf{Y}$ can then be expressed as in (3.5) using the relation in (3.4).

$$\mathbb{E}[\mathbf{Y}] = \frac{\sum_{j=1}^{n} \mathbb{E}[C_j] x_j p(I_j)}{\sum_{j=1}^{n} x_j p(I_j)} = \left(\frac{m}{n}\right) \frac{\sum_{j=1}^{m} x_j p(I_j)}{\sum_{j=1}^{m} x_j p(I_j)} = \frac{m}{n} \tag{3.5}$$

Moreover, if m is set so that $m := r\lceil n/\log n \rceil$ for $r \in \{1, \dots \log n\}$, then we obtain the convergence function in (3.6) which constitutes an upper bound on the error.

$$f_{conv}(r) = 1 - r\left(\lceil \frac{n}{\log n} \rceil / n\right) \geq 1 - \frac{r}{\log n} \tag{3.6}$$

This proves that for $r \in \{1, \dots, \log n\}$, selecting a set of m intervals uniformly at random and feeding them in progressively bigger sets repeatedly to *Algorithm 1*, one achieves an linearly increasing ratio, and consequently also a linear decrease in the error bound by (3.6). □

In the rare case where the heaviest interval already constitutes a bigger portion of the sum of intervals than m/n, then the expected linear convergence in (3.6) might be marginally improved upon by incorporating a preprocessing search routine that determines the heaviest interval. The error function, while giving an expected convergence, does in the worst-case scenario not guarantee any profit; thus, without such a routine, the expected numbers of selections before finding the heaviest interval is $n/2$. The profit of the heaviest interval, which one could guarantee in the first round, can then not be guaranteed until much later.

3.4 Sorting and Representation

Achieving a linear convergence is arguably straightforward if one is prepared to sacrifice parts of, or loosen, the definition and consider the error stochastically. However, achieving a good convergence for a deterministic algorithm can be rather arduous. In the previous sections on algorithm design, we attempted to target and analyse very specific implementation parts. The following section takes a broader view. We discuss the sorting and the representation intuitively, as if viewed from afar, this time without focusing on implementation-related details. One major bottleneck, sorting, was identified in the previous section. Here we continue that pursuit of pinpointing bottlenecks.

Initially, we look for signs of bottlenecks by turning our attention to the weight-free Interval Scheduling problem for which all weights can be regarded as equal and therefore disregarded. This weight-free variant can be solved even by a greedy algorithm; however, even then, though it appears simpler, the greedy approach is still

as dependent on the intervals being sorted as the Dynamic Programming approach with weights.

As mentioned, sorting specifically constitutes an obvious bottleneck and requires a great deal of time. However, conceptually it is necessary to reify the representation of the overlaps, without which no solution can be verified. Even without having to consider weights, and consequently without having to perform any intermediate re-sorting to satisfy different subroutines' ordering criteria, one may conjecture that building a full representation of such intervals' overlaps requires $\mathcal{O}(n \log n)$ time in the worst-case scenario. In conclusion, by analysing the role sorting plays, we conclude that the limitation lies in our ability to construct a better representation of the overlaps, and in turn, in our ability to more quickly verify the lack of overlaps in a potential solution. Therefore, this limitation applies to the weighted variant as well. In light of these limitations, we proceed in the thesis with sorting-based algorithms and disregard alternatives.

Through two lemmas, we demonstrate a new way in which progressive sorting can be designed using $\log n$ rounds, such that for each elapsed round $\lceil n/\log n \rceil$ additional elements are sorted. This is achieved by utilising a well-known heap-sort algorithm, iteratively in rounds. This sorting algorithm will guarantee that greater numbers are sorted and presented before lesser. Another more general progressive sorting algorithm that guarantees that all elements are sorted, regardless of which $\lceil n/\log n \rceil$ elements are selected in each round, is presented in [6].

Lemma 1. *There is a well-known algorithm that, given a heap filled with n natural numbers (that is, so that the number in every node is larger than the numbers of its children) can extract the m greatest numbers ($1 \leq m \leq n$) in sorted order, using $\mathcal{O}(m \log n)$ time.*

Proof. Given a balanced and filled heap, the greatest or, alternatively, the smallest numbers can be iteratively retrieved in $\mathcal{O}(\log n)$ time each [10]. Once the set has been filled and ordered, this quality allows for the extraction of the m greatest numbers in sorted order using only $\mathcal{O}(m \log n)$ time. □

Lemma 2. *There is a progressive sorting algorithm that, given a set of natural numbers, can by sorting* $m := \lceil n/\log n \rceil$ *numbers in each round using* $\mathcal{O}(n)$ *time, guarantee two things: continuously, after the* r^{th} *round it has sorted the* mr *largest numbers, where* $1 \leq r \leq \log n$*. Then after the* $\log n^{th}$ *round it has sorted all* n *numbers using* $\mathcal{O}(n \ \log n)$ *time.*

