IT 10 048
Examensarbete 30 hp
Oktober 2010

Filtering Algorithms for Discrete Cumulative Resources

Joseph Scott

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Filtering Algorithms for Discrete Cumulative Resources

Joseph Scott

Edge finding is an algorithm commonly used in constraint-based scheduling.
For problems of n tasks, several O(n log(n)) algorithms exist for edge finding in unary resource scheduling; however, edge finding with cumulative resources is more complicated, resulting in $O(kn^2)$ complexity at best (where k is the number of distinct resource demands).
Recently Vilím(2009) proposed a new algorithm for cumulative edge finding, which uses a novel data structure called the theta tree to improve the complexity to O(kn log(n)).

This thesis presents work performed while implementing this new algorithm.
Several areas not covered in the original paper, such as the choice of tree structure and best loop order in the adjustment phase, are discussed here.
Additionally, several minor improvements to the algorithm are described, and two significant errors in the original paper identified.
The symmetry of adjusting each task's upper and lower bounds is considered in detail, and a proof of the monotonicity of the algorithm is provided.
Also, issues in creating a parallel version of this algorithm are discussed, and implementation details for several parallel version are provided.
In a separate paper, Vilím used the theta tree to design a max energy filtering algorithm, which handles tasks with variable durations and resource demands; this algorithm is also implemented here, and combined with the edge-finding algorithm.

Handledare: Mats Carlsson
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 10 048
Tryckt av: Reprocentralen ITC

Acknowledgements

This thesis is based on work carried out as part of an internship with the Combinatorial Problem Solving group of the Swedish Institute of Computer Science. I'd especially like to thank my advisor, Mats Carlsson, for his guidance, encouragement, and above all his patience in dealing with a barrage of drafts.

Thanks also to the ASTRA Research Group of Uppsala University, particularly to my reviewer Justin Pearson for his insightful feedback, and to Pierre Flener, whose excellent constraint programming course opened my eyes to this fascinating field.

I'd also like to thank Christian Schulte for taking the time to talk to me about edge-finding, and for pointing out several interesting questions on the topic.

I must also thank my parents, Tim and Susan Scott, for their unwavering belief in me over the years, and their constant encouragement and support throughout my occasionally quixotic academic career. They, along with Rick and Nancy Ast, provided the immeasurable help that made this sojourn in Sweden a reality, and I doubt I'll ever be able to thank any of them enough.

Last, but far from least, I would like to thank my wife, Jennifer, without whose seemingly boundless love, support, and good humor none of this would have been possible. *Tu sei una stella...la mia stella.*

In closing, a note to readers who find themselves bored quickly with these pages: I humbly suggest perseverance up to page 3, at which point a sudoku has been provided for your amusement.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

The purpose of this thesis is to provide implementation details for two filtering algorithms, called edge-finding and max energy, for the scheduling of cumulative resources in constraint-based scheduling. This chapter provides background for these concepts. Section 1.1 introduces the concept of constraint programming. Section 1.2 defines the Cumulative Scheduling Problem, the problem that the edge-finding filtering algorithm will be used to solve. Section 1.3 explains the organization of the remainder of the thesis.

1.1 Constraint Programming

> Constraint satisfaction, in its basic form, involves finding a value for each one of a set of problem variables where constraints specify that some subsets of values cannot be used together. [17]

Constraint programming is a declarative programming method for modeling combinatorial optimization problems in terms of constraints. This is a fairly dense statement, so let us address the concepts one at a time.

Declarative programming is a style of programming distinguished from the traditional *procedural* style. In procedural programming, a problem is solved through the execution of a series of instructions. In contrast, in the declarative style the programmer states the logic of the problem in some form; the procedure of solving the problem so defined is determined by the computer.

In an *optimization* problem, the goal is to find the "best" configuration, commonly defined as the maximum or minimum value yielded by a set of functions over several variables; such a problem is *combinatorial* when there exists only a finite set of possible solutions [25]. The problems generally have concise representations, such as a graph or set of equations, but the number of possible solutions is too large to simply compare them all. While some combinatorial optimization problems are polynomial, a large class of interesting problems is NP-Complete[1], where the number of possible solutions grows exponentially

[1] A detailed discussion of complexity and intractability is beyond the scope of this work, but a few basic definitions may be helpful for the uninitiated reader. Optimization problems can be divided into two classes, based on how quickly the time required to solve the problem grows as the size of the problem instance grows. Problems for which there exists a solution algorithm that completes in a time that grows in a polynomial fashion with the problem size fall into

with the problem size [29]. NP-Complete problems are intractable, but often it is possible to find an interesting subproblem that is tractable, or to use a polynomial approximation to find a solution that is at least close to the optimum [19].

A *constraint* is a relation on the domains of one or more variables, the purpose of which is to specify those combinations of values that are acceptable. Constraints define conditions that must be true for all solutions to the problem. An example of a simple constraint is $X < Y$, which says that in any solution, the value of the variable X must be less than the value of the variable Y.

Constraints work by reducing the domain of the values that a variable may take in a solution. For example, say the domains of variables X and Y are defined as:

$$X \in \{2, 3, 4, 6\}$$
$$Y \in \{1, 2, 4, 5\}$$

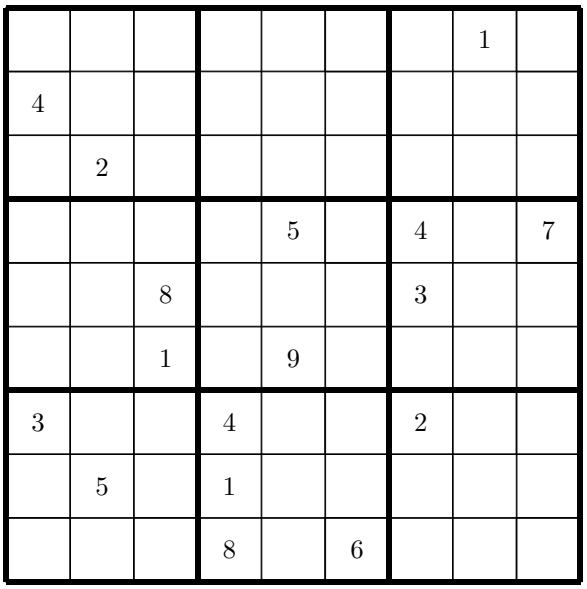
The constraint $X < Y$ would remove the value 6 from the domain of X, because there is no value in the domain of Y that makes the statement $6 < Y$ true. Similarly, the is no value in the domain of X for which $X < 1$ or $X < 2$, so the constraint would remove the values 1 and 2 from the domain of Y. This removal of values is called *strengthening* the domains of the variables. Constraints never add values to domains. If a constraint removes no values from the domains of any variables in the current state, then that constraint is said to be at *fixpoint*.

Constraint programming is often defined in terms of the Constraint Satisfaction Problem (CSP), which is: given a set of variables and constraints on those variables, does a solution exist that satisfies all the constraints? For many problems, providing any solution (or demonstrating that no solution exists) is sufficient; for other problems a listing of all solutions is required. Clearly this type of satisfaction problem is less strict than the general combinatorial optimization problem discussed above; however, the CSP can be easily modified to maximize or minimize an objective function, in what is called a Constrained Optimization Problem [3]. The CSP is sufficient for the purposes of this thesis, however.

---

the set P. Then there is another set of problems, for which there is a polynomial algorithm that can verify the correctness of a solution, but no known algorithm to find a solution in polynomial time; these problems are called NP problems. The hardest of the NP problems, those that are at least as hard as any other problem in NP, form the subset NP-Complete. It is generally supposed that there is no overlap between P and NP-Complete problems (i.e., that NP-Complete problems cannot be solved in polynomial time, instead of having as-yet undiscovered polynomial algorithms), although this conjecture, known as $P = NP$, remains unproven. There is also a broader class of very hard problems, those that may not even have a polynomial algorithm for verifying solutions; these problems are called NP-Hard. Many rigorous discussions of NP-completeness and intractability are available, primarily [19].

Figure 1.1 – A sudoku puzzle with 17 initial values, and exactly 1 solution [28]. The solution is provided in Appendix B.



1.1.1 Example: Sudoku

Sudoku is a popular combinatorial number puzzle, consisting of a 9×9 grid that is further subdivided into 9 non-overlapping 3×3 regions.

In a solution, each of the 81 squares in the grid must hold a number between 1 and 9, inclusive; furthermore, solutions satisfy the following conditions:

1. every number in a horizontal row must be different;
2. every number in a vertical column must be different; and
3. every number in a 3×3 region must be different.

In the initial state, several squares have values specified; these values are referred to as *hints*. The puzzle is solved by finding consistent values for all 81 squares.

These relatively simple rules yield a puzzle that cannot reasonably be solved through simple trial and error. There are 9^{81}, or slightly less than 2×10^{77}, possible ways to fill out the 9×9 grid, of which approximately 6.67×10^{21} are grids that satisfy all three rules [16]. And yet, by fixing the initial values of as few as 17 squares, it is possible to create a puzzle with only one solution [28]. Other grid sizes are possible—we can imagine a hexadecimal suduko where 16 numbers are arranged on a 16×16 grid divided into 4×4 regions—and the state space is highly exponential, with $n^{2(n+1)^2}$ assignments possible for an n-sudoku that has a $n^2 \times n^2$ grid. It is therefore no surprise that generalized sudoku has been shown to be NP-Complete [44].

The game of sudoku is easily encoded as a Constraint Satisfaction Problem, however. Each position in the grid is represented by a decision variable (except for positions that correspond to a hint) with an initial domain of [1..9]. Each

rule can be encoded as a collection of simple constraints; every rule specifies that the value in a particular cell cannot be equal to any of the values in a set of 8 other cells, so all three rules can be enforced for a single cell through the use of 20 $\neq$ constraints (there is an overlap of 2 cells between Rule 3 and each of the other two rules, so there are only 20 inequalities for the 24 cells). Taken all together, and throwing out duplicate inequalities, there are 738 inequalities required to model the game.

This formulation is inefficient for the user, who is forced to specify this large number of constraints; it is also inefficient for the constraint solver. Suppose there are three cells, a, b, and c, that are in the same row, with domains

$$a, b \in \{1, 2\} \qquad c \in \{2, 3\} \tag{1.1}$$

With the constraints specified as pairs of inequalities, there are no values that will be removed from the domain of c—for each value of the current domain of c, there exist values in the domains of a and b that make the constraints $a \neq c$ and $b \neq c$ true. Yet $c = 2$ does not actually participate in any solution, since the only possible assignments for the other two variables are $\{a = 1, b = 2\}$ and $\{a = 2, b = 1\}$, either of which will cause the removal of 2 from the domain of c. The problem is that each constraint only knows about two of the variables, so the solver is unable to make deductions that require knowledge of several variables at once.

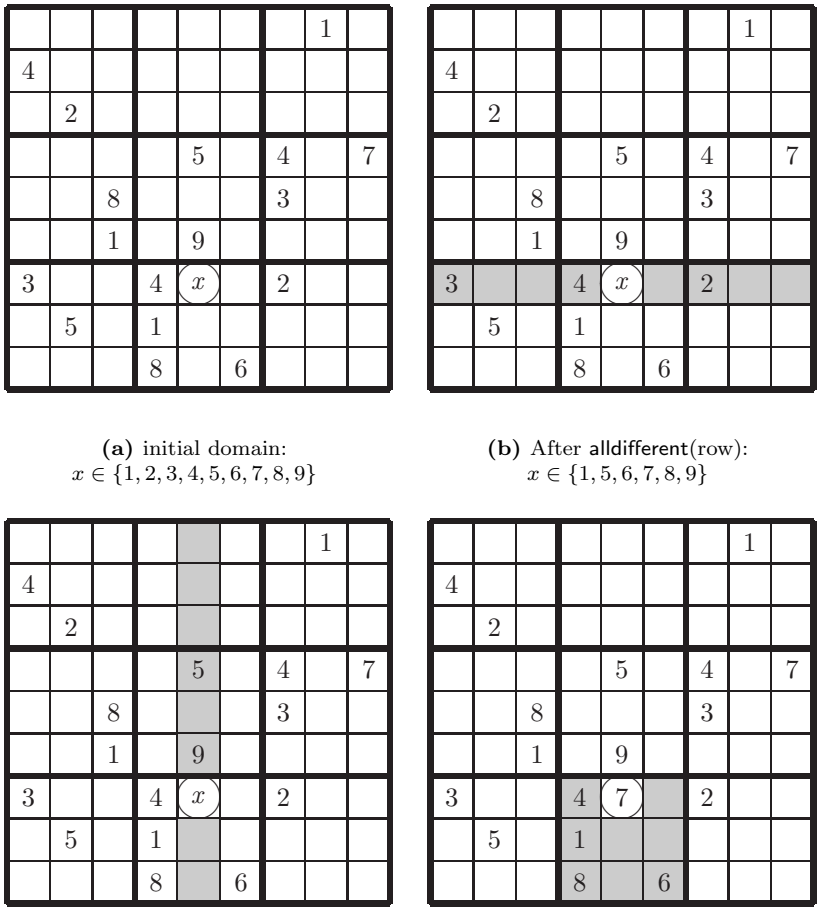
By encoding the rules as *global* constraints, this problem is eliminated. Global constraints apply to arbitrary-size sets of variables. Usually global constraints can be decomposed into several simple constraints; the availability of a global constraint not only simplifies the specification of the CSP, but also helps the solver to work more effectively by clarifying the underlying structure of the problem itself [36].

For sudoku solving, the alldifferent global constraint is useful. The most famous and widely studied of global constraints, alldifferent specifies that each of the variables it applies to must take a different value. A set of 27 alldifferent constraints, one for each row, column and region, is sufficient to define the sudoku problem. Figure 1.2 illustrates the use of sequential alldifferent constraints to determine the value of a variable.

Such an implementation will also perform better in situations like the one described above, if the algorithm that implements the alldifferent constraint is complete, meaning that the algorithm removes from the domains of its variables every value that does not participate in a solution. Fortunately, such algorithms exist for alldifferent, such as the elegant matching theory based algorithm proposed by Règin [26]. Given the domains in (1.1), a complete alldifferent algorithm will use the fact that either a or b must be equal to 2 to eliminate that value from the domain of c.

To solve the sudoku, the 27 alldifferent constraints are evaluated one at a time, possibly removing values from the domains of some of the variables on each evaluation. As the constraints overlap in terms of the variables they cover, several of the constraints will need to be executed repeatedly: each time a variable's domain is reduced by one constraint, all the other constraints that cover that same variable will need to be reevaluated.

During evaluation, it is possible that the domain of one of the variables will become empty, meaning that there is no solution to the problem that satisfies



(a) initial domain:
$x \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

(b) After alldifferent(row):
$x \in \{1, 5, 6, 7, 8, 9\}$

(c) After alldifferent(column):
$x \in \{1, 6, 7, 8\}$

(d) After alldifferent(region):
$x \in \{7\}$

Figure 1.2 – Application of the three alldifferent constraints to the circled value, showing how the domain is reduced by each constraint.

all the constraints. Otherwise, the evaluation of constraints continues until all of the constraints are at fixpoint; that is, until none of the constraints is able to remove any further values from the domains of the variables. At this mutual fixpoint, if the domain of every variable has only one value, then that fixpoint represents a solution to the puzzle. Puzzles designed for humans are generally of this type: there is one, and only one, solution, and it can be deduced without guessing any of the values.

For some puzzles, there is a mutual fixpoint in which one or more variables still have multiple values in their domains. Puzzles falling into this category are very challenging for humans, but can still be solved easily by constraint solvers by means of a search algorithm. One variable with a domain containing multiple values is chosen, and the domain is fixed to one of the possible values. Using this value, the constraints are evaluated again, reducing all the domains

further, until one of the three end states is reached again. If the end state is a solution, the problem is solved; if it is a failure, the solver goes back to the last variable it made a choice for, and makes a different choice; if the end result still has multiple choices for some variables, another is selected and another guess is made. This process is repeated until either a solution is found, or all possible choices have been eliminated.