Proof. It follows from Lemma 1 that once a balanced heap has been constructed, all n items can be sorted iteratively by extracting $m := \lceil n/\log n \rceil$ items at the time for $r \in \{0, \ldots, \log n\}$ rounds. For each round, the heap data structure by design enforces a relationship between the items. This structure ensures that if elements are popped from the heap one-by-one, then any number extracted earlier will be greater to any number extracted later. Consequently, after $\log n$ rounds of repeatedly extracting m numbers, then numbers are presented in sorted order. The construction of a heap requires $\mathcal{O}(n)$ time [10], and the sorting, as demonstrated above, requires $\mathcal{O}(\log \ n)$ time, thus the total time of this sorting algorithm is $\mathcal{O}(n \ \log \ n)$. □

3.5 Taking an Adversary's View on the Problem

It is clear that a sorting routine exists that allows for intermediate extraction. However, while facilitating a round-based approach, as the routine currently does, it neither impacts the convergence profile, nor provides a valid solution with regards to the error function. Consequently, it is not much use in isolation. However, by using the routine to sequentially provide the parent-algorithm with new internally sorted interval groups whenever such are available, we have *Algorithm 2* which, unlike *Algorithm 1* presented earlier, is a progressive algorithm.

The proposed *Algorithm* 2 shares the inner mechanism of *Algorithm* 1 with one exception: instead of sorting the entire interval set once, it utilises a heap to distribute the sorting asymptotically-evenly among all rounds. Adopting the progressive sorting from Lemma 2 now allows the algorithm for each round to sequentially extract m new sorted intervals, compute the solution, and extend the memoisation and backtrack table, all in asymptotically $O(n)$ time.

To analyse more easily which limits are associated with the convergence, we can imagine a game with two players, denoted as player and adversary, respectively. In this game, the player starts by selecting a deterministic group-enumeration algorithm. Then the adversary must provide as bad an input set as possible in an attempt to force the worst possible convergence.

Assume the player chose either of *Algorithm* 1 or 2, then we assert that, consequently, the adversary can design an effective intervals set A by letting it fulfil two criteria. Firstly, A has to contain a subset of exactly $m := \lceil n/\log n \rceil$ non-overlapping intervals B with a sum that is γ times heavier than the sum of the complementing set $A - B$, that is

$$\sum_{i \in B} p(i) = \gamma \sum_{j \in A-B} p(j) \tag{3.7}$$

Secondly, all the intervals in B have to have all the necessary characteristics needed for them to be the last ones processed. For *Algorithm* 1 or 2, since they process intervals on their decreasing right endpoint, the intervals in B can be guaranteed to be evaluated last simply by letting $f_k < f_l$ for $0 \leq k < n - m$ and $n - m \leq l \leq n$.

Adopting such a strategy would prove effective for the adversary, who then can allow γ to approach ∞ to reach the worst-case convergence. Then, no decrease in error is achieved before the last round. In conclusion, the algorithm considered up to this point has a significant weakness as a progressive algorithm. It does not consider the profits of the intervals with regard to the convergence, and it relies on a fixed deterministic enumeration order.

When examining methods for incorporating weight relations, we find that the player can possibly improve the bound slightly by adding to *Algorithm* 2 an initial preprocessing procedure wherein the heaviest interval is found automatically

and included in the set of intervals that will be sorted in the first sorting round. A lower bound on the profit sum after the first round is thus the profit of the heaviest interval a_h by itself.

Algorithm 2 WIS_2 computes a non-overlapping interval set with the highest profit from progressively bigger subsets of input interval. It sorts the intervals progressively by using a heap.

```
1:  procedure WIS_2(Array intervals)
2:      Integer n = size(intervals)
3:      Integer m = ⌊n / log n⌋
4:      Array M = instantiateMemoisationArray()
5:      Heap heap = instantiateHeap(intervals);
6:      Array copiedIntervals = newArray();
7:      for r := 1, 2, ..., ⌈log n⌉ do
8:          for p := m(r − 1), ..., m(r − 1) + m do
9:              copiedIntervals = addTo(pop(heap), copiedIntervals);
10:         end for
11:         for p := m(r − 1), ..., size(copiedIntervals) do
12:             Integer next = f_next(p, copiedIntervals);
13:             Integer pick = profit(copiedIntervals[p]) + M[next]
14:             Integer ignore = M[p]
15:             M = addTo(max(pick, ignore), M)
16:         end for
17:         S = backtrack(copiedIntervals, M)
18:         Output S
19:     end for
20: end procedure
```

Moreover, the worst possible convergence can still be obtained by the adversary using the same approach as in (3.7). The same convergence holds even if we design the set of intervals to contain a subset B consisting of $\lceil n/\log n \rceil$ non-overlapping intervals with a combined profit of γ times the profit of its complement.

Notably, γ is now not only bounded from below but also from above (3.8).

$$\sum_{i \in B} p(i) = \gamma \sum_{j \in A-B} p(j), \quad 1/n < \gamma \leq \lceil n/\log n \rceil \tag{3.8}$$

We create *Algorithm 3* by first extending *Algorithm 2* with a preprocessing routine (see *Algorithm* 3, rows 5-6) and then by moving the sorting step (see Algorithm 2, row 14).

Algorithm 3 WIS_3 computes a non-overlapping interval set with the highest profit from progressively bigger subsets of input interval. It considers the heaviest interval during its first round.

```
1: procedure WIS_3(Array intervals)
2:     Integer n = size(intervals)
3:     Integer m = ⌊n / log n⌋
4:     Array M = instantiateMemoisationArray()
5:     Interval heaviestInterval = getHeaviestInterval(intervals)
6:     intervals = removeFrom(heaviestInterval, intervals)
7:     Heap heap = instantiateHeap(intervals);
8:     Array copiedIntervals = newArray();
9:     copiedIntervals = addTo(heaviestInterval, copiedIntervals);
10:    for r := 1, 2, ..., ⌈log n⌉ do
11:        for p := m(r − 1), ..., m(r − 1) + m do
12:            copiedIntervals = addTo(pop(heap), copiedIntervals);
13:        end for
14:        copiedIntervals = sortOnIncEnd(intervals);
15:        for p := m(r − 1), ..., size(copiedIntevals) do
16:            Integer next = f_next(p, copiedIntervals);
17:            Integer pick = profit(copiedIntervals[p]) + M[next]
18:            Integer ignore = M[p]
19:            M = addTo(max(pick, ignore), M)
20:        end for
21:        S = backtrack(copiedIntervals, M)
22:        Output S
23:    end for
24: end procedure
```

Theorem 2. *There is a progressive algorithm for the Weighted Interval Scheduling Problem requiring* $\log n$ *rounds, denoted by indices* r*, in* $\mathcal{O}(n \log n)$ *time with a round time of* $\mathcal{O}(n)$*. This algorithm results in a convergence bounded by the function* $f_{conv}(r) := 1 - \frac{\log n}{r} \times \frac{\log n}{n+2\log n}$ *with respect to the error function* $error_{WIS}(S_r)$*.*

Proof. The recurrence and the backtracking (see, *Algorithm* 3, rows 16-21) of *Algorithm* 3 remain identical to those of *Algorithm* 2. The correctness of the solutions then follows.

The additional computational time added, apart from the $\mathcal{O}(n)$ needed for the preprocessing, are $\mathcal{O}(\log\ n)$ time to insert the heaviest interval into the sorted working set (see *Algorithm* 3, row 9) and $\mathcal{O}(rn/\log n)$ to recompute the memoisation array. The asymptotic time will remain the same as for *Algorithm* 1.

An upper bound for the error can be computed for *Algorithm* 3 through the following series of algebraic manipulations:

$$\begin{aligned} error_{WIS}(S_r) &= 1 - \frac{\sum_{i \in S} p(i)}{\sum_{j \in o(D)} p(j)} \\ &\leq 1 - \frac{p(a_h)}{(1 + \lceil \frac{n}{\log n} \rceil) p(a_h)} \\ &\leq 1 - \frac{\log n}{n + 2 \log n} \end{aligned}$$

As $f_{conv}(r)$ must be greater than $error_{WIS}(S_r)$ for all $r \in \{1, \ldots, \log n\}$, it follows that (3.9) is a valid convergence function for this algorithm with regard to this error function.

$$f_{conv}(r) := 1 - \frac{\log n}{r} \cdot \frac{\log n}{n + 2 \log n} \tag{3.9}$$

This is illustrated in Figure 3.1 below. □

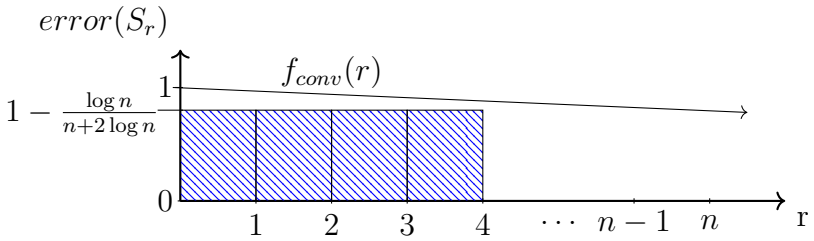

Figure 3.1: The convergence, obtained by an consideration of the heaviest interval already during the first round. The solid line is a convergence asymptote that is continuous up until the last round n.

However, for a large n, the decrease in error obtained from finding $p(a_h)$ approaches zero

$$\lim_{n \to \infty} \frac{\log n}{n + 2 \log n} = 0$$

When the convergence is slightly improved, as we demonstrated by letting $n \to \infty$, the adversary can still rather effortlessly construct an interval set whereby this worst-case convergence can be achieved.

In this section, we have argued that if the traditional algorithm is provided with the entire set of intervals, on the one hand, we have the upside that the algorithm only needs to sort the intervals once. On the other hand, the downside is that the knowledge of the convergence will then be limited. We have also argued that if the heaviest interval is removed from the interval set before the sorting and the algorithm is modified to output either it or the intermediate solution, we can guarantee the convergence given by (3.9). As long as the heaviest interval can be kept separated from intermediate solutions, the memoisation array can be extended and does not

need to be recomputed for each round. However, after the interval eventually is incorporated the memoisation must be recomputed for each round.