1.1.2 Solving Constraints

The central idea of constraint programming is that the user defines the problem as a series of constraints, which are then given to a *constraint solver*. The solver is responsible for the procedural aspects of solving problem, which may be broadly divided into two categories: *inference* and *search*.

Inference

Inference is used to reduce the state space by creating more explicit networks of constraints. The constraints that make up the existing constraint network are combined logically to deduce an equivalent, but tighter, constraint network. The process can be repeated until either an inconsistency is detected or the solutions to the problem are found [13]. Compared to brute-force searching, inference provides better worst-case time complexity, but the space complexity is exponential [14].

Since solving the problem directly through inference is too costly, constraint programming generally uses more restricted forms of inference, reducing the state space only partially. This process is called *enforcing local consistency*, or *constraint propagation* [13]. The simplest form of local consistency enforced in propagation is *node consistency*, which ensures simply that the domain of each variable does not contain values excluded by any one of the constraints. More powerful is *arc-consistency*, which infers new constraints over each pair of variables (values are removed from the domain of a given variable if there exists another variable that cannot take a matching value to satisfy one of the constraints). Stronger types of consistency, such as *hyper-arc consistency*, *path consistency*, or *k-consistency* are less commonly enforced [3].

During propagation, each of the constraints is applied in turn, strengthening the domains of the variables when possible. One constraint may be applied several times, until it reaches fixpoint; once at fixpoint, the constraint can then be ignored until the domain of any of its variables is strengthened, at which point it will need to be executed again until it finds a new, stronger fixpoint. As the constraints of the problem are likely to overlap in their coverage of the variables, a cyclical process of strengthening generally arises. Each constraint strengthens some domains and then, reaching fixpoint, sleeps; this wakes other constraints affecting the variables of the strengthened domains, which results in further strengthening, possibly waking the first constraint once more. Eventually, the strongest mutual fixpoint for all the propagators is reached, and no further propagation is possible [3].

Search

For non-trivial problems, then, inference alone generally does not solve the problem. Any solutions may be found in the remaining, reduced state space; to find these solutions, a search algorithm is used. Search algorithms are ubiquitous in AI and combinatorial optimization methods, which typically use some variant of a heuristic search strategy.

In discussing search strategies, it is useful to think of the state space as a tree. Each node of the tree represents a partial solution; the edges between nodes represent transformations from one state to another [21]. In terms of constraints, the root node represents the state when all the variables have their initial domains. Each child node represents a removal of one or more values from one or more of the variable domains present in the parent node. Taken together, a set of sibling nodes represent all the possible tightened domains of the parent node. The leaf nodes with one value for each variable represent solutions.

In a heuristic search, a simpler (polynomial) calculation is used to determine a promising next step in the search. This heuristic is essentially making an informed guess based on limited information; this guess limits the search space, but there is no guarantee that the best solution, or for that matter any solution, is present in this reduced search space. As a result, heuristic search can get stuck on local optima or even fail to find a solution [21]. The worst-case performance of heuristic search is equivalent to a brute force enumeration of the state space; however, in practice a search with a good heuristic can be very efficient, although the performance of heuristic methods is often highly dependent on the minute adjustment of search parameters [15]. Neither the lack of a guaranteed solution nor the worst-case performance can be eliminated by developing a better heuristic or search technique, as both problems are inherent to the method [19].

The search algorithms in constraint solving are different from the heuristic search in one important way: by enforcing local consistency, the constraint solver only removes non-solutions form the search space. Depending on the search strategy used, a heuristic might fail to find a solution that was present in the original state space; constraint inference, however, is guaranteed to leave that solution in the reduced state space, so that it must eventually be located in a search [17].

The search algorithm employed in a constraint satisfaction problem is *backtracking*. Backtracking is the foundational search method in AI, and works as follows: viewing the state space as a tree, backtracking starts at the root and proceeds by selecting a child node; the process is repeated until a leaf node is reached. If the leaf node is a solution, the search terminates; if the leaf node represents an inconsistent state, the search moves back through the visited nodes until it reaches the most recently visited node that has as-yet unvisited children. Selecting one of these children, the search then proceeds as before, continuing until either a solution is found or the state space is exhausted [21].

In constraint programming, backtracking is intertwined with propagation to create an efficient search method. It has already been described how inference is used to reduce the initial state space; viewing the space as a tree, subtrees that demonstrably cannot hold solutions are pruned before the backtracking algorithm selects the next node. By selecting a child node, the search algorithm is further reducing the domain of one of the variables. At this point, propagation

is repeated, thus tightening domains again; from the view of the state space tree, some nodes are pruned from the subtree rooted at the current node. If this process results in an inconsistent state (i.e., a variable with an empty domain), the search backtracks to the most recently visited node with unvisited child nodes, restoring the domains of the variables to the state represented by that node, and the search continues from there [17].

1.1.3 Formal Definition

The informal definition of constraint programming given above is fairly broad, and can be applied to many different constraint systems. For a more formal treatment, we will restrict ourselves to one of these systems in particular, and consider only *finite domain* constraint solvers, in which variables are constrained to values from given, finite sets. These domains can be defined by enumerating each member of the set, or as a bounded interval; enumerated domains allow the constraint solver to tighten the domains more, but are only tractable for small domains [18].

We start by defining the Constraint Satisfaction Problem. This definition, as well as the remainder of this section, is adapted from the presentations in [32, 30].

Definition 1.1 (CSP)**.** *A* constraint satisfaction problem (CSP) *is a triple* $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ *where*

- $\mathbf{X} = \{x_1, x_2, \ldots x_n\}$ *is a set of* decision variables*,*
- $\mathbf{D} = \{D_1, D_2, \ldots D_n\}$ *is a set of* finite domains *associated with the variables in* $\mathbf{X}$ *(so* $x_i \in D_i$*),*
- $\mathbf{C} = \{C_1, C_2, \ldots C_k\}$ *is a finite set of* constraints*. A constraint* C_j *is a pair* $\langle R_j, S_j \rangle$*, where:*
  - S_j *is the* scope *of* C_j*, defined as* $S_j = \{x_{j_1}, x_{j_2}, \ldots x_{j_m}\} \subseteq \mathbf{X}$*, and*
  - R_j *is the* range *of acceptable values for the variables in the scope;* $R_j \subseteq D_{j_1} \times D_{j_2} \times \ldots D_{j_m}$*.*

The objective of a CSP is to find a *solution*: an assignment of values to all the variables that satisfies all the constraints of the problem.

Definition 1.2 (Assignment)**.** *An* assignment *on a CSP is an n-tuple* $a = \langle a_1, a_2, \ldots a_n \rangle \in D_1 \times D_2 \times \ldots D_n$ *which maps each variable of the CSP to a value.*

Definition 1.3 (Solution)**.** *An assignment a is a* solution *for a constraint C with a scope* $S = \langle x_1, x_2, \ldots, x_n \rangle$ *and range R, written* $a \in C$*, if*

$$\langle a(x_1), a(x_2), \ldots, a(x_n) \rangle \in R$$

a is a solution *to the CSP if a satisfies all* $C \in \mathbf{C}$*.*

Finally, we define the concept of a constraint store, which represents the current state of the variable domains at a particular point in the propagation process.

Definition 1.4 (Constraint Stores)**.** *A* constraint store *(or simply a* store*) s is a complete mapping from a fixed set of variables V to finite sets of values.*

1. *A store s is* failed *if $s(x) = \emptyset$ for some $x \in V$.*
2. *A variable $x \in V$ is* fixed *by a store s if $|s(x)| = 1$.*
3. *A store s_1 is* stronger *than a store s_2, written $s_1 \sqsubseteq s_2$, if $s_1(x) \subseteq s_2(x)$ for all $x \in V$.*
4. *A store s_1 is* strictly stronger *than a store s_2, written $s_1 \sqsubset s_2$, if $s_1(x) \subseteq s_2(x)$ for all $x \in V$, and there exists an $x \in V$ such that $s_1(x) \subset s_2(x)$.*
5. *An assignment A is* contained *in a store s, written $a \in s$, if $a(x) \in s(x)$ for all $x \in V$.*
6. *An* assignment-store *is a store in which all variables are fixed; the assignment-store for an assignment a, written $\mathsf{store}(a)$, is defined as*

$$\forall x \in V\colon \mathsf{store}(a)(x) = \{a(x)\}$$

During propagation, the constraint solver examines each of the constraints to attempt to reduce the state space. When the range of a variable x_i is reduced by one constraint, the program must then reconsider every other constraint that affects the same variable, in other words all C_j such that $x_i \in S_j$. When the reapplication of a given constraint causes no further reduction in the range of any variable in its scope, the constraint is said to be at *fixpoint*. Propagation continues until all of the constraints are at fixpoint.

1.1.4 Propagators

Algorithms that implement constraints are called *propagators*, or *filters*. Propagators are functions operating on constraint stores, and must have certain properties. Propagators must be *contracting*: they can remove values from stores, but can never add values. Propagators must be *correct*: they can never remove solutions for the constraint they implement. Finally, propagators must be *checking*: they must be able to distinguish solutions from non-solutions. What follows is a definition that meets these criteria:

Definition 1.5 (Propagator)**.** *A propagator p is a function from stores to stores that is:*

1. *contracting: $p(s) \sqsubseteq s$ for all stores s.*
2. *monotonic: for all stores s_1 and s_2, $s_1 \sqsubseteq s_2 \implies p(s_1) \sqsubseteq p(s_2)$*

A propagator p implements *a constraint C iff*

$$\forall a\colon a \in C \iff p(\mathsf{store}(a)) = \mathsf{store}(a)$$

The correspondence between constraints and propagators is not one-to-one. There may be several propagators that implement the same constraint. Similarly, a constraint may be implemented as a collection of simple propagators, and these propagators may be reused to implement parts of many constraints [32].

Propagators and Constraint Solvers

The activities of a propagator may be divided into two categories: external and internal. The external activities of the propagator handle the interaction of the propagator with the constraint solving system. The solver needs information regarding the current state of the propagator, such as:

Fixpoint A store s is a *fixpoint* for a propagator p when $p(s) = s$. A propagator at fixpoint does not need to be executed again by the solver until one of the domains of the variables of the propagator is reduced.

Entailment A propagator p is *entailed* by a store s if all stores $s' \sqsubseteq s$ are fixpoints for p. An entailed propagator can be removed from the set of active propagators; it does not need to be executed again.

Rewriting After some of the variables affected by a propagator have become fixed, it may be possible to replace the propagator with a simpler propagator over the remaining, unfixed variables

Solvers are not able to determine when a propagator is entailed, or when it can be rewritten, as these states depend on the internal logic of the propagator; the propagator must determine for itself when such a state has been reached, and inform the solver. On the other hand, propagators don't know when they need to executed again; solvers must have some sort of event notification system, to inform propagators when (and how) variable domains have been modified.

Filtering Algorithms

The internal activity of the propagator is, once invoked, to attempt to remove values from the domains of its variables. This part of the propagator is called the *filtering algorithm* [32].

So far it has been assumed that the propagator is able to achieve complete filtering for the constraint it implements; in other words, for the variables it applies to, the propagator removes every value that does not participate in a solution. As described in [36], this assumption may sometimes be too strong, resulting in computations that are too costly, perhaps to the point of intractability. In these cases, it may make more sense to implement a partial filter for the constraint.

One type of partial filtering has already been mentioned here: bound consistency. Instead of attempting to calculate consistency based on an enumeration of every possible value in the domains of the variables, a propagator that implements bound consistency uses only the minimum and maximum values in the domains of each of its variables, and adjusts these minimum and maximum values when possible.

For some constraints, finding a solution is NP-hard, and there is no known way to find a polynomial filtering algorithm. Propagators for these constraints can use a form of partial filtering called *relaxation*. For a constraint C with a set of solutions R, a relaxation is a constraint C' with a set of solutions R', such that $R \subset R'$; in other words, every solution of C is also a solution of C'. Propagation with the relaxed constraint will not remove as many non-solutions as the original constraint, but a relaxed constraint with a polynomial algorithm is clearly preferable to a tighter, exponential algorithm.

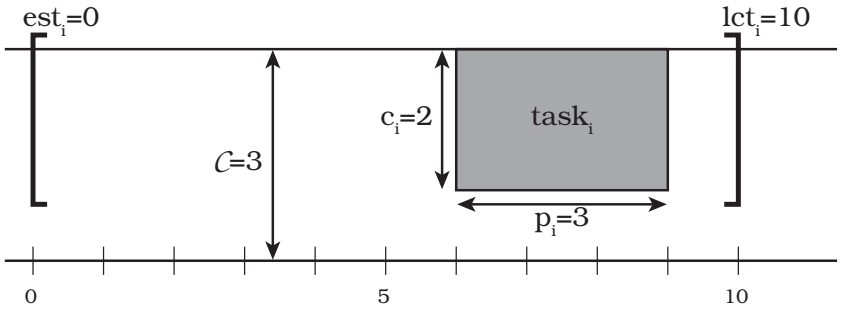


Figure 1.3 – A graphical representation of several characteristics of a task for a constraint scheduling problem.

1.2 Scheduling

In the most general sense, a scheduling problem consists of a set of tasks that share one or more finite resources, and that must be ordered into a sequence [4]. This rather broad characterization covers a multitude of scheduling models that have arisen in Operations Research and Constraint Programming over the years (comprehensive discussions from the OR viewpoint may be found in [4, 9, 8]). Most of these paradigms fall well outside the scope of this thesis. This discussion of scheduling will focus on a single category of scheduling problem, the Single Resource, or Cumulative, Scheduling Problem [1, 6].

1.2.1 The Cumulative Scheduling Problem

In the Cumulative Scheduling Problem (CuSP), there is a single resource with a finite capacity denoted by $\mathcal{C}$, and a set of tasks (sometimes called *activities* or *operations*) T that require that resource. The problem is to find a start time for each task, so that the amount of the resource required by all the tasks executing at a given time never exceeds the overall capacity.

Following the notation used in [40, 39], a task $i \in T$ is defined by these attributes:

- earliest possible start time est_i (sometimes called the release time r_i)
- latest possible completion time lct_i (sometimes called the deadline d_i)
- processing time (duration) p_i
- required capacity $c_i \leq \mathcal{C}$

These attributes may be graphically represented, as shown in Figure 1.3.

The values c_i and p_i are treated as constant within the CuSP, although they may of course be domain variables that are affected by other constraints (the max energy propagator, discussed in Chapter 5, implements such a constraint). The decision variables in the CuSP are the start times of the tasks; the possible start times fall in the bounded interval $[\mathsf{est}_i \,..\, \mathsf{lct}_i - p_i]$. There is no need to treat the interval of possible completion times separately, as the start times and

completion times are tied together by the durations of the tasks. It might be clearer to define the interval as being bounded by the earliest and last possible start times; however, as will be become clear when the filtering algorithm is discussed, the tightening of the earliest start times and the last *completion* times is symmetric, so from an algorithmic standpoint it is preferable to define the interval in those terms [12].