The conclusion is that providing the algorithm with subsets in any other order than that described above likely comes with significant computational overhead. This overhead arises from the associated necessary interval re-sorting and from the associated recurrent need to entirely recompute the memoisation array.

While the previous section has focused on this drawback and argued for a group enumeration approach, enumeration with the aim of improving the bound remains unexplored. Therefore, in the following section, we turn our attention away from the search for improvements through preprocessing and continue by weighing the advantages of incorporating a set of selected enumeration strategies.

3.6 Designing a Deterministic Algorithm

In this section, we will present a second deterministic algorithm which relies on a combination of sorting and partitioning strategies. The resulting algorithm's convergence function dominates $f_{conv}(r) \leq f^*(r)$ where $f^*(r)$ denotes the convergence function

$$f^*(r) := 1 - (\frac{\log n}{r}) \cdot (1 + \lceil \frac{n}{\log n} \rceil)^{-1}$$

for $0 \leq r < \log n$.

We let h denote a one-to-one mapping that maps an unordered set of intervals labelled $x_i, \ldots, x_n$ to an ordered set consisting of the same intervals. We will refer to the order of the sorted set as the enumeration order and the function h as the enumeration strategy.

Moreover, we limited our set of strategies and considered only "compound-attribute" strategies to be of interest. By the compound adjective "compound-attribute," in contrast to "single-attribute," we mean orderings based on several attributes, and not those that are based on a ratio of attributes or based on only one attribute, such as left endpoint, right endpoint, weight (individually, or as a sum), or overlaps. For example, a compound order could be an ordering in which for every even turn the interval is chosen e.g. based on its left endpoint, and for every odd turn, the interval is chosen e.g. based on its weight.

From *Algorithm* 3, we create *Algorithm* 4 by replacing the heap-pop set extractions with a new step which instead culls from a predefined enumeration mapping (see *Algorithm* 4, row 8). When considering strategies, we ignore all enumerations strategies that utilise only one attribute.

Algorithm 4 WIS_4 computes a non-overlapping interval set with the highest profit. The subroutine h decides the enumeration order.

```
 1: procedure WIS_4(Array intervals)
 2:     Integer n = size(intervals)
 3:     Integer m = ⌊n / log n⌋
 4:     Array M = instantiateMemoisationArray()
 5:     Array copiedIntervals = newArray();
 6:     Array pivots = newArray();
 7:     for r := 1, 2, ..., ⌈log n⌉ do
 8:         for p := m(r − 1), ..., m(r − 1) + m do
 9:             copiedIntervals = addTo(h(copiedIntervals, pivots, n), copiedIntervals);
10:         end for
11:         copiedIntervals = sortOnIncEnd(copiedIntervals);
12:         for p := 1, ..., size(copiedIntevals) do
13:             Integer next = f_next(p, copiedIntervals);
14:             Integer pick = profit(copiedIntervals[p]) + M[next]
15:             Integer ignore = M[p]
16:             M = addTo(max(pick, ignore), M)
17:         end for
18:         S = backtrack(copiedIntervals, M)
19:         Output S
20:     end for
21: end procedure
```

In the remainder of this section, we investigate a partitioning approach. We will compute the time complexity for a promising breadth-first heuristic h_1 whereby sets of intervals are iteratively extracted while being partitioned into $\lceil n \log n \rceil$ externally-sorted subsets.

For the first round, the heuristic h_1 starts with one set containing all intervals. We refer to this set as its working set. The heuristic proceeds to use a median-of-means algorithm to compute a pivot interval which it then uses as a *divider* to create two new sets. Intervals with increasing right endpoints greater than that of the pivot are channelled into the first set, and those that are smaller into the second set. For proceeding iterations of the above steps, larger sets are divided before smaller and ties are resolved randomly.

By continuing to find, divide, and channel the intervals in this manner, the heuristic eventually ends up with at least $n \log n$ subsets that are externally sorted. There is, however, no guarantee of them being internally sorted. This is when the dividing part stops. From the first division iteration until it has reached its desired number of subsets, the heuristic essentially does two things in each round: in the first step, it performs as many division iterations as it can fit within the given time window; in the second, it composes and outputs intermediate results by strategically selecting without replacement elements based on multiple criteria. After it has the desired number of subsets, later rounds will omit the division step. Finally, these repeated

removals of intervals result in the heuristic having no more intervals to process. The heuristic then terminates.

Through Lemma 3, we argue that one of the intervals that should be selected each time from each set is the interval with the rightmost left endpoint. Through Lemma 4, we argue that another interval is the heaviest. Lemma 5 deals with the fact that the subsets may differ in size. Selecting equally as many intervals from each set will thus lead to some sets being depleted before others. These are then combined in Theorem 3 to assess the limitations of h_1 when used as the enumeration strategy of *Algorithm* 4. During later rounds, this suggested approach will provide a relationship between overlaps and the optimal sum.

```
1: procedure h1(List intervals, List pivots, Integer n)
2:     Integer m = ⌊n / log n⌋
3:     if size(pivots) < m - 1 then
4:         List newPivots = addTo(computePivots(intervals, pivots), newPivots);
5:         intervals = addTo(partition(intervals, pivots, newPivots), intervals);
6:         pivots = addTo(newPivots, pivots);
7:     end if
8:     List roundResult = popIntervals(intervals, pivots)
9:     Output (pivots, roundResult)
10: end procedure
```

Definition 1. A series of m sets $B_1, \ldots, B_m$ of intervals is **externally sorted** on increasing right endpoints, if all intervals a_j have in every later set B_j a right endpoint f_{a_j} that is left of or equal to the right endpoints in a later set B_k where $1 \leq j < k \leq m$.

Lemma 3. *Let $B_1, \ldots, B_m$ be a series of m sets externally ordered by increasing right endpoints and $I = \{i_1, \ldots, i_m\}$ be a set containing exactly m non-overlapping intervals, such that $\{i_j\} = I \cap B_j$ for each set B_j where $1 \leq j \leq m$. Then a set $R = \{r_1, \ldots, r_{\lceil m/2 \rceil}\}$ consisting of the intervals with the rightmost left endpoints in sets with an odd index B_{2k-1}, such that $\{r_k\} = R \cap B_{2k-1}$ for $1 \leq k \leq m/2$, constitutes a set of mutually non-overlapping intervals with cardinality $\lceil m/2 \rceil$.*

Proof. Let r_j and i_j denote the interval in B_j with the rightmost left endpoint, and the interval from I found in both sets $i_j \in I \cap B_j$, respectively. Because r_j has the rightmost left endpoint of the intervals in its set, it follows that the left endpoint of r_j is right of or equal to that of i_j:

$$s_{i_j} \leq s_{r_j} \tag{3.10}$$

Because the intervals in I are mutually non-overlapping and the sets are externally ordered by increasing right endpoint, the right endpoint of i_j is left of the left endpoint of i_{j+1}, given that $j < m$:

$$f_{i_j} < s_{i_{j+1}}, \quad 1 \leq j < m \tag{3.11}$$

The fact that the sets are externally ordered by increasing right endpoint provides the following relationship, given that $j < m$:

$$f_{r_j} \leq f_{i_{j+1}}, \quad 1 \leq j < m \tag{3.12}$$

Moreover, the inequalities (3.11, 3.12, 3.13) together form the following relationships:

$$f_{r_j} \leq f_{i_{j+1}} < s_{i_{j+2}} \leq s_{r_{j+2}}, \quad 1 \leq j < m-1$$

In conclusion, by unfolding the recurrence $f_{r_j} < s_{r_{j+2}}$, it follows that the resulting set consisting of all r_k for odd indices $k = 2l-1 \in \{l \in \mathbb{Z} : 1 \leq l \leq m/2\}$ constitutes a overlap-free set with cardinality $\lceil m/2 \rceil$. □

Lemma 4. *Let S_r be independent sets containing $\lceil n/m \rceil$ weighted intervals constructed by collecting exactly one interval from each of m externally ordered sets $B_1, \ldots, B_m$, in m rounds denoted by indices r, and let the heaviest interval a be one of the intervals picked in the first round $a \in S_1$. Then the ratio between the sum of the interval weights from an overlap-free subsets denoted C where $C \subseteq S_m$, and the sum of the interval weights from an overlap-free subsets denoted D, where $D \subseteq \bigcup_{r=0}^{m-1} S_r$, is bounded from above by $|C|$.*

Proof. The intervals in C will have a weight sum within the range (3.13).

$$0 < \sum_{i \in C} p(i) \leq |C| p(a) \tag{3.13}$$

The profit of the overlap-free set D is, consequently, at the last round greater or equal to $p(a)$ and less than $r|C|p(a)$.

$$p(a) \leq \sum_{i \in D} p(i) \leq r|C|p(a) \tag{3.14}$$

These relations (3.13, 3.14) give the following *upper* and *lower* bounds on the weight ratio

$$0 < \frac{\sum_{i \in C} p(i)}{\sum_{j \in D} p(j)} \leq |C| \leq \lceil \frac{n}{m} \rceil$$

□

Lemma 5. *There is an interval partitioning algorithm that can in $\lceil \log(m) \rceil$ rounds, with round time $\mathcal{O}(n)$, partition n intervals into m sets $B_1, \ldots, B_m$ externally ordered by increasing right endpoints, such that each set has cardinality $|B_i| \geq \lfloor n/m \rfloor$ where $1 \leq i \leq m \leq \log(n)$.*