It is useful also to be able to refer to some derived attributes of a task:

- earliest possible completion time ect_i
- latest possible start time lst_i

The following conditions must always hold:

$$\begin{aligned} \mathsf{ect}_i &= \mathsf{est}_i + p_i \\ \mathsf{lst}_i &= \mathsf{lct}_i - p_i \\ \mathsf{est}_i &\leq \mathsf{lst}_i \leq \mathsf{lct}_i \\ \mathsf{est}_i &\leq \mathsf{ect}_i \leq \mathsf{lct}_i \end{aligned}$$

The definition of the start and completion times can be extended to a set of tasks $\Theta \subseteq T$:

$$\begin{aligned} \mathsf{est}_\Theta &= \min\{\mathsf{est}_i, i \in \Theta\} \\ \mathsf{lct}_\Theta &= \max\{\mathsf{lct}_i, i \in \Theta\} \end{aligned} \tag{1.2}$$

Unfortunately, the calculation of the earliest completion time and last start time for a set of tasks is non-trivial; in fact, it is an NP-Hard problem [23]. There are polynomial algorithms for computing a lower (upper) bound for ect_Θ (lst_Θ) which may be computed in a reasonable time frame; following [40], these bounds will be denoted Ect_Θ and Lst_Θ to distinguish them from the actual earliest completion and last start times.

The CuSP is a satisfiability problem, not an optimization problem; only one solution need be found to consider the problem solved. Nevertheless, CuSP is NP-Complete in the strong sense [6] (as are scheduling problems in general [19]), which means that no polynomial algorithm exists for solving it.[2]

1.2.2 Classifications of the Cumulative Scheduling Problem

Single Resource Scheduling Problems may be divided into two types based on the shared resource. In *unary* (or *disjunctive*) problems only one task may be scheduled at a time. In *cumulative* problems there is a resource with a limited capacity $\mathcal{C}$, and multiple tasks may be processed at one time on the resource as long as this capacity is not exceeded. Unary scheduling is a special case of cumulative scheduling, specifically that in which $\mathcal{C} = 1$, and $\forall i \in T : c_i = 1$; however, the unary scheduling model is sufficient to model many problems, and unary scheduling algorithms generally have substantially lower complexity than their cumulative counterparts, so the distinction remains a useful one. Nevertheless, the algorithms discussed here are used for cumulative scheduling.

[2] Except in the unlikely event that $P = NP$.

Within the category of cumulative problems, further division is possible. If, once a task is started, it must continue until it is finished without interruption, then the scheduling problem is said to be *non-preemptive*; on the other hand, if a task may be suspended while another task uses the resource, then the problem is a *preemptive* one. It is also possible to classify scheduling problems based on the *elasticity* of the tasks. In an *inelastic* problem, the capacity c_i is required by the task for its entire duration; in this type of problem we may define the *energy* of the task as:

$$\mathsf{e}_i = c_i p_i$$

In an *elastic* problem, at each time a task t uses an amount of the resource between 0 and c_t; the constraint is that the sum of the resource used over all times for the task must be equal to a required amount of energy. However the energy of an individual task is defined, it is possible to extend the concept of energy to a set of tasks:

$$\mathsf{e}_\Theta = \sum_{i \in \Theta} \mathsf{e}_i$$

Further discussion of preemptive and elastic scheduling may be found in [7]; for the remainder of this thesis, only the inelastic, non-preemptive case will be considered.

1.2.3 The cumulative Constraint

First defined in [1], cumulative is a global constraint that enforces the following condition on a set of tasks T, over time intervals t:

$$\forall t \quad \sum_{\substack{i \in T \\ s_i \leq t \leq s_i + p_i}} c_i \leq \mathcal{C}$$

The values c_i and p_i have already been defined; s_i is the decision variable, and represents the start time of the task i. In the terms of the CuSP defined above, s_i is the inclusive interval $[\mathsf{est}_i \,..\, \mathsf{lct}_i - p_i]$.

As mentioned previously in Section 1.2.1, the Cumulative Scheduling Problem is NP-Complete; as such, there is no algorithm that implements the cumulative constraint in polynomial time. Propagators for cumulative utilize filtering algorithms that implement a relaxation that can be computed in polynomial time [6]. Several of the most commonly used relaxations of the cumulative constraint are based on the calculation of overloaded time intervals.

As given in [23], the cumulative resource overload rule is:

Definition 1.6 (Overload)**.** *Whenever a subset of activities exists that requires more energy than that which is available from the start time of the subset to the end time of the subset, the problem is said to be* overloaded*, and has no solution. Formally:*

$$\forall \Omega \subset T : (\mathsf{e}_\Omega > \mathcal{C}(\mathsf{lct}_\Omega - \mathsf{est}_\Omega) \implies \textbf{\textit{overload}})$$

A situation with no overload is said to be *e-feasible*.

Overload checking is itself a relaxation of the cumulative constraint—the existence of any overloaded set implies that there is no solution to the CuSP

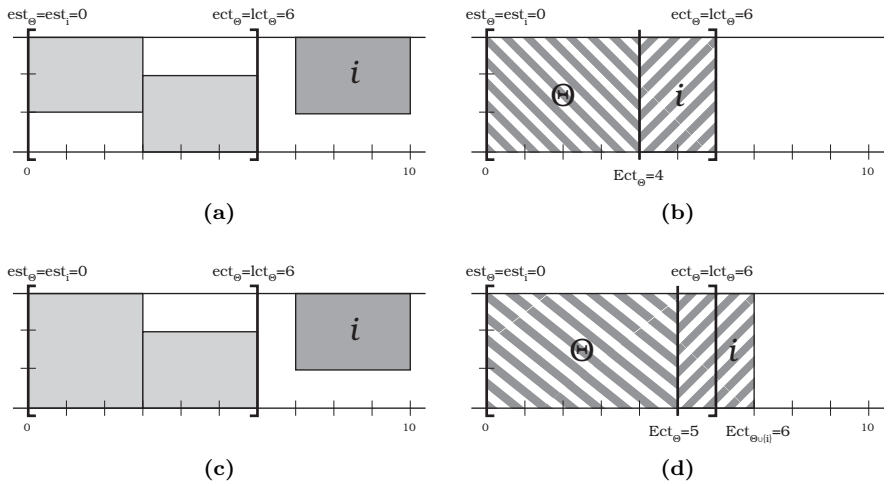


Figure 1.4 – Two examples of the elastic relaxation of the cumulative constraint. In both (a) and (c), task i cannot be scheduled in the interval $[\mathsf{est}_\Theta \,..\, \mathsf{lct}_\Theta]$. In the relaxations of these same two examples, (b) and (d), the energy of both Θ and i is treated as though it were elastic. In (d) the relaxation is strong enough to determine that i must come after Θ, but in (b) it is not.

[43]. It is possible, however, to modify the overload rule to go beyond failure checking, and begin to detect task bounds that must be adjusted. Starting with a feasible set of tasks Θ, these new algorithms locate another task $i \notin \Theta$ that, given its current earliest start time and last completion time, must be scheduled either during the interval $[\mathsf{est}_\Theta .. \mathsf{lct}_\Theta]$ or outside that interval. Next, it is determined if requiring i to execute inside this interval would cause an overload:

$$\mathsf{e}_{\Theta\cup\{i\}} > \mathcal{C}(\mathsf{lct}_\Theta - \mathsf{est}_\Theta) \tag{1.3}$$

Any i that satisfies (1.3) *must* be scheduled either before or after all the tasks in Θ. Using this knowledge, the bounds of i can be adjusted. Figure 1.4 demonstrates how this elastic relaxation works. In (a) and (c), a casual observation shows that, assuming the tasks to be inelastic, there is no way that task i can start before the time $t = 6$, due to the interference of the two tasks in Θ. Yet, the elastic relaxation of (a), shown in (b), fails to detect this fact; treated elastically, there is enough capacity between est_Θ and lct_Θ to schedule both i and all the tasks in Θ. On the other hand, the relaxation of (c), illustrated in (d), is sufficient to demonstrate that i cannot start until after the tasks in Θ. What we can see in (a) and (c) is that the earliest possible completion time for the tasks in Θ (ect_Θ) is $t = 6$. The problem is that, in the general case, computing the earliest completion time for a set of tasks is NP-hard [23]. The elastic relaxation of earliest completion time, here denoted Ect_Θ following [40], is easily computable, but accepts some solutions that a complete filer would not.

Several partial filtering algorithms for cumulative are based on this kind of elastic relaxation. In the *not-first/not-last* algorithm, this type of reasoning is used to locate an i that must come after (or before) at least one task in Θ [34]. In *edge-finding*, which is the focus of Chapters 2 and 3, the algorithm looks for

an i that must be scheduled before or after all the tasks in Θ [39]. What is being detected is a precedence relationship among the tasks; once this precedence is established, similar methods are employed to determine how much the bounds of i can be adjusted.

Relaxations can only implement partial filtering for the cumulative constraint. Relaxations based on other methods implement a different partial filtering, so it is possible to more closely approach complete filtering by combining two or more partial filters. For example, edge-finding is usually combined with a *time-tabling* filter that tracks resource availability over time. For example, if a task i exists such that $\mathsf{lst}_i < \mathsf{ect}_i$, then it is known that c_i units of the resource are unavailable to other tasks during that interval. If the remaining capacity is not enough to allow for the scheduling of a task j (i.e., $\mathcal{C} - c_i > c_j$), the the bounds of j can be adjusted to avoid this interval. Combining time-tabling with edge-finding usually results in stronger adjustment than either algorithm achieves on its own [5].

1.3 Structure of this thesis

The focus of this thesis is on recent work done by Vilím on cumulative scheduling problems. In earlier work [37, 42, 38], Vilím developed versions of several disjunctive scheduling algorithms with complexity $\mathcal{O}(n \log n)$ using a novel data structure called the Θ-tree. In [39], he uses an extension of this data structure to provide an elegant algorithm for cumulative edge-finding, with a complexity of $\mathcal{O}(kn \log n)$. In [40], he again uses the Θ-tree to develop an energetic filtering algorithm, which allows for filtering of tasks with variable duration and/or capacity requirements. This thesis presents an implementation of these two algorithms.

Chapter 2 provides a general discussion of edge-finding algorithms, and an overview of the new algorithm proposed by Vilím [39].

Chapter 3 The data structures required by the algorithm are discussed in regards to implementation. Finally, several portions of the algorithm that are not discussed in detail in [39] are fleshed out more thoroughly, such as the symmetry of earliest start time and last completion time adjustment, an efficient method for α-β-tree "cutting", and the need for overload checking.

Chapter 4 presents a discussion of the possibility of parallelizing the edge-finding algorithm. The performance of some different parallel implementations is analyzed.

Chapter 5 considers the max energy filtering algorithm proposed by Vilím [40] in detail. Modifications to the Θ-tree structure, required by the algorithm, are reviewed, and a pseudo-code implementation is presented.

1.4 Contributions

The main contributions of this thesis are as follows. Several portions of the edge-finding algorithm in [39] which were either omitted or discussed only in brief in the original paper are developed more fully here, such as the most appropriate choice of data structure for the Θ-tree, the proper use of the prec to retrieve detected precedences, and several factors which affect loop order in

the adjustment phase. Most significantly, a complete algorithm is given for performing the $\mathcal{O}(\log n)$ "cut" of the Θ-tree during the update calculation. Several minor improvements to the algorithm are outlined here, such as the elimination of an $\mathcal{O}(\log n)$ search procedure in the detection phase. Also, some errors in the original paper are identified, including one significant theoretical error, and an incorrect formula provided in one of the algorithms. Additionally, a parallel version of the Θ-tree edge-finder is given here, as well as a combined edge-finding/max energy filter.

Chapter 2

Edge Finding: Theory

Edge-finding is a filtering technique used in constraint scheduling problems to strengthen bounds on the possible scheduling time of tasks. Edge-finding was first applied to disjunctive scheduling problems in [2], and since that time several efficient algorithms ($\mathcal{O}(n \log n)$) have been developed for the disjunctive case [10, 41]. An efficient edge-finding algorithm for the cumulative case has proved somewhat elusive, however. An $\mathcal{O}\left(n^2 k\right)$ algorithm (where k is the number of distinct capacity requirements in the tasks) by Nuijten [23] was later improved to $\mathcal{O}\left(n^2\right)$ [7]; unfortunately, this algorithm was recently demonstrated to be incorrect [22], and while a corrected version of the $\mathcal{O}\left(n^2 k\right)$ algorithm was given, the $\mathcal{O}\left(n^2\right)$ algorithm had to be set aside. This chapter will consider a more recent version of the cumulative edge-finding algorithm [39] with $\mathcal{O}(kn \log n)$ complexity.

2.1 Overview

The central idea [7] of edge-finding algorithms is to find a set of tasks Θ and a task $i \notin \Theta$, in which it can be demonstrated that i will have to be scheduled before (or after) all tasks in Θ. For example, in Figure 2.1 task D must be

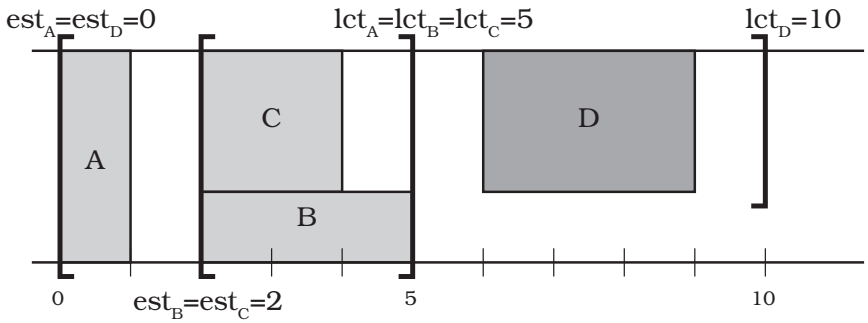


Figure 2.1 – A simple cumulative scheduling problem, redrawn from [39, Figure 1]. D must be scheduled after the set of tasks $\{A, B, C\}$.

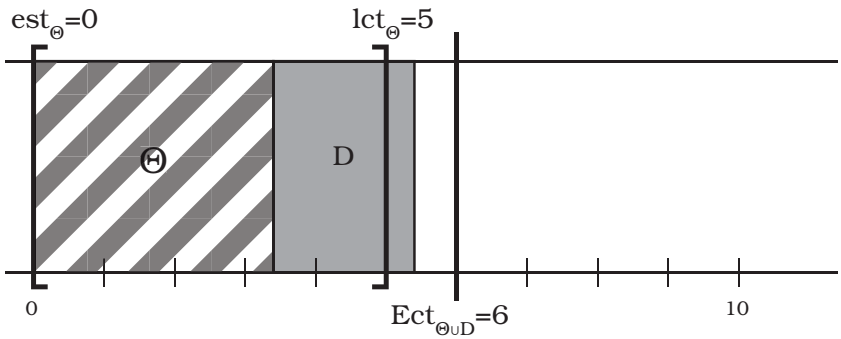


Figure 2.2 – The earliest completion time of the tasks from Figure 2.1, shown here as a combination of the fully elastic energy of the set $\Theta = \{A, B, C\}$ and D. This is a lower bound on $\mathsf{ect}_{\Theta \cup D}$, but is still sufficient to demonstrate that $\Theta \cup D$ cannot end before lct_{Θ}.

scheduled after all the tasks in the set $\{A, B, C\}$. The logic of this statement is as follows: no matter how we arrange the tasks in $\{A, B, C\} \cup D$, D cannot be completed before $t = 7$, but all tasks in $\{A, B, C\}$ must end before $t = 5$; therefore, D must end after the end of all tasks in $\{A, B, C\}$. Formally:

$$\mathsf{ect}_{\Theta \cup i} > \mathsf{lct}_{\Theta} \implies \Theta \lessdot i$$
$$\mathsf{lst}_{\Theta \cup i} < \mathsf{est}_{\Theta} \implies \Theta \gtrdot i$$

Unfortunately, figuring out the ect of a set of tasks is generally quite complex in cumulative scheduling. Edge-finding typically uses a simpler rule to determine a good lower bound for ect (which following [39, 40] is denoted here as Ect) by treating the energy of the tasks in Θ as fully elastic (see Figure 2.2). This lower bound may be stated as:

$$\mathsf{Ect}_{\Theta} = \mathsf{est}_{\Theta} + \lceil \frac{\mathsf{e}_{\Theta}}{\mathcal{C}} \rceil$$

Restating Ect somewhat gives us the standard expression of the edge-finding rule:

Definition 2.1 (Cumulative Edge-Finding Rule)**.** *Let T be the set of tasks to be scheduled, $\Theta \subset T$, and $i \in T, i \notin \Theta$. Then,*

$$\mathsf{e}_{\Theta \cup \{i\}} > C(\mathsf{lct}_{\Theta} - \mathsf{est}_{\Theta \cup \{i\}}) \implies \Theta \lessdot i$$
$$\mathsf{e}_{\Theta \cup \{i\}} > C(\mathsf{lct}_{\Theta \cup \{i\}} - \mathsf{est}_{\Theta}) \implies \Theta \gtrdot i$$

where "$\Theta \lessdot i$" means that all activities in Θ must end before the end of i, and "$\Theta \gtrdot i$" means that all activities in Θ must start after the start of i.

Once a set Θ and task i that satisfy the edge-finding rule have been located, it is possible to compute a new bound for i; however, a determination of the exact new bound for i is an NP-hard problem [7]. Once more, an update more easily computed is generally used, as illustrated in Figure 2.3. Once again, the energy of the tasks in Θ is treated as completely elastic, but now it is divided into two portions: the maximum amount of energy that can be scheduled without

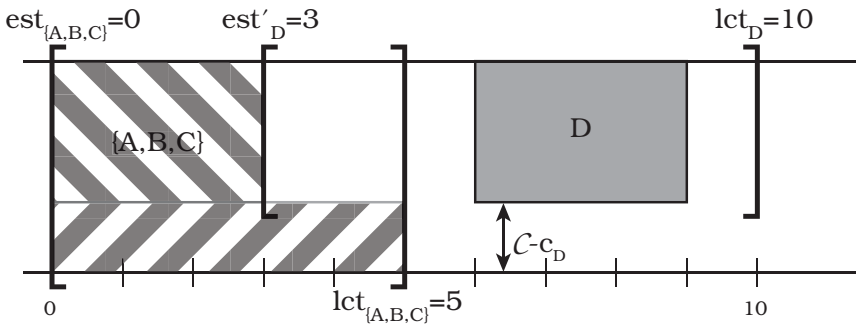


Figure 2.3 – Adjustment of est_D based on the energy of $\{A, B, C\}$. The lower diagonally striped area represents the part of $\mathsf{e}_{\{A,B,C\}}$ which need not interfere with the scheduling of D, while the upper region represents $rest_{\{A,B,C\}}$. Note that $\mathsf{est}'_D = 3$, which is not the strongest possible update for that bound.

interfering with i, and the rest of the energy (unnamed in [39], but elsewhere denoted rest_Θ [23]).

The update $\mathsf{est}'_D = 3$ shown in Figure 2.3 is not the strongest possible update; an examination of Figure 2.1 will show that $\mathsf{est}' = 4$ would also be correct. To strengthen this update, the sets $\omega \subseteq \Theta$ must be considered, as it is possible for some ω to give a stronger update than that given by all of Θ. For example, in Figure 2.4 it can be seen that with $\omega = \{B, C\}$ it is possible to discover $\mathsf{est}' = 4$, which happens to be the strongest possible update in this case. The interesting subsets are specifically those which have enough energy to prevent the simultaneous scheduling of i; in other words those subset in which $\mathsf{rest}_\omega > 0$, given [7]:

$$\mathsf{rest}_\omega = \mathsf{e}_\omega - (\mathcal{C} - c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega) \tag{2.1}$$

For each ω that meets this condition, an adjusted start time may be determined by dividing $rest_\omega$ by c_i, which gives the minimum processing time for the leftover

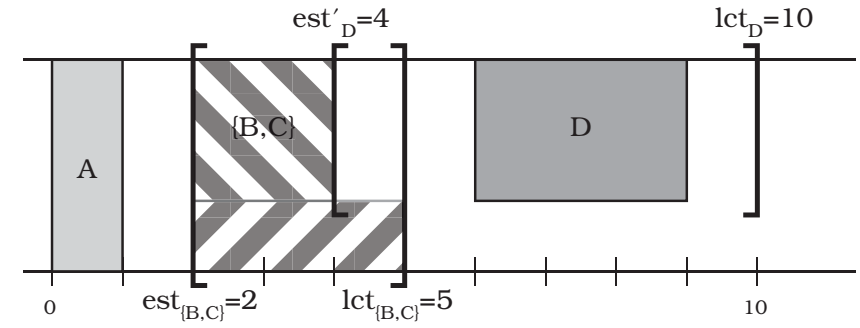


Figure 2.4 – Here est_D is updated based on tasks $\{B, C\}$ only, which generates a stronger update than using $\{A, B, C\}$

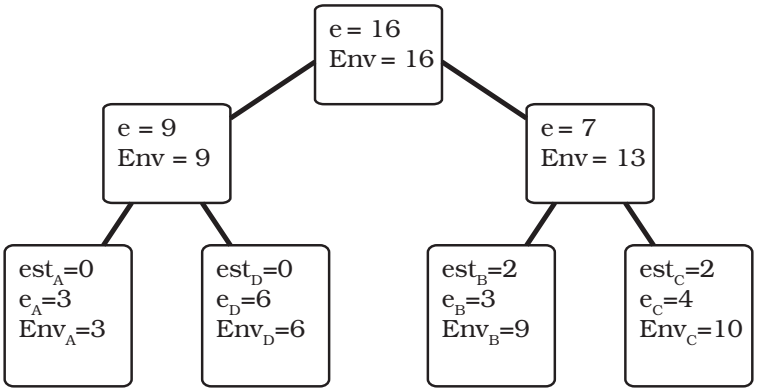


Figure 2.5 – A Θ-tree for the tasks represented in Figure 2.1. Adapted from [39, Figure 2].

energy. This yields an updated est_i of [7]:

$$\mathsf{est}'_i = \mathsf{est}_\omega + \left\lceil \frac{\mathsf{rest}_\omega}{c_i} \right\rceil \tag{2.2}$$

The strongest adjustment obtained from any ω that has been checked is then used to update est_i.

2.2 Edge-Finding using Θ-trees

2.2.1 Energy Envelope

In [39], Ect_Θ is modified by noting that all the tasks in Θ cannot end before the earliest completion time of any subset of tasks in Θ; or [40, Section 2.1]:

$$\mathsf{Ect}(\Theta) = \max_{\Omega \subseteq \Theta} \left\{ \mathsf{est}_\Omega + \left\lceil \frac{\mathsf{e}_\Omega}{\mathcal{C}} \right\rceil \right\}$$

This modification does not immediately appear to be helpful—the calculation of Ect_Θ now depends on the subsets of Θ, which should have an undesirable effect upon the complexity of the operation. In actuality, though, it allows for a great simplification. By restating the problem in terms of the energy envelope, or Env:

$$\mathsf{Env}(\Theta) = \max_{\Omega \subseteq \Theta} \{\mathcal{C}\, \mathsf{est}_\Omega + \mathsf{e}_\Omega\} \tag{2.3}$$

$$\mathsf{Ect}(\Theta) = \left\lceil \frac{\mathsf{Env}(\Theta)}{\mathcal{C}} \right\rceil$$

it becomes possible to quickly compute the Ect for all the subsets of Θ, using a novel data structure called a Θ-tree, as described in [40, Section 3.1].

In this structure, illustrated in Figure 2.5, the tasks are stored in the leaves and sorted in order of the bound (earliest start time or last completion time) which is being adjusted. Each interior node v holds computed values for the

subset of tasks represented by the leaves of the subtree rooted at v; the root node therefore holds the computed values of the complete set of tasks, Θ. Since the energy envelope of a set is the maximum energy envelope for any of its subsets, this means that the root node now holds the greatest energy envelope of any subset of tasks in the tree. Part of the elegance of the Θ-tree is in its handling of single node alterations: the insertion, removal, or alteration of a single task affects only the values in nodes lying on a leaf-to-root path, which can be visited in $\mathcal{O}(\log n)$ time. As a result, the maximum earliest completion time of any set of tasks in the tree may now be computed in $\mathcal{O}(\log n)$ time, as long as the set of tasks Θ is modified only by removing or adding single tasks between such re-computations.

For the calculation of Env, the tree needs to hold two values in each node: Env itself, and e. For interior nodes, these values are calculated as [39, (4),(5)]:

$$\mathsf{e}_v = \mathsf{e}_{left(v)} + \mathsf{e}_{right(v)} \tag{2.4}$$

$$\mathsf{Env}_v = \max\{\mathsf{Env}_{left(v)} + \mathsf{e}_{right(v)}, \mathsf{Env}_{right(v)}\} \tag{2.5}$$

Env_{root} therefore is equal to the largest Env_Ω for any $\Omega \subseteq \Theta$.

The edge finding rule can now be restated in terms of the energy envelope:

$$\forall j \in T,\ \forall i \in (T \setminus \mathsf{LCut}(T,j)):$$
$$\mathsf{Env}(\mathsf{LCut}(T,j) \cup \{i\}) > \mathcal{C} * \mathsf{lct}_j \implies \mathsf{LCut}(T,j) \lessdot i \tag{2.6}$$

where:

$$\mathsf{LCut}(T,j) = \{l, l \in T \ \&\ \mathsf{lct}_l \leq \mathsf{lct}_j\} \tag{2.7}$$

$\mathsf{LCut}(T,j)$ is the set of all tasks which must be processed before or concurrently with task j; this is a special case of a task interval [11]:

Definition 2.2 (Task Interval)**.** *Let $L, U \in T$. The task interval Ω_L^U is the set of tasks $\Omega_L^U = \{k \in T | \, \mathsf{est}_k \geq \mathsf{est}_L \wedge \mathsf{lct}_k \leq \mathsf{lct}_U\}$*[1]*.*

Stated in these terms, the set $\mathsf{LCut}(T,j) = \Omega_{min}^j$, where *min* is the task with the minimum earliest start time.

It is demonstrated in [22] that an edge-finder need only consider subsets of T which are task intervals, and further that to be complete an edge-finder adjusting est_i must consider all those intervals where $\mathsf{lct}_U < \mathsf{lct}_i$ and $\mathsf{est}_L \leq \mathsf{est}_i$. The Θ-tree edge-finding algorithm starts with all the tasks in the set Θ; in non-increasing order by last completion time, the tasks are moved into another set Λ, so that Θ always represents $\mathsf{LCut}(T,j)$ for some j, and Λ is the set of tasks which could conflict with that set. As already discussed, the definition of $\mathsf{LCut}(T,j)$ incorporates the upper bound of the task interval; in Proposition 2.1 it is shown that the calculation of Env_Θ using a Θ-tree will always define the lower bound of a task interval, as well.

Proposition 2.1. *Let Θ be a set of tasks, and let $\Omega \subseteq \Theta$ be the subset of those tasks with the maximum Env_Ω; then*

$$\exists L \in \Theta : \Omega = \{t \in \Theta, \mathsf{est}_t \geq \mathsf{est}_L\}$$

[1] The task interval notation used here follows that in [22].

Proof. By (2.3), $\mathsf{Env}_\Theta = \mathsf{Env}_\Omega$; further in [40, Proposition 2] it is shown that Env_Θ may be computed using a Θ-tree using the rules (2.4) and (2.5). Now consider an interior node v of this Θ-tree, with left and right child nodes l and r. If l and r are both leaf nodes, then (2.5) may be rewritten as:

$$\mathsf{Env}_v = \max\{\mathcal{C}\,\mathsf{est}_l + \mathsf{e}_l + \mathsf{e}_r, \mathcal{C}\,\mathsf{est}_r + \mathsf{e}_r\}$$

Note that Env_v depends on either $\mathcal{C}\,\mathsf{est}_l$ or $\mathcal{C}\,\mathsf{lct}_r$, but not both; (2.5) is applied recursively to all the interior nodes of the Θ-tree, so the Env of each node in the tree gets this first term from exactly one leaf node. Let the leaf node responsible for this term in the root node be L.

Next it will be demonstrated that

$$\mathsf{Env}_{root} = \mathcal{C}\,\mathsf{est}_L + \sum_{\substack{t\in\Theta \\ \mathsf{est}_t \geq \mathsf{est}_L}} \mathsf{e}_t \tag{2.8}$$

By definition, $\mathsf{Env}_L = \mathcal{C}\,\mathsf{est}_L + \mathsf{e}_L$; rewriting (2.8) we get

$$\begin{aligned}\mathsf{Env}_{root} &= \mathcal{C}\,\mathsf{est}_L + \mathsf{e}_L + \sum_{\substack{t\in\Theta\setminus\{L\} \\ \mathsf{est}_t \geq \mathsf{est}_L}} \mathsf{e}_t \\ &= \mathsf{Env}_L + \sum_{\substack{t\in\Theta\setminus\{L\} \\ \mathsf{est}_t \geq \mathsf{est}_L}} \mathsf{e}_t \end{aligned} \tag{2.9}$$

There are three cases to consider:

1. $\mathbf{est_t < est_L}$. As the leaves of the Θ-tree are arranged in non-decreasing order by earliest start time, t must lie to the left of L. Therefore any interior node v that has t and L in different subtrees must have t in the left subtree and L in the right. Since L is by definition the node responsible for the $\mathcal{C}\,\mathsf{est}$ term of Env_{root}, we know that $\mathsf{Env}_v = \mathsf{Env}_{right(v)}$. As e_v is not included in Env_v, it is not passed any further up the tree, so it is not included in Env_{root}.

2. $\mathbf{est_t > est_L}$. In this case, t must lie to the right of L in the Θ-tree. This means that L cannot be the rightmost leaf in the tree, so at some point as (2.5) is applied recursively up the tree there must be a node v with L in the left subtree and r in the right subtree. Once again, by definition L is responsible for the $\mathcal{C}\,\mathsf{est}$ term of Env_{root}, and hence for Env_v as well, so we know that $\mathsf{Env}_v = \mathsf{Env}_{left(v)} + \mathsf{e}_{right(v)}$. So e_t i included in Env_v, and eventually in Env_{root}.

3. $\mathbf{est_t = est_L}$. In this case the order of the two tasks in the tree is arbitrary. Assume that t lies to the left of L in the leaves of the Θ-tree. There must exist an interior node that has t in its left subtree and L in its right subtree, call this interior node v. L, by definition, is the node that supplies the term $\mathcal{C}\,\mathsf{est}$ to Env_{root}, so $\mathsf{Env}_v = \mathsf{Env}_{right(v)}$. $\mathsf{Env}_{right(v)}$ must also take this term from L; what is more, $\mathsf{Env}_{right(v)}$ may be derived from, at most, all of the leaf nodes in $right(v)$, giving it a maximum value of:

$$\mathsf{Env}_{right(v)} \leq \mathcal{C}\,\mathsf{est}_L + \mathsf{e}_{right(v)}$$

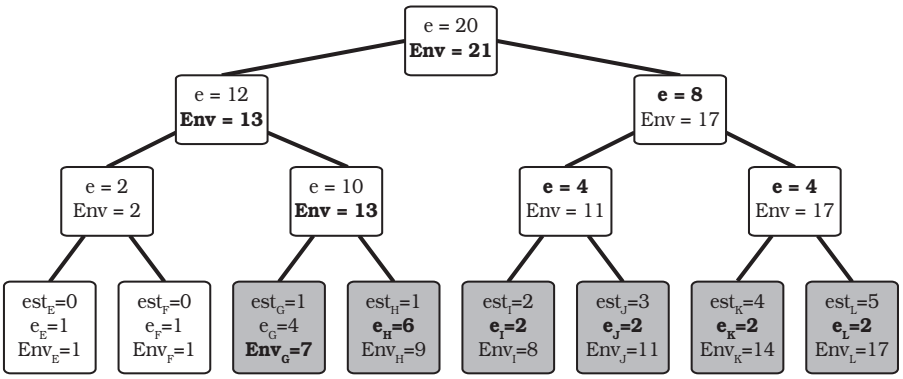


Figure 2.6 – Calculation of Env_Θ using a Θ-tree. Values in bold are those that contribute to the final calculation of Env_{root}; shaded leaves are those with energy included in that final calculation. These shaded leaves correspond to the set of tasks Ω, responsible for generating Env_Θ.