Proof. There exist several selection algorithms that can find the k:th smallest interval from an unsorted list in $\mathcal{O}(n)$ time [11] [12] [10]. Let $D_r = \{\lfloor j \frac{n}{2^r} \rfloor : 0 < j < 2^r - 1, j \in \mathbb{Z}\}$ be an ordered set of divider indices whereby n intervals are partitioned recursively, such that after each round it holds for all divider indices $d_j \in D_r$ and intervals $a_1 \ldots a_n$ that

$$f_{a_{d_j - \beta}} \leq f_{a_{d_j}} \leq f_{a_{d_j + \alpha}} \tag{3.15}$$

where $0 < \alpha \leq n - d_j$ and $0 \leq \beta < d_j$. For each round r, computing new divider indices, finding the mean a selection algorithm, and enforcing (3.15) require a total of $\mathcal{O}(n)$ time. The number of rounds r_1 needed before all divisor distances $d_j - d_{j-1}$ are less or equal to $\lfloor n/m \rfloor$ can be solved for from the following equation $2^r \geq m$ which gives

$$r_1 = \lceil \log(m) \rceil \tag{3.16}$$

After $r = \lceil \log(m) \rceil$ rounds requiring $\mathcal{O}(n \log m)$ time, let $E = \{\lfloor j \frac{n}{m} \rfloor : 0 < j < m, j \in \mathbb{Z}\} \cap \{1, n\}$ be a new ordered set of new divisor indices $e_1 \dots e_m$. Let b_j be the intervals with e_j:th rightmost endpoint computed by applying the k:th selection algorithm individually on each interval set $\{a_{d_i} \dots a_{d_{i+1}}\}$ constructed from the divisor ranges $(d_i, d_{i+1}), 1 \leq i \leq m-1$. Re-partition the intervals for each divisor $e \in E$ and direct enclosing divider indices (d_i, d_{i+1}) that fulfils $d_i < e < d_{i+1}$ such that

$$f_{b_{e-\beta}} \leq f_{b_e} \leq f_{b_{e+\alpha}}$$

where $0 \leq \alpha \leq d_{j+1} - e$ and $0 \leq \beta \leq e - d_j$. Finally m sets $B_1 \dots B_m$ can be created from the ranges constituted by the indices $e \in E$. □

Lemma 6. *Let $B_1, \dots, B_m$ be weighted interval sets externally ordered by increasing right endpoints. Then there is a deterministic progressive algorithm that is using m rounds, denoted by indices r, with round time $\mathcal{O}(n)$, and with a convergence function $f_{conv}(r) < f^*(r)$ with respect to the error function $error_{WIS}(S_r)$.*

Proof. Let C_j^d denote the set of intervals whose left endpoint s_a is left of the right endpoint f_b of the interval b with the rightmost endpoint from the d:th preceding set B_j such that $s_a \leq f_b$ where $a \in B_j$, $b \in B_{j-d}$, $d+1 \leq j \leq m$ and $1 \leq d < m-1$. The upper bound u_1 on the sum of interval profits of a non-overlapping interval set from B_j is given by $|B_j|p(a)$, where a is the heaviest intervals of all sets $B_1, \dots, B_m$. The upper bound u_2 on the sum from C_j^d, however, is given by $p(a)$ because each interval $a \in C_j^d$, by definition overlaps with all other intervals in C_j^d at f_b. The upper bound $u : S \in \mathbb{R}^+$ on the union of all sets B_j is thus given by

$$u(\bigcup_{i=1}^{m} B_i) = p(a) \sum_{i=1}^{m} |B_i| \tag{3.17}$$

for $1 \leq j \leq m$, whereas the following upper bound on the union of all sets C_j^d ,

$$u(\bigcup_{i=1}^{m} C_i^d) = p(a)\frac{m}{d}, \quad 2 \leq d \leq m \tag{3.18}$$

is significantly lower because each interval picked from a set C_j^d by definition overlaps with all intervals in $d-1$ preceding sets, and thus consequently cannot constitute a non-overlapping set with any of these. It follows from Lemma 4 that, as long as the heaviest interval is found during the first round, the worst possible convergence for $B_1, ..., B_m$ at the penultimate round is bounded from above by

$$f_{conv}(r) \leq 1 - \frac{p(a)}{|B_j|p(a)} = 1 - \frac{1}{|B_j|}$$

The worst possible convergence for $C_1^d, ..., C_m^d$, on the other side, is significantly lower, as it is bounded from above by

$$f_{conv}(r) \leq 1 - \frac{p(a)}{p(a)\frac{m}{d}} = 1 - \frac{d}{m}, \quad 2 \leq d \leq m$$

To define each round's output set S_r, let $\alpha = \lceil n/2m^2 \rceil$ represent half of the number of intervals that needs to be collected from each set $B_1, ..., B_m$ to collect a minimum of $\lceil n/m \rceil$ intervals. Let $g : X \to Z$ compute the subset $Z = \{z : s_y \geq s_c; \{c, y\} \subseteq Y\}$ containing the intervals whose left endpoint is right of or equal to α:th rightmost left endpoint, denoted c, from $Y = \{l : p(e) \leq p(x); \{x, e\} \subseteq X\}$ the subset of intervals from set X whose weight is larger than the α:th heaviest interval denoted e. Let L_r and R_r denote two aggregated sets,

$$\begin{aligned} L_r &= \{l_1 \dots\ l_\beta\} \subseteq \bigcup_{i=1}^{m} g(B_i - C_i^r - \cup_{k=0}^{r-1}(L_k \cup R_k)) \\ R_r &= \{g_1 \dots\ g_\gamma\} \subseteq \bigcup_{i=1}^{m} g(C_i^r - \cup_{k=0}^{r-1}(L_k \cup R_k)) \\ L_0 &= R_0 = \emptyset \end{aligned}$$

where, $0 < r < m$, $\beta \leq \alpha$, $\gamma = \lceil n/m \rceil - \beta$ such that $|L_r| + |R_r| = \lceil n/m \rceil$. Using L_r and R_r, let $S_1, \dots, S_m$ constitute the outputs of the rounds where each S_r is given by equation (3.19).