Furthermore, by (2.5), $\mathsf{Env}_{left(v)} + \mathsf{e}_{right(v)} \leq \mathsf{Env}_{right(v)}$, so

$$\mathsf{Env}_{left(v)} + \mathsf{e}_{right(v)} \leq \mathcal{C}\,\mathsf{est}_L + \mathsf{e}_{right(v)}$$
$$\mathsf{Env}_{left(v)} \leq \mathcal{C}\,\mathsf{est}_L$$

But recall that $left(v)$ includes t, so $\mathsf{Env}_{left(v)} \geq \mathcal{C}\,\mathsf{est}_t + \mathsf{e}_t$. Since $\mathsf{est}_t = \mathsf{est}_L$, this gives us:

$$\mathcal{C}\,\mathsf{est}_L + \mathsf{e}_t \leq \mathsf{Env}_{left(v)} \leq \mathcal{C}\,\mathsf{est}_L$$

This is a contradiction; therefore, t must lie to the left of L in the Θ-tree, and the proof is the same as case 2.

Taken together, these cases prove (2.9), and therefore (2.8).

Ω was defined as the set responsible for Env_Θ, so by (2.8):

$$\mathsf{Env}_\Omega = \mathcal{C}\,\mathsf{est}_L + \sum_{\substack{t \in \Theta \\ \mathsf{est}_t \geq \mathsf{est}_L}} \mathsf{e}_t$$

Comparing with (2.3), we can conclude that

$$\Omega = \{t \in \Theta | \, \mathsf{est}_t \geq \mathsf{est}_L\}$$

□

Figure 2.6 illustrates an example of a task interval responsible for generating Env_Θ; assuming that the tree shown represents the set $\mathsf{LCut}(T, j)$ for some task j, it can be seen that Ω is the task interval Θ^j_G. Note that e_G is considered to be included as it forms part of the calculation of Env_G according to (2.3).

2.2.2 Algorithm

Through an extension of the Θ-tree called the Θ-Λ-tree, it is possible to calculate a value of Env that includes all the tasks in Θ and at most one task from Λ [39,

Section 6.1]; this data structure is used to quickly locate tasks that make the implication in (2.6) true. In this way, the strongest ordering relationship for each task is detected.

Combining (2.2) with the ordering relationship just detected yields [39, Section 7, (EF2)]:

$$\mathsf{LCut}(T,j) \lessdot i \implies \mathsf{est}'_i = \max\{\mathsf{update}(j,c_i), \mathsf{est}_i\}$$

$$\mathsf{update}(j,c_i) = \max_{\substack{\omega \subseteq \mathsf{LCut}(T,j) \\ \mathsf{e}_\omega > (\mathcal{C}-c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}} \left\{ \mathsf{est}_\omega + \left\lceil \frac{\mathsf{e}_\omega - (\mathcal{C}-c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_i} \right\rceil \right\}$$

The approach taken therefore is to compute all values of $\mathsf{update}(j,c_i)$, which may then easily be used to update the values of earliest start time.

Once again, the ordered nature of the $\mathsf{LCut}(T,j)$ subsets is used to eliminate repetitive work. Consider that:

$$\mathsf{lct}_{j_1} < \mathsf{lct}_{j_2} \implies \mathsf{LCut}(T,j_1) \subseteq \mathsf{LCut}(T,j_2)$$

The computation of $\mathsf{update}(j_2,c)$ can therefore be divided into two parts: first compute $\mathsf{update}(j_1,c)$, then find the maximum update generated by a subset of $\mathsf{LCut}(T,j_2)$ which is not also a subset of $\mathsf{LCut}(T,j_1)$. Since, by definition, $\mathsf{lct}_{\mathsf{LCut}(T,j_2)} = \mathsf{lct}_{j_2}$, the calculation of $\mathsf{update}(j_2,c)$ can ignore all subsets ω where $\mathsf{lct}_\omega < \mathsf{lct}_{j_2}$—the only subsets which have not been seen before are those which include the task j_2. The update calculation can therefore be restated as [39, equation 10]:

$$\mathsf{update}(j_l,c) = \begin{cases} \mathsf{diff}(j_1,c) & \text{when } l = 1 \\ \max\{\mathsf{update}(j_{l-1},c),\ \mathsf{diff}(j_l,c)\} & \text{when } l > 1 \end{cases} \tag{2.10}$$

where [39, equation 11]:

$$\mathsf{diff}(j,c) = \max_{\substack{\omega \subset \mathsf{LCut}(T,j) \\ \mathsf{e}_\omega > (\mathcal{C}-c)(\mathsf{lct}_j - \mathsf{est}_\omega)}} \left\{ \mathsf{est}_\omega + \left\lceil \frac{\mathsf{e}_\omega - (\mathcal{C}-c)(\mathsf{lct}_j - \mathsf{est}_\omega)}{c} \right\rceil \right\} \tag{2.11}$$

Recall that in [22] it was demonstrated that the edge-finder only needs to consider task intervals, not other subsets of the tasks. In equation (2.10), these task intervals are considered according to their upper bound: $\mathsf{update}(j_l,c)$ is the maximum of $\mathsf{update}(j_{l-1},c)$ and the update generated by any task interval of the form $\omega_{j_k}^{j_l}$, where $k < l$. With $\mathsf{update}(j_{l-1},c)$ already calculated, all that remains is to determine the task j_k which forms the lower bound of the task interval which yields the strongest update.

During the detection phase, Env_Θ was used to determine the lower bound of the most energetic task interval in Θ, but that method is not sufficient here. Consider the example in Figure 2.7. The strongest adjustment of est_Q depends on the set $\{N,O,P\}$ ($\mathsf{update}(N,c_Q) = 4$, while $\mathsf{update}(P,c_Q) = 3$); however

$$\mathsf{Env}_O = \mathcal{C}\,\mathsf{est}_O + \mathsf{e}_O = 19 > 15 = \mathcal{C}\,\mathsf{est}_{\{N,O,P\}} + \mathsf{e}_{\{N,O,P\}} = \mathsf{Env}_{\{N,O,P\}}$$

so the calculation of $\mathsf{Env}(\mathsf{LCut}(T,O))$ will depend only on O, which clearly does not have enough energy to interfere with the scheduling of Q.

Instead, the algorithm must determine the task interval with the maximum energy envelope of any task interval that is also energetic enough to cause an update. This is accomplished in two stages; the first is to determine the task with the maximum earliest start time of any task that also forms the lower bound of a task interval energetic enough to cause an update. This earliest start time is called $\mathsf{maxest}(U, c_i)$, where U is the task forming the upper bound of the task interval, and c_i is the capacity requirement of the task whose bound is to be readjusted [2]. In the example in figure 2.7, $\mathsf{maxest}(O, c_Q) = P$; that is, P is the task with the greatest earliest start time that also forms the lower bound of a task interval ($\Theta_P^Q = \{O, P\}$) that is energetic enough to interfere with Q. Rather than iterate through all the possible lower bounds and determining whether each forms a task interval that would cause an update, in [39] the task responsible for $\mathsf{maxest}(U, c_i)$ is located in a root-to-leaf traversal of the Θ-tree. This procedure is given as [39, Algorithm 2], and runs in $\mathcal{O}(\log n)$ time.

The task interval with a lower bound defined by the task responsible for $\mathsf{maxest}(j_l, c)$ is the smallest task interval which needs to be considered when calculating $\mathsf{update}(j_l, c)$; all of the tasks in this interval are guaranteed to be part of the task interval which generates the strongest update. Consider that a smaller task interval, with the same upper bound and a lower bound possessing a greater earliest start time, cannot by definition have enough energy to interfere with the scheduling, and will not therefore generate an update at all; while a larger task interval, with a lower bound possessing a lower earliest start time, will perforce include all the tasks in the original interval.

Recall from Proposition 2.1 that the energy envelope is equal to the energy envelope of the task that forms the lower bound of the task interval, plus the sum of the energy of all tasks to the right of that lower bound in the Θ-tree. So in this example, it must be ensured that e_P and e_O are included in that sum. Essentially, these tasks can be ignored when computing the maximum envelope of the tasks in the Θ-tree; as these two tasks are the rightmost tasks in the tree, their energy can be added to the energy envelope calculated with the smaller tree, and the new set of tasks will still form a valid task interval.

Figure 2.8 shows the division of a Θ-tree for the current example at the task

[2] The explanation of *maxest* directly contradicts the definition of *minest* given in [39, Section 7]; the rationale for this change is presented in Section 3.3.2

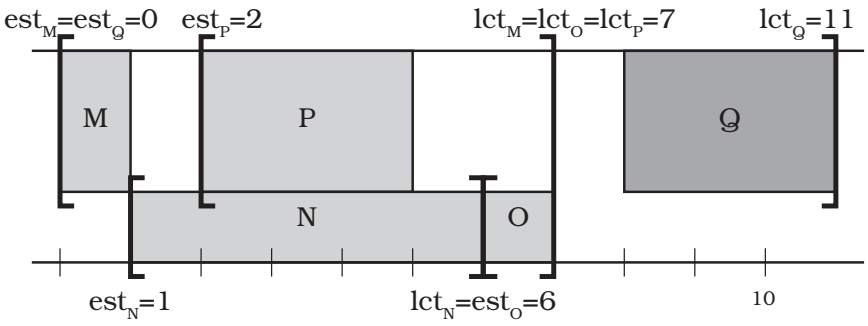


Figure 2.7 – Calculation of the strongest update for est_Q depends on the task interval $\Theta_N^P = \{N, O, P\}$.

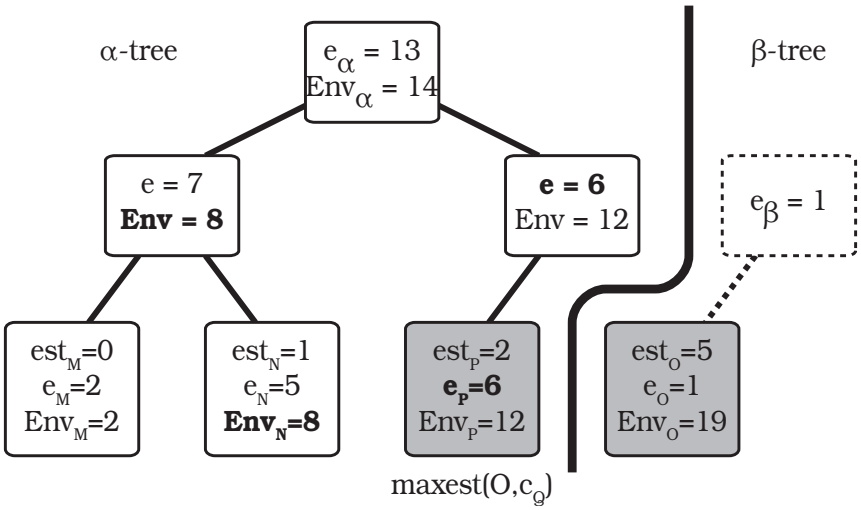


Figure 2.8 – A Θ-tree for the tasks in $\mathsf{LCut}(T, O)$ for the example in Figure 2.7. The shaded tasks are those forming the minimum task interval capable of interfering with the scheduling of task Q. O becomes the sole task in the β-tree, while P is the rightmost task in the α tree (guaranteeing that e_P will form part of Env_α). The bold values are those that are responsible for the final calculation of Env_α—note that task M does not contribute to this final calculation, indicating that M is not part of the task interval that yields the strongest update for est_Q.

$P = \mathsf{maxest}(O, c_Q)$. The task on the right, O, is the sole leaf of a tree that holds the members of the required minimum task interval. Note that P is not included in this set, even though it is a member of the interval; recall from Proposition 2.1 that e_P is already part of Env_P, and will also be part of any energy envelope based on nodes to its left in the tree, which is all the remaining nodes. An Env-like calculation, performed solely on the tasks remaining in the left part of the tree, finds that the maximum energy envelope is provided by the set $\{N, P\}$. Adding the energy envelope of the right set and the energy of the left set gives the energy envelope required for the update calculation.

These two sets of tasks are defined as:

$$\alpha(j, c) = \{l, l \in \mathsf{LCut}(T, j) \wedge \mathsf{est}_l \leq \mathsf{maxest}(j, c)\}$$
$$\beta(j, c) = \{l, l \in \mathsf{LCut}(T, j) \wedge \mathsf{est}_l > \mathsf{maxest}(j, c)\}$$

The maximum energy envelope of any task interval that can interfere with the scheduling of a task with resource requirement c is then calculated as [39, equation 15]:

$$\mathsf{Env}(j, c) = \mathsf{e}_{\beta(j,c)} + \mathsf{Env}(\alpha(j, c))$$

The result is used to compute the updated bound [39, equation 13]:

$$\mathsf{diff}(j, c) = \left\lceil \frac{\mathsf{Env}(j, c) - (\mathcal{C} - c)\,\mathsf{lct}_j}{c} \right\rceil \tag{2.12}$$

In practice, $\mathsf{Env}(j, c)$ is computed by first locating the node responsible for $\mathsf{maxest}(j, c)$, and then "cutting" the tree into α and a β subtrees. The "cut"

operation has a complexity of $\mathcal{O}(\log(n))$ (this is investigated in detail in Section 3.3.3).

For the actual bound adjustments, the precedences detected in the first stage of the algorithm are consulted. Once it is found that a task interval $\mathsf{LCut}(T, j)$ precedes the task i, the stored updates are used to modify the bound of i as follows:

$$\mathsf{LCut}(T, j) \lessdot i \implies \mathsf{est}'_i = \max\{\mathsf{update}(j, c_i), \mathsf{est}_i\}$$

Care must be taken to locate the correct such set $\mathsf{LCut}(T, j)$; this topic is discussed in detail in Section 3.3.1.

2.3 Monotonicity of Edge-Finding

Monotonicity is an important property for propagators to possess, as monotonic propagators ensure that propagation will yield the same final results, regardless of the order in which constraints are applied. The definition of monotonicity for propagators was given as part of Definition 1.5. What follows is a proof that the edge-finding filter in [39] is monotonic.

Proofs of monotonicity involve a comparison of different constraint stores (see Definition 1.4). To distinguish between the bounds of a task in different constraint stores, est_i^s will be used here to denote the earliest start time of task i in the store s, and the same for lct_i^s, ect_i^s, etc. Also, the left cut of T by j over the store s will be defined as:

$$\mathsf{LCut}^s(T, j) = \{l \colon l \in T \wedge \mathsf{lct}_l^s \leq \mathsf{lct}_j^s\}$$

Lemma 2.2 will simplify the monotonicity proof that follows:

Lemma 2.2. *Given constraint stores s_1, s_2 of a set of tasks (as defined by CuSP) T:*

$$s_1 \sqsubseteq s_2 \implies \forall i \in T \colon \mathsf{est}_i^{s_1} \geq \mathsf{est}_i^{s_2} \wedge \mathsf{lct}_i^{s_1} \leq \mathsf{lct}_i^{s_2}$$

Proof. Let $i \in T$, and suppose that $\mathsf{est}_i^{s_1} < \mathsf{est}_i^{s_2} \vee \mathsf{lct}_i^{s_1} > \mathsf{lct}_i^{s_2}$. By (1.2.1)

$$\begin{aligned} \mathsf{lct}_i^{s_1} \leq \mathsf{lct}_i^{s_2} &\implies \mathsf{lct}_i^{s_1} - p_i \leq \mathsf{lct}_i^{s_2} - p_i \\ &\implies \mathsf{lst}_i^{s_1} \leq \mathsf{lst}_i^{s_2} \end{aligned}$$

By definition, est_i and lst_i are the bounds of the domain of the start time of i:

$$start_i \in [\mathsf{est}_i \,..\, \mathsf{lst}_i]$$

Since by assumption at least one of the bounds of $start_i$ in s_2 must lie outside the bounds in s_1, $s_1(i) \not\subseteq s_2(i)$, and therefore by Definition 1.4.3 s_1 is not stronger than s_2, thereby proving the implication. □

Up to this point, the capacity requirement and duration of tasks has been assumed to be constant. This makes sense when considering just the edge-finding filter, as it does not adjust either of these task attributes. However, as noted in Section 1.2.1, these attributes could be decision variables for other filters in the CSP. If c and p are decision variables, then the calculations in the edge-finding algorithm will always be made with the *minimum* values in their domains; these minimums define the amount of energy the task is guaranteed

to require, and all of the logic of the edge-finding algorithm is based on required energy. Note that variable energy implies that stronger stores will guarantee:

$$\forall i \in T\colon c_i^{s_1} \geq c_i^{s_2} \wedge p_i^{s_1} \geq p_i^{s_2}$$

In the proof of monotonicity that follows, c and p will always refer to the minimum values in their respective ranges. Furthermore, to simplify the proof under variable capacity requirement, the following lemma is offered:

Lemma 2.3. *For an e-feasible CuSP over tasks T, where $\omega \subseteq T$, and $i, j \in T$ such that $c_i \geq c_j$:*

$$\frac{\mathsf{e}_\omega - (\mathcal{C} - c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_i} \geq \frac{\mathsf{e}_\omega - (\mathcal{C} - c_j)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_j}$$

Proof. Suppose that

$$\frac{\mathsf{e}_\omega - (\mathcal{C} - c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_i} < \frac{\mathsf{e}_\omega - (\mathcal{C} - c_j)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_j}$$

Then:

$$\begin{aligned}
\frac{\mathsf{e}_\omega - \mathcal{C}(\mathsf{lct}_\omega - \mathsf{est}_\omega) + c_i(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_i} &< \frac{\mathsf{e}_\omega - \mathcal{C}(\mathsf{lct}_\omega - \mathsf{est}_\omega) + c_j(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_j} \\
\frac{\mathsf{e}_\omega - \mathcal{C}(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_i} + (\mathsf{lct}_\omega - \mathsf{est}_\omega) &< \frac{\mathsf{e}_\omega - \mathcal{C}(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_j} + (\mathsf{lct}_\omega - \mathsf{est}_\omega) \\
\frac{\mathsf{e}_\omega - \mathcal{C}(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_i} &< \frac{\mathsf{e}_\omega - \mathcal{C}(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_j} \qquad (2.13)
\end{aligned}$$

By the definition of e-feasibility, $\mathsf{e}_\omega < \mathcal{C}(\mathsf{lct}_\omega - \mathsf{est}_\omega)$. So the two sides of (2.13) have equal, negative numerators, while $c_i \geq c_j$ by definition. Therefore (2.13) is a contradiction, and the lemma must be true. □

With those preliminaries out of the way, we proceed with a proof that the edge-finding rule defined in [39] is monotonic. Note that the rule given here as (TEF) is in a different form than the one presented in [39]; for clarity's sake, the detection and update phases of that presentation have been combined into a single rule, in a format similar to that in [22].

Proposition 2.4. *A propagator,* p*, for which*

$$\mathsf{est}_i^{p(s)} = \max\{\mathsf{est}_i^s, \mathsf{newEst}_i^s\}$$

given that newEst_i^s *is calculated by the Θ-tree Edge-Finding Rule:*

$$\begin{aligned}
\mathsf{newEst}_i = \max_{\substack{j \in T \\ \mathsf{Env}(\mathsf{LCut}(T,j) \cup \{i\}) > \mathcal{C}\,\mathsf{lct}_j}} \; \max_{\substack{\omega \subseteq \mathsf{LCut}(T,j) \\ \mathsf{rest}(\omega, c_i) > 0}} & \left\{ \mathsf{est}_\omega + \left\lceil \frac{\mathsf{rest}(\omega, c_i)}{c_i} \right\rceil \right\} \\
\text{where } \mathsf{rest}(\omega, c_i) = & \begin{cases} \mathsf{e}_\omega - (\mathcal{C} - c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega) & \text{if } \omega \neq \emptyset \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \qquad \text{(TEF)}$$

and having a symmetric rule for the calculation of $\mathsf{lct}_i^{p(s)}$*, is monotonic.*

Proof. Let T be a set of tasks, and p be a propagator that implements edge-finding by the given rule. Further, let s_1, s_2 be constraint stores for T, such that $s_1 \sqsubseteq s_2$.

Let i be a task in T. We will now show that $\mathsf{est}_i^{p(s_1)} \geq \mathsf{est}_i^{p(s_2)}$. There are two cases to consider:

1. $\mathsf{est}_i^{p(s_2)} = \mathsf{est}_i^{s_2}$. By Lemma 2.2, $\mathsf{est}_i^{s_1} \geq \mathsf{est}_i^{s_2}$; and by definition $\mathsf{est}_i^{p(s_1)} \geq \mathsf{est}_i^{s_1}$. Therefore $\mathsf{est}_i^{p(s_1)} \geq \mathsf{est}_i^{p(s_2)}$.

2. $\mathsf{est}_i^{p(s_2)} = \mathsf{newEst}_i^{s_2}$. Let $j_2 \in T$ be the maximal task j in the calculation of $\mathsf{newEst}_i^{s_2}$ by (TEF), and let $\omega \subseteq \mathsf{LCut}^{s_2}(T, j_2)$ be the maximal subset ω from the same calculation.

   By the definition of stronger stores, $c_i^{s_1} \geq c_i^{s_2}$. Lemma 2.3 allows us to conclude that

   $$\mathsf{est}_\omega^{s_2} + \left\lceil \frac{\mathsf{rest}^{s_2}(\omega, c_i^{s_1})}{c_i^{s_1}} \right\rceil \geq \mathsf{est}_\omega^{s_2} + \left\lceil \frac{\mathsf{rest}^{s_2}(\omega, c_i^{s_2})}{c_i^{s_2}} \right\rceil$$

   Also by the definition of stronger stores we can see that $\mathsf{e}_\omega^{s_1} \geq \mathsf{e}_\omega^{s_2}$. Combining Lemma 2.2 with (1.2), we see that

   $$\mathsf{est}_\omega^{s_1} \geq \mathsf{est}_\omega^{s_2} \qquad \text{and} \qquad \mathsf{lct}_\omega^{s_1} \leq \mathsf{lct}_\omega^{s_2}$$

   This allows us to conclude that $\mathsf{lct}_\omega^{s_1} - \mathsf{est}_\omega^{s_1} \leq \mathsf{lct}_\omega^{s_2} - \mathsf{est}_\omega^{s_2}$. Taken together, these inequalities allow us to conclude that $\mathsf{rest}^{s_1}(\omega, c_i) \geq \mathsf{rest}^{s_2}(\omega, c_i)$, and further

   $$\mathsf{est}_\omega^{s_1} + \left\lceil \frac{\mathsf{rest}^{s_1}(\omega, c_i^{s_1})}{c_i^{s_1}} \right\rceil \geq \mathsf{est}_\omega^{s_2} + \left\lceil \frac{\mathsf{rest}^{s_2}(\omega, c_i^{s_1})}{c_i^{s_1}} \right\rceil \geq \mathsf{est}_\omega^{s_2} + \left\lceil \frac{\mathsf{rest}^{s_2}(\omega, c_i^{s_2})}{c_i^{s_2}} \right\rceil$$

   Now, let $j_1 \in \mathsf{LCut}^{s_2}(T, j_2)$ be the task in $\mathsf{LCut}^{s_2}(T, j_2)$ with the maximum lct^{s_1}—the maximum completion time in s_1 of any task in that set, which was defined in s_2. By the definition of LCut^{s_1} we have:

   $$\omega \subseteq \mathsf{LCut}^{s_2}(T, j_2) \subseteq \mathsf{LCut}^{s_1}(T, j_1)$$

   Since $j \in T$, and all tasks in T are considered by the outer maximization in (TEF), we know that $\mathsf{LCut}^{s_1}(T, j_1)$ must be considered in that maximization. By definition ω was considered in the innermost maximization in s_2, so $\mathsf{rest}^{s_2}(\omega, c_i) > 0$). We already know that $\mathsf{rest}^{s_1}(\omega, c_i) \geq \mathsf{rest}^{s_2}(\omega, c_i)$; combined with the fact that $\omega \subseteq \mathsf{LCut}^{s_1}(T, j_1)$, this means that ω must be considered by the innermost maximization in s_1 as well. So the minimum value for the nested maximizations to take is

   $$\mathsf{est}_\omega^{s_1} + \left\lceil \frac{\mathsf{rest}^{s_1}(\omega, c_i^{s_1})}{c_i^{s_1}} \right\rceil$$

   Therefore:

   $$\mathsf{newEst}_i^{s_1} \geq \mathsf{est}_\omega^{s_1} + \left\lceil \frac{\mathsf{rest}^{s_1}(\omega, c_i^{s_1})}{c_i^{s_1}} \right\rceil \geq \mathsf{est}_\omega^{s_2} + \left\lceil \frac{\mathsf{rest}^{s_2}(\omega, c_i^{s_2})}{c_i^{s_2}} \right\rceil = \mathsf{newEst}_i^{s_2}$$

Taken together, these two cases show that $\mathsf{est}_i^{p(s_1)} \geq \mathsf{est}_i^{p(s_2)}$; by a similar demonstration it can be shown that $\mathsf{lct}_i^{p(s_1)} \leq \mathsf{lct}_i^{p(s_2)}$. Therefore, by Lemma 2.2, $p(s_1) \sqsubseteq p(s_2)$, and p is monotonic. □

Proposition 2.5. *The edge-finding algorithm presented in [39] implements the* Θ*-tree Edge-Finding Rule* (TEF)*.*

Proof. The algorithm does not need to consider every subset of T: (TEF) only makes use of the task j and subset ω responsible for the strongest update to the bound. For a task $i \in T$, [39, Algorithm 1] only considers the subsets $\mathsf{LCut}(T, j)$ for $j \in T$: $\mathsf{lct}_j \leq \mathsf{lct}_i$; but $\mathsf{LCut}(T, j)$ is a task interval T_L^U where $U = j$, and [22, Proposition 3] shows that an edge-finding algorithm can ignore task intervals that do not meet the test $\mathsf{lct}_U < \mathsf{lct}_i$. Furthermore, [39, Algorithm 1] stops comparing i against further tasks j once the first $\mathsf{LCut}(T, j)$ has been found that meets the criterion:

$$\mathsf{Env}(\mathsf{LCut}(T, j) \cup \{i\}) > \mathcal{C}\, \mathsf{lct}_j$$

However, the tasks j are selected in decreasing order by lct_j, and [39, Section 6] shows that under these conditions that fist j that is found will yield the strongest update.

Similarly, [39, Algorithms 2 & 3] do not actually consider every subset of $\mathsf{LCut}(T, j)$; however, as demonstrated in Section 2.2.2, these algorithms do consider every task interval that is a subset of $\mathsf{LCut}(T, j)$, and [22, Proposition 6] shows that the maximal subset will be one of these task intervals. Therefore, the edge-finding algorithm in [39] will update the task bounds using the same subsets that (TEF) would find. □

Taken together, Propositions 2.4 and 2.5 show that the edge-finding algorithm in [39] is monotonic.

Chapter 3

Edge Finding: Implementation

In this chapter, a detailed discussion of a complete implementation for the edge-finder described in [39] is provided. In Section 3.1, the storage requirements of the algorithm are explored, and the pseudo-code of an implementation of that storage is given. In [39], only the algorithm for adjusting earliest start time is discussed, so Section 3.2 provides the symmetric method for adjusting last completion time, along with pseudo-code for a generic algorithm for adjusting both bounds. Finally, in Section 3.3, those parts of the algorithm not explained in [39] are discussed, and some errors in the original paper are noted and corrected.

3.1 Data Structures

3.1.1 Θ-tree

The core of the cumulative edge-finding algorithm in [39] is the data structure called the Θ-tree. Specifically, this is an extended version of the Θ-tree called a Θ-Λ-tree; however, the concept of the Θ-tree is used throughout Vilím's scheduling algorithms (such as disjunctive not-first/not-last, disjunctive detectable precedences [37, 38], disjunctive edge-finding[42, 38], and cumulative max energy filtering[40]), and as such it is worth considering the general attributes and implementation of Θ-tree structures before moving on to the more specific case.

More familiar tree structures, such as heaps and binary search trees, are designed to facilitate sorting or retrieving items based on a key, but Θ-trees perform a different sort of function. As mentioned above, tasks are stored in the leaves of the Θ-tree, sorted by some criterion. This sorting is required for the proper functioning of the tree, but the tree itself does not implement the sorting; the tasks must be sorted before the Θ-tree can be built. The values for all interior nodes must be calculated when the tree is created, but further single node alterations can be handled in $\mathcal{O}(\log n)$ time.

```
   Requires: num, cap > 0
   Requires: order is an array of num tasks sorted by bound
   Ensures : tree is an internally consistent Θ-tree
 1 buildTree(tree, order, num, C)
 2 begin
 3     tree.n ← num
 4     tree.C ← C
 5     for i = tree.size − 1 to tree.size − tree.n do
 6         computeLeafVals(tree, i)
 7     for i = tree.size - tree.n - 1 to 0 do
 8         computeNodeVals(tree, i)
 9 computeLeafVals(tree, i)
10 begin
11     base-Env^C_i ← −∞
12     addToTheta(tree, i)
13     e^Λ_i ← −∞
14     Env^Λ_i ← −∞
15     resp(e^Λ)_i ← i
16     resp(Env^Λ)_i ← i
17 addToTheta(tree, i)
18 begin
19     t ← order[i − (tree.n − 1)]
20     e_i ← e_t
21     Env_i ← tree.C ∗ bound_t + e_t
22     Env^C_i ← base-Env^C_i
23 clearNode(tree, i)                    // Θ ← Θ \ {i}
24 begin
25     e_i ← 0
26     Env_i ← −∞
27     Env^C_i ← −∞
28     e^Λ_i ← −∞
29     Env^Λ_i ← −∞
30 clearTree(tree)
31 begin
32     for i = tree.size - 1 to 0 do
33         clearNode(tree, i)
```

Algorithm 3.1 – buildTree takes a Θ-tree (see Figure 3.2) and builds the internal tree structure. The Θ-tree created will have leaves which correspond to the tasks indicated by order. computeLeafVals and computeNodeVals implement the calculations performed on the specific data stored in this Θ-tree.

```
   Requires: ∃(left(i) ∧ right(i)) ⟹ bound_left(i) ≤ bound_right(i)
 1 computeNodeVals(tree, i)
 2 begin
 3     if left(i) does not exist then
 4         clearNode(tree, i)
 5     else if right(i) does not exist then
 6         tree[i] ← tree[left(i)]
 7     else
 8         e_i ← e_left(i) + e_right(i)
 9         Env_i ← max(Env_left(i) + e_right(i), Env_right(i))
10         Env^c_i ← max(Env^c_left(i) + e_right(i), Env^c_left(i))
11         if e^Λ_left(i) + e_right(i) > e_left(i) + e^Λ_right(i) then
12             e^Λ_i ← e^Λ_left(i) + e_right(i)
13             resp(e^Λ)_i ← resp(e^Λ)_left(i)
14         else
15             e^Λ_i ← e_left(i) + e^Λ_right(i)
16             resp(e^Λ)_i ← resp(e^Λ)_right(i)
17         if Env^Λ_left(i) + e_right(i) > Env_left(i) + e^Λ_right(i) then
18             if Env^Λ_left(i) + e_right(i) > Env^Λ_right(i) then
19                 Env^Λ_i ← Env^Λ_left(i) + e_right(i)
20                 resp(Env^Λ)_i ← resp(Env^Λ)_left(i)
21             else
22                 Env^Λ_i ← Env^Λ_right(i)
23                 resp(Env^Λ)_i ← resp(Env^Λ)_right(i)
24         else
25             if Env_left(i) + e^Λ_right(i) > Env^Λ_right(i) then
26                 Env^Λ_i ← Env_left(i) + e^Λ_right(i)
27                 resp(Env^Λ)_i ← resp(e^Λ)_left(i)
28             else
29                 Env^Λ_i ← Env^Λ_right(i)
30                 resp(Env^Λ)_i ← resp(Env^Λ)_right(i)
```

Algorithm 3.2 – computeNodeVals is used to calculate the values for interior nodes of the Θ-tree. It requires $0 \leq i < n-1$; furthermore, for all nodes $v \in tree$, if $v > n$ then values for v must already be computed.