$$S_r = \begin{cases} L_r \cup R_r, & r < m \\ \bigcup_{i=1}^{m} B_i - L_r \cup R_r, & r = m \end{cases} \tag{3.19}$$

From here and onwards, note that for an interval $a_j \in B_j$ there are now only two options to consider as to its profit contributions. If there is an interval a_o from an optimal solution O contained in $B_j \cup O$, then either (1) a_j has a weight equal to a_o and is in L_r, or (2) a_j has a weight less than a_o and a_o is in R_r. As case (1) directly implies optimal weights $p(a_j) = p(a_o)$ in L_r, it is implied from the definition and Lemma 4 that the the heaviest interval with the leftmost left endpoint taken from every d:th set constitutes non-overlapping set of cardinality $\lfloor \max\{b, 2\}/2 \rfloor$. The following convergence function, therefore, applies to L_r if case (1) applies to all sets B_j

$$f_{conv(1)}(r) \leq 1 - \frac{\lfloor \max(b, 2)/2 \rfloor}{\min(b, m - r)|B_j|}, \quad 0 \leq b \leq m \tag{3.20}$$

where b denotes the number of sets containing at least one interval from the optimal solution $1 \leq b \leq m$. Furthermore, from the definition of S_r it follows that for case (1) a_o must always be in $C_r^{d=2}$, and for case (2) it must be in $B_j - C_j^{d=2}$. The maximum number of intervals both in $B_j - C_j^{d=2}$ and the optimal solution O is given

by $|B_j - C_j^{d=2} \cup O| \leq |B_j|$, while the maximum number intervals in $C_j^{d=2}$ and O is given by $|C_j^{d=2} \cup O| \leq 1$. Case (2) can, therefore, be bounded by the following convergence function

$$f_{conv(2)}(r) = 1 - \frac{\lfloor \max(b_1, 2)/2 \rfloor}{\min(b_1, m-r)|B_j| + (b - b_1)}, \quad 0 \leq b_1 \leq b \tag{3.21}$$

where b_1 denotes the number of sets $B_j - C_j^{d=2}$ containing an interval from the optimal solution, and $b - b_1$ consequently denotes the number of sets $C_j^{d=2}$ containing an interval from the optimal solution. Moreover, let b_d represent the number of intervals in C_j^d from the optimal solution and from (3.21) follows (3.22).

$$\lim_{\sum_{i=1}^{d-1} b_i \to 0} b \to \frac{m}{d} \tag{3.22}$$

Because $m/d \leq |B_j|$, (3.21) dominates both (3.20) and $f^*(r)$ for all value of r unless $b = 1$. If the entire optimal solution is within one set or if b is small enough, the following general bound dominates $f_{conv}(r)$

$$f_{conv(g)}(\delta + \epsilon) < 1 - \epsilon \frac{\lceil \frac{|B_j|}{m} \rceil}{|B_j|}, \quad b \geq 1 \tag{3.23}$$

for $1 \leq \epsilon \leq \log n - \delta$ and $\delta = \log(n)(b - 1/b)$. In conclusion, the optimal independent sets from each output $S_1, \ldots, S_m$ has an *error* bounded by the following convergence function

$$f_{conv}(r) = \min(f_{conv(2)}(r), f_{conv(g)}(r))$$

which dominates $f^*(r)$ such that $f_{conv}(r) < f^*(r)$ for all $1 \leq r < \log n$. The sets L_r and R_r can be created in $\mathcal{O}(n)$ time, and thus, so can S_r.

□

Theorem 3. *There is a progressive algorithm for the Weighted Interval Scheduling Problem that runs in* $\log n$ *rounds, denoted by indices* r*, in* $\mathcal{O}(n \log n)$ *time with a round time of* $\mathcal{O}(n)$*, and that obtains a convergence* $f_{conv}(r) \leq f^*(r)$ *with respect to the error function* $error_{WIS}(S_r)$*.*