Storage Design

The Θ-tree may be implemented as any binary tree which meets two minimal criteria: (i) it is balanced; and (ii) it has a time complexity of $\mathcal{O}(\log n)$ for leaf insertion/deletion, and $\mathcal{O}(1)$ for finding the root node [38, Section 2.5.2]. There is, however, a further consideration which influences the choice of implementation, namely the relatively static structure of the Θ-tree. The sort order of the Θ-tree is based on one of the bounds of the tasks, the same bound which is being readjusted by the algorithm (i.e., one uses a Θ-tree sorted by earliest start time to calculate adjusted earliest start time values); it is not necessary to resort the tasks by the adjusted bound, however, as the algorithm is not idempotent—adjustments are made at the end of an iteration, and then

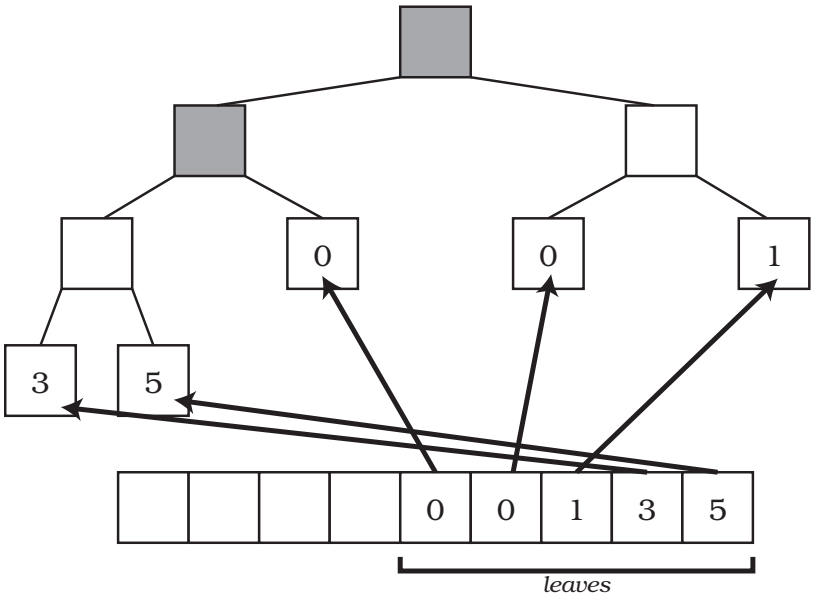


Figure 3.1 – Five tasks, sorted by their earliest start time, in the minimum tree with 5 leaves. The tasks are sorted in the array, but in the tree the array represents the left-to-right sort order of the respective tasks is broken, leading to invalid energy envelope computations in the shaded nodes.

the algorithm is repeated until a fixpoint is reached, with a new Θ-tree on each iteration. Within an iteration, the tree would only need to be rebalanced on node insertion or deletion. This can be altogether avoided by simulating deletions with an "empty" node, leaving the structure of the tree unaltered; insertions simply reverse this procedure[38, Section 2.5.2]. So the use of a self-balancing tree, such as an AVL tree, is unwarranted.

In fact, even the initial balancing of the tree is quite trivial, as both the number and order of the tasks is known when the tree is constructed. For a set of n tasks, the smallest balanced tree has $2n - 1$ nodes, and a depth of $\lceil \log_2(n) \rceil + 1$. There is, therefore, little utility in the choice of a dynamic, "node and pointer" design for the Θ-tree. An array implementation has lower storage overhead, and is well suited for storing such a stable tree. For the node at index i of the array, let:

$$\begin{aligned} \mathsf{parent}(i) &= (i-1)/2 \\ \mathsf{left}(i) &= 2i+1 \\ \mathsf{right}(i) &= 2i+2 \\ \mathsf{sibling}^{\mathsf{l}}(i) &= i-1 \\ \mathsf{sibling}^{\mathsf{r}}(i) &= i+1 \end{aligned}$$

Using the smallest possible tree, every node with $index \geq n - 1$ would be a leaf node. An in-order traversal of leaves would be accomplished by simple iteration over the last n elements of the array. (Traversals of all the nodes in the

tree would be more complicated, but these sorts of traversals are not generally meaningful for Θ-trees, since all the tasks are stored in the leaves.)

Care has to be taken to preserve the left-right sort order of the leaves, however. The equations for updating the values of interior nodes depend on the order of the tasks in the left and right subtrees; for example, [39, equation 5] is valid only if all tasks in the left subtree have a lower est than all tasks in the right subtree. Using the smallest tree with n nodes, and storing the tasks in sorted order in the last n nodes, results in a tree with several interior nodes where at least one leaf in the left subtree corresponds to a node with a greater earliest start time than the nodes in the right subtree for any tree where n is not a power of 2 (see Figure 3.1). To prevent this, it is necessary to keep all the task nodes together at the tree's greatest depth, which in turn introduces some nodes at higher levels which have no tasks in their subtrees. These nodes may be treated in an identical fashion with the "empty" nodes mentioned above. This new condition leads to a minimal tree size of:

$$2^{\lceil \log_2(n) \rceil} + n - 1$$

Also note that for a node n_i (where i is the index of n), we can be certain that $j > i$ for every node n_j in the subtree rooted at n_i. By iterating over all nodes in decreasing order of index, we ensure that we never visit a node without first visiting every node in its subtree. Since the computed values in a Θ-tree node depend solely on the computed values in that node's subtree, we see that we can perform all of the computations in a single such traversal of the tree, which has $\mathcal{O}(n)$ complexity (see Algorithm 3.1).

Θ-tree Operations

One drawback of the generic Θ-tree algorithm is that it is difficult to properly encapsulate the calculations used in tree operations. There are generally two types of computation: values such as e_i, which are computed by summing the left and right subtree values; and Env_i-type values, which are based on a comparison of the corresponding Env_i value in the right subtree with the sum of the left subtree Env_i and the right subtree e_i. In practice, however, there are too many variations in the specifics of calculation to make this a useful generalization. It is possible to define a a minimum set of operations required for any Θ-tree to function, specifically:

- set the values of a single leaf (presumably from a task)
- disable a leaf
- compute the values of an internal node from its left and right children

These functions are provided in the next section, in Algorithm 3.3.

Node Access

In most tree structures, node access is accomplished in one of two methods: either by starting at the root node and tracing a down down path to the node, or performing a traversal of all nodes in the tree. In the Θ-tree, this first method is useful only in limited circumstances (see Section 3.3.3 for an example), as

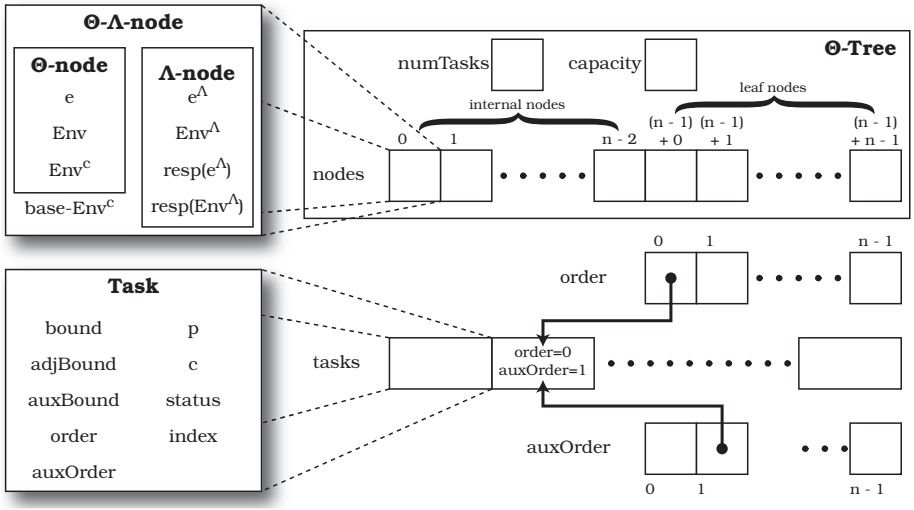


Figure 3.2 – A Θ-tree and its associated lists for a set of n tasks, showing how to navigate from a task in auxOrder to the corresponding node in tree. The contents of the Θ-Λ-node are dealt with in Section 3.1.2, except for $\mathsf{Env}^{\mathsf{c}}$ and $\mathsf{base\text{-}Env}^{\mathsf{c}}$, which are explained in Section 3.3.3.

the calculated values of the interior nodes are not generally useful in locating a specific task. Likewise, ordered traversals of all the Θ-tree nodes are not generally meaningful: tasks are stored only in leaf nodes, while traversing the interior nodes is equivalent to moving through various subsets of these tasks, in an order not required by the algorithm. Instead, the edge finding algorithm iterates through the tasks in sorted order by one or the other of the tasks bounds (earliest start time or last completion time). When this iteration is in the same order as the sorting of the tasks in the Θ-tree itself, it is accomplished by a simple walk through the last n elements of the Θ-tree node array.

Just as often, however, the algorithm in [39] requires an iteration through the tasks stored in the Θ-tree in the order by their *other* bound, i.e. the tasks in a Θ-tree sorted in order by earliest start time must be accessed in either increasing or decreasing order of last completion time. This order cannot be derived from the structure of the tree, so two additional lists are required: an order list that is sorted by the same bound as the tasks in the tree, and an auxOrder list sorted by the other bound. The order list is used to provide the task order during tree construction, but can then also be used for in-order traversals of the tasks. While it is useful to think of the leaves of the Θ-tree as being equivalent to tasks, actually storing the tasks in the leaf nodes is not necessary. It is possible to simplify the Θ-tree structure by using the same nodes for both leaves and interior nodes, storing only the calculated values, while keeping the task data in a separate task list, accessed by means of order and auxOrder.

Since all of these lists are of fixed size, and each needs to be sorted once at most, they can be easily stored in arrays. Between the Θ-tree and the order array there exists an implicit mapping:

$$\mathsf{order}[i] \longleftrightarrow \mathsf{tree}[i + (n-1)]$$

The mapping from auxOrder to order needs to be explicit, however. To begin, note that auxOrder and order are permutations of the set of tasks; it would be redundant to store the same task information in both arrays. Instead, task information will be stored in a tasks array, and order and auxOrder will be arrays of pointers into tasks. In turn, tasks will store not just the task data, but also the indices of that task in both order and auxOrder. By this means it becomes possible to move from an element in auxOrder to the corresponding node in the Θ-tree: the auxOrder points to the relevant task in tasks, which in turn holds the index of the task in order, which can be mapped onto the index of the task in tree (see Figure 3.2). To adjust the other bound, the roles of order and auxOrder are simply reversed, and the tree is rebuilt.

3.1.2 Θ-Λ-tree

The Θ-Λ-tree is an extension of the Θ-tree that allows for the partitioning of the tasks into two sets, Θ and Λ, with the object of quickly calculating both energy envelope and energy for a set of tasks that include all the tasks in Θ and at most one task in Λ. The structure of the tree itself is unchanged, but additional values are computed for each node. Where the nodes of the Θ-tree held only $\mathsf{e}(\Theta)$ and $\mathsf{Env}(\Theta)$, the nodes of the Θ-Λ-tree also hold $\mathsf{e}(\Theta, \Lambda)$ and $\mathsf{Env}(\Theta, \Lambda)$; these new values can include one node from Λ. The complete structure of the Θ-Λ-node is given in Figure 3.2.

Once ${\mathsf{Env}^{\Lambda}}_{root}$ indicates that there exists a task $t \in \Lambda$ which conflicts with the scheduling of Θ, t must be located, the precedence between the t and Θ recorded, and t removed from Λ. In Vilím's algorithm, t is located by checking whether t must have come from the left or right subtree of the root node, moving to the root of that subtree, and repeating until a leaf node is reached. This approach is unnecessarily redundant, however, as the decision being made at each node is essentially reconstructing the choice of a Λ value which occurred during the calculation of e^{Λ} or Env^{Λ}. It is more efficient to record the Λ node responsible for these values at each interior node as the initial calculations are performed [31]; to do so is the purpose of the $\mathsf{resp}(\mathsf{e}^{\Lambda})$ and $\mathsf{resp}(\mathsf{Env}^{\Lambda})$ data elements in the Λ-node structure.

The Θ-Λ-tree introduces new operations. Adding and removing nodes from the set Θ are accomplished through the addToTheta and clearNode functions provided in Algorithm 3.3. Additionally, it must now be possible to move a node from Θ to Λ, and to remove a node from Λ. Note that it is not necessary to have a separate operation for adding a disabled node to Λ, as the operation is never performed; neither is there any need to move a node from Λ to Θ. These new procedures are presented as Algorithm 3.3.

3.1.3 update Storage

The adjustment phase presents another data structure issue. The adjustment of the bound for task i relies on the calculation of an update value based on both c_i and the task j, where [39, equation EF2]

$$\mathsf{LCut}(T, j) \lessdot i \implies \mathsf{est}'_i \leftarrow \max\{\mathsf{update}(j, c_i), \mathsf{est}_i\}$$

Since $\mathsf{update}(j, c_i)$ does not depend on any other computations in the detection or adjustment phases, it is possible to precompute the values for all j and c

```
   Requires: tree is an internally consistent Θ-tree
   Requires: i is a leaf node
 1 enableNode(tree, i)                          // Θ ← Θ ∪ {i}
 2 begin
 3     addToTheta(tree, i)
 4     updateTree(tree, parent(i))
 5 disableNode(tree, i)
 6 begin
 7     clearNode(tree, i)
 8     updateTree(tree, i)
 9 moveToLambda(tree, i)                  // Θ ← Θ \ {i}, Λ ← Λ ∪ {i}
10 begin
11     e^Λ_i ← e_i
12     Env^Λ_i ← Env_i
13     e_i ← 0
14     Env_i ← −∞
15     Env^c_i ← −∞
16     updateTree(tree, parent(i))
17 removeFromLambda(tree, i)                   // Λ ← Λ \ {i}
18 begin
19     e^Λ_i ← −∞
20     Env^Λ_i ← −∞
21     updateTree(tree, parent(i))
   Requires: i is a node
   Ensures : tree is an internally consistent Θ-tree
22 updateTree(tree, i)
23 begin
24     while i ≥ 0 do
25         computeNodeVals(tree, i)
26         i ← parent(i)
```

Algorithm 3.3 – enableNode and disableNode are used to add or remove a node from the set Θ, while moveToLambda and removeFromLambda add or remove a task from Λ. All must be invoked on a leaf node ($n-1 \leq i < 2n-1$); after changing the values in node i, each invokes updateTree, which performs a leaf-to-root traversal to update the values of the affected internal nodes.

before proceeding with the adjustment phase, as discussed in section 3.3.4; the results of the precomputation then have a space complexity of $\mathcal{O}(kn)$, where k is the number of unique values of c in the tasks. These values may be stored in a $k \times n$ array, but c is not suitable for use as a numeric index (as the values of c are in no way constrained to be contiguous).