Proof. Assuming that h_r requires no more than $\mathcal{O}(n)$ time per round, *Algorithm 4* with h_r as enumeration strategy h (see *Algorithm 4*, row 9) will guarantee a correct solution, a convergence of at least $f^*(r)$, a round time of $\mathcal{O}(n)$, and a total time of $\mathcal{O}(n \log n)$. Let algorithm h_r use the partitioning algorithm described in Lemma 6 for each round $0 \leq r < \lceil \log(\log n) \rceil$ and let it internally keep $D_r = \{\lfloor j(n/2^r) \rfloor : 0 < j < 2^r - 1, j \in \mathbb{Z}\}$ and $E = \{\lfloor j(n/m) \rfloor : 0 < j < m, j \in \mathbb{Z}\} \cap \{1, n\}$ as ordered sets of divider indices. Moreover, let F_r and G denote the intervals corresponding to the indices of D_r and E, respectively, and let A_r denote the output set from the r:th execution of h_r, where A_r is defined as follows

$$A_{r < \lceil \log(\log n) \rceil} = (F_r - \cup_{i=0}^{r-1} G_i) \cap H_r, \quad 2^r < \lfloor \log(n) \rfloor$$

where H_r is a sorted set consisting of $|H_r| = (\lfloor n/\log(n) \rfloor - |F_r - F_{r-1}|)$ intervals constructed by selecting $\lceil n/2^r \rceil$ intervals uniformly at random from each of the 2^r divider indices' ranges constituted by indices 0, n and the divider indices in D_r.

Adopting heuristic h_r will ensure two things: firstly, that after $r = \lceil \log(\log n) \rceil$ rounds, with round time $\mathcal{O}(n)$, h_r has partitioned the intervals into m externally ordered sets; and secondly, the set of unevaluated intervals has by round $r = \lceil \log(\log n) \rceil$ shrunk by $|I - \bigcup_{r=1}^{\lceil \log(\log n) \rceil} A_r|$ intervals.

For later rounds $r \geq \lceil \log(\log n) \rceil$, let $B_1 \dots B_m$ denote the m externally ordered interval subsets of cardinality $\lceil n/m \rceil$ constituted by the divider indices' ranges from the divider indices in E; the rest follows from Lemma (6). □

4

Concluding Remarks

The result of this thesis is two progressive algorithms for the Weighted Interval Scheduling Problem, utilising two different enumeration strategies, one stochastic and one deterministic.

Even though Dynamic Programming builds its solution from the solutions of its partial subproblems, we did not find any natural way of utilising these subproblems when designing our progressive algorithms. Instead, we abstracted the Dynamic Programming to a mean of getting an optimal solution. The sorting, which is necessary e.g. to build the memoisation table, proved to constitute the biggest bottleneck timewise. The focus moved to the order in which groups of intervals was fed to it, an order which we denoted the algorithm's *enumeration strategy*.

The stochastic enumeration strategy enables an expected linear monotone decrease in the error, as to the error function presented. The deterministic partitioning based enumeration strategy tied the decrease in the error to two rounds, which we denoted notable rounds.The round wherein the enumeration first considers the heaviest interval constitutes our first notable round. The round wherein the enumeration has internally partitioned the remaining intervals into equally sized *externally ordered* subsets constitutes our second notable round.

Alterations to the traditional enumeration strategy resulted in a recurrent need for re-sorting to fulfil the prerequisite of the Dynamic Programming routine adopted. With this came the risk of the total time of the progressive algorithm approaching the total time of consecutive re-runs of the traditional non-progressive algorithm. The framework adopted does only consider the time complexity asymptotically. Consequently, the actual time-constants of each rounds won't be directly deducible, but rather appear fluid. The framework, therefore, effectively hides the actual distribution of time-constants between rounds.

It would be interesting to conduct a more detailed time analysis on the time impact the re-sorting has on each round. Further optimisation efforts could focus on reducing the amount of re-sorting needed for these middle rounds. More research is required to bridge the gap between our progressive algorithm and its conventional non-progressive siblings as well as to smooth its convergence profiles.

Bibliography

[1] J.-D. Fekete and R. Primet, "Progressive analytics: A computation paradigm for exploratory data analysis," 2016.

[2] J. Jo, J. Seo, and J. Fekete, "Panene: A progressive algorithm for indexing and querying approximate k-nearest neighbors," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, pp. 1347–1360, Feb 2020.

[3] M. Kubina, M. Varmus, and I. Kubinova, "Use of big data for competitive advantage of company," *Procedia Economics and Finance*, vol. 26, pp. 561 – 565, 2015. 4th World Conference on Business, Economics and Management (WCBEM-2015).

[4] L. Einav and J. Levin, "Economics in the age of big data," *Science*, vol. 346, no. 6210, p. 1243089, 2014.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[6] S. P. A. Alewijnse, T. M. Bagautdinov, M. de Berg, Q. W. Bouts, A. P. ten Brink, K. Buchin, and M. A. Westenberg, "Progressive geometric algorithms," in *Proceedings of the Thirtieth Annual Symposium on Computational Geometry*, SOCG'14, (New York, NY, USA), pp. 50:50–50:59, ACM, 2014.

[7] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Indexing for interactive exploration of big data series," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 1555–1566, ACM, 2014.

[8] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Efficient and progressive group steiner tree search," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 91–106, ACM, 2016.

[9] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[10] C. E. R. R. L. S. C. Cormen, Thomas H.; Leiserson, *Introduction to Algorithms (3rd ed.)*. MIT Press and McGraw-Hill, 2009.

[11] A. Alexandrescu, “Fast deterministic selection,” 06 2017.

[12] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, “Time bounds for selection,” *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448 – 461, 1973.