Instead, c will be used as a key to hash into a table of k entries, each corresponding to an array of size j. For each task i, $\mathsf{update}(j, c_i)$ will be computed for every task j, and the array of values added to the table. For subsequent tasks, the hash can be used to quickly determine if $\mathsf{update}(j, c_i)$ has already been computed—a determination made in constant time in the best case, and linear time in the worst case. For new values of c_i, the computation is repeated

```
 1 updateNew(n) begin
 2     new update
 3     update.n = n
 4     update.hashTable = hashAllocate(n);
 5 updateFree(update) begin
 6     hashFree(update.hashtable)
 7 updateContains(upd, c) begin
 8     return hashContains(update.hashtable, c)
 9 updateGet(update, j, c) begin
10     if hashContains(update.hashtable, c) then
11         arrayPtr := hashGet(update.hashtable, c)
12         return arrayPtr[j]
13     return-1;
14 updateSet(update, j, c, val) begin
15     if hashContains(update.hashtable, c) then
16         arrayPtr := hashGet(update.hashtable, c)
17     else
18         arrayPtr := allocate(integer array of size update.n)
19         hashAdd(update.hashtable, c, arrayPtr)
20     arrayPtr[j] = val
```

Algorithm 3.4 – Operations in the data structure update, which stores the values update(j, c) for all j and c. updateContains checks only that the passed value of c has been seen before; it is assumed that when unseen values of c are encountered, update(j, c) is computed and stored for all j.

and the new values added to the table.

The minimal interface required by this data structure is presented as Algorithm 3.4.

3.2 Strengthening of Last Completion Time

Edge-finding is a member of a family of task-based scheduling propagators, which utilize symmetric algorithms to update earliest start time and last completion time for each task; as is generally true in demonstrations of these algorithms, Vilím provides only the earliest start time case. In order to properly handle this symmetry with a minimum of redundancy, the implementation will have to handle two sorts of reflection: the equations will have to be symmetric, and the sort orders of the bounds will need to be reversed. For the adjustment of earliest start time, the Θ-tree is built from a list of tasks sorted by increasing earliest start time, and another list sorted by decreasing last completion time is used in the algorithm; for the adjustment of last completion time, a tree must be built that is sorted in decreasing order of last completion time, and then a list of tasks in increasing earliest start time order will be used in the adjustment. The equations that make the tree work, however, have an inherent chirality: they treat the left and right subtrees differently, and this relationship must be maintained in the symmetric tree. Our intuition is that the algorithm can realize both parts of the symmetry by switching the two ordered lists of

bounds, and then negating their contents. In fact, it is demonstrated in [35] that this negation appropriately models the symmetry of all similar task-based scheduling algorithms. As Vilím's algorithm depends on a novel data structure, however, it seems worthwhile to verify that this method will produce the desired results through a derivation of the equations for a Θ-Λ-tree with tasks sorted in increasing order of negated last completion time. Along the way, we will generalize Vilím's detection [39, Algorithm 1] and adjustment [39, Algorithms 2, 3] algorithms, so that they may be used equally for each bound.

In previous work, Vilím gives these earliest start time and last completion time adjustment rules for edge finding with a unary resource, which point the way to similar rules for cumulative resource edge-finding [38, equations 2.3, 2.4]:

$$\forall \Omega \subset T,\, \forall i \in (T \setminus \Omega) : \\ (\mathsf{est}_{\Omega\cup\{i\}} + p_{\Omega\cup\{i\}} > \mathsf{lct}_\Omega \implies \Omega \ll i \implies \mathsf{est}_i \leftarrow \max\{\mathsf{est}_i, \mathsf{Ect}_\Omega\})$$

$$\forall \Omega \subset T,\, \forall i \in (T \setminus \Omega) : \\ (\mathsf{lct}_{\Omega\cup\{i\}} - p_{\Omega\cup\{i\}} < \mathsf{est}_\Omega \implies i \ll \Omega \implies \mathsf{lct}_i \leftarrow \min\{\mathsf{lct}_i, \mathsf{Lst}_\Omega\}) \quad (3.1)$$

The cumulative resource case is more complex, but can be broken down into two phases: a detection phase (homologous to the first implication in (3.1)) and an adjustment phase (homologous to the the second implication in (3.1)). We will start by investigating the cumulative resource detection phase.

3.2.1 Detection

For a group of activities, we compute an upper bound of the group's last start time as:

$$\mathsf{preLst}(\Theta) = \mathsf{lct}_\Theta - \left\lceil \frac{\mathsf{e}_\Theta}{\mathcal{C}} \right\rceil$$

where:

$$\mathsf{lct}_\Theta = \max_{i\in\Theta}\{lct_i\}$$
$$\mathsf{e}_\Theta = \sum_{i\in\Theta} \mathsf{e}_i$$

For a better estimate of last start time we use:

$$\begin{aligned} \mathsf{Lst}(\Theta) &= \min_{\Omega\subseteq\Theta}\{\mathsf{preLst}(\Omega)\} \\ &= \min_{\Omega\subseteq\Theta}\left\{\mathsf{lct}_\Omega - \left\lceil \frac{\mathsf{e}_\Omega}{\mathcal{C}} \right\rceil\right\} \\ &= \left\lceil \frac{\min_{\Omega\in\Theta}\{\mathcal{C} * \mathsf{lct}_\Omega - \mathsf{e}_\Omega\}}{\mathcal{C}} \right\rceil \\ &= \left\lceil \frac{\mathsf{Env}'(\Theta)}{\mathcal{C}} \right\rceil \end{aligned}$$

where (compare to [40, (3)]):

$$\mathsf{Env}'(\Theta) = \min_{\Omega\subseteq\Theta}\{\mathcal{C} * \mathsf{lct}_\Omega - \mathsf{e}_\Omega\} \quad (3.2)$$

Negating (3.2) gives us:

$$\begin{aligned} -\mathsf{Env}'(\Theta) &= -1 * \left(\min_{\Omega \subseteq \Theta} \{\mathcal{C} * \mathsf{lct}_\Omega - \mathsf{e}_\Omega\} \right) \\ &= \max_{\Omega \subseteq \Theta} \{-1 * (\mathcal{C} * \mathsf{lct}_\Omega - \mathsf{e}_\Omega)\} \\ &= \max_{\Omega \subseteq \Theta} \{\mathcal{C} * (-\mathsf{lct}_\Omega) + \mathsf{e}_\Omega\} \end{aligned} \tag{3.3}$$

(3.3) is very nearly the same as the equation used by Vilím to set up the Θ-tree for calculating *est* [39, equations 1 and 5]. Calculation of $-\mathsf{Env}'_\Theta$ with a $-\mathsf{lct}_v$ sorted Θ-tree will have to use the formula:

$$-\mathsf{Env}'_v = \begin{cases} \mathcal{C} * (-\mathsf{lct}_v) + \mathsf{e}_v & \text{if } v \text{ is a leaf node} \\ \max\{-\mathsf{Env}'_{left(v)} + \mathsf{e}_{right(v)}, -\mathsf{Env}'_{right(v)}\} & \text{if } v \text{ is an internal node} \end{cases}$$

For leaf nodes, this equation will obviously hold; for internal nodes, we will demonstrate that the formula is correct by following [40, Proposition 2]:

Proposition 3.1. *For an internal node v, $-\mathsf{Env}'_v$ can be computed using the recursive formula:*

$$-\mathsf{Env}'_v = \max\{-\mathsf{Env}'_{left(v)} + \mathsf{e}_{right(v)}, -\mathsf{Env}'_{right(v)}\} \tag{3.4}$$

Proof. By equation (3.3), we have:

$$-\mathsf{Env}'_v = \max_{\Omega \subseteq \mathsf{Leaves}(v)} \{\mathsf{Env}'_{\mathsf{left}(v)} + \mathsf{e}_{rnode(v)}, \mathsf{Env}'_{\mathsf{right}(v)}\}$$

There are two cases to consider:

1. $\mathsf{Left}(v) \cap \Omega = \emptyset$, so $\Omega \subseteq \mathsf{Right}(v)$. Clearly,

$$\max_{\Omega \subseteq \mathsf{Leaves}(v)} \{\mathcal{C} * (-\mathsf{lct}_v) + \mathsf{e}_v\} = -\mathsf{Env}'(\mathsf{Right}(v)) = -\mathsf{Env}'_{\mathsf{right}(v)}$$

2. $\mathsf{Left}(v) \cap \Omega \neq \emptyset$ Let S be the set of all possible Ω considered in this case:

$$S = \{\Omega, \Omega \subseteq \Theta \ \&\ \Omega \cap \mathsf{Left}(v) \neq \emptyset\}$$

By definition, $\mathsf{lct}_\Omega = \max_{i \in \Omega}\{\mathsf{lct}_i\}$, so $-\mathsf{lct}_\Omega = \min_{i \in \Omega}\{-\mathsf{lct}_i\}$. As leaf nodes in this Θ-tree are sorted by $-\mathsf{lct}_i$, the minimum $-\mathsf{lct}_i$ in Ω must be in the left subtree, and $-\mathsf{lct}_\Omega = -\mathsf{lct}_{\Omega \cap \mathsf{Left}(v)}$. Utilizing this fact, and breaking e_Ω into its left and right components, we get:

$$\max_{\Omega \in S}\{\mathcal{C} * (-\mathsf{lct}_\Omega) + \mathsf{e}_\Omega\} = \max_{\Omega \in S}\{\mathcal{C} * (-\mathsf{lct}_{\Omega \cap \mathsf{Left}(v)}) + \mathsf{e}_{\Omega \cap \mathsf{Left}(v)} + \mathsf{e}_{\Omega \cap \mathsf{Right}(v)}\}$$

The maximum is achieved only by an Ω for which $\mathsf{Right}(v) \subsetneq \Omega$, giving us:

$$= \max_{\Omega \in S}\{\mathcal{C} * (-\mathsf{lct}_{\Omega \cap \mathsf{Left}(v)}) + \mathsf{e}_{\Omega \cap \mathsf{Left}(v)}\} + \mathsf{e}_{\mathsf{Right}(v)}$$

As $\Omega \cap \mathsf{Left}(v)$ must enumerate all possible subsets of $\mathsf{Left}(v)$, the first term reduces thus:

$$= \mathsf{Env}_{\mathsf{left}(v)} + \mathsf{e}_{\mathsf{Right}(v)}$$

```
1  DETECTORDER(tree, order, auxOrder, prec, n, C)
2  begin
3      for i = 0 to n do
4          prec[i] ← −∞
5      for j = 0 to n - 1 do
6          t ← auxOrder[j]
7          if Env_root > C ∗ auxBound_t then
8              FAIL                // resource is overloaded
9          while Env^Λ_root > C ∗ auxBound_t do
10             i ← resp(Env^Λ)_root
11             prec[i − (n − 1)] ← auxBound_t
12             REMOVEFROMLAMBDA(tree, i)
13         MOVETOLAMBDA(tree, rank_t +n)
14     for i = 0 to n- 1 do
15         t ← order[i]
16         if auxBound_t! = bound_t +p_t then
17             prec[i] ← max(prec[i], bound_t +p_t)
```

Algorithm 3.5 – DETECTORDER takes a data set and determines a partial ordering of its tasks. This order is stored in the array prec, where $\mathsf{prec}[i] = \mathsf{lct}_j$ means that task_i must be preceded by the set $\mathsf{LCut}(T, j)$ (see [39, Section 6]); alternately, $\mathsf{prec}[i] = \mathsf{est}_j$ means that $task_i$ must precede the set $\mathsf{LCut}'(T, j)$ (see (3.5)).

Taken together, the results of cases 1 and 2 prove (3.4). □

As hoped, we conclude that a Θ-tree storing $-\mathsf{lct}$ in place of earliest start time may be used to calculate $-\mathsf{Env}'(\Theta)$; it can readily be seen that $-\mathsf{Env}'(\Theta, \Lambda)$ may be calculated in the same manner.

We also need a symmetric version of the set $\mathsf{LCut}(T, j)$[39, Section 5], which we will define as:

$$\mathsf{LCut}'(T, j) = \{l, l \in T \ \&\ \mathsf{est}_l \geq \mathsf{est}_j\}$$

We can now modify Vilím's *est* detection rule for cumulative resources [39, Section 5] in the same way that the rule for unary resources(3.1) was modified to get the rule for *lct*, giving:

$$\forall j \in T, \forall i \in (T \setminus \mathsf{LCut}'(T, j)) : \\ \mathsf{Lst}(\mathsf{LCut}'(T, j) \cup \{i\}) < \mathsf{est}_j \implies \mathsf{LCut}'(T, j) \gg i \quad (3.5)$$

where $\gg$ denotes the relationship "begins after beginning"—in this case, that in all solutions, $\mathsf{LCut}(T, j)$ begins after the beginning of i. For the meaning of this rule, we can paraphrase Vilím's explanation of the *est* detection rule: the set $\mathsf{LCut}'(T, j)$ consists of tasks which must start after the time est_j; if there is not enough time to process this set and i after that time, then i must begin before $\mathsf{LCut}'(T, j)$. Using the definition of Env' above, we can rewrite this rule as (compare to [39, (EF1)]):

$$\forall j \in T, \forall i \in (T \setminus \mathsf{LCut}'(T, j)) : \\ \mathsf{Env}'(\mathsf{LCut}'(T, j) \cup \{i\}) < \mathcal{C} * \mathsf{est}_j \implies \mathsf{LCut}'(T, j) \gg i \quad (3.6)$$

Once again, in order to make the *lct* and *est* detection phases more symmetric, we can rewrite (3.6) as:

$$\forall j \in T,\, \forall i \in (T \setminus \mathsf{LCut}(T,j)) : \\ -\mathsf{Env}'(\mathsf{LCut}(T,j) \cup \{i\}) > \mathcal{C} * (-\mathsf{est}_j) \implies \mathsf{LCut}(T,j) \gg i$$

With our Θ-tree storing $-lct$ we can already calculate $-\mathsf{Env}'(\mathsf{LCut}'(T,j) \cup \{i\})$, so all that remains to generalize the detection algorithm is to use $-\mathsf{est}_j$ in the comparison here. So, for the detection phase at least, we can define a single algorithm for adjusting earliest start time and last completion time, if we simply negate both bounds for each task during the last completion time detection case, presented here as Algorithm 3.5. (The substitution of LCut' for LCut creates another difference: during the last completion time case, the set Λ is built by adding tasks in non-decreasing order by earliest start time, instead of non-increasing order by last completion time. We already have a list of tasks sorted in non-decreasing earliest start time order however, which was used to build the earliest start time sorted Θ-tree; so the two ordered sets can simply be switched when changing to last completion time bound adjustment).

3.2.2 Adjustment

For the lct adjustment phase, we need to find an equivalent of the second implication in (3.1):

$$i \ll \Omega \implies \mathsf{lct}_i \leftarrow \min\{\mathsf{lct}_i, \mathsf{Lst}_\Omega\}$$

The relationship $\mathsf{LCut}'(T,j) \gg i$ which we detected in the previous phase is a much weaker relationship than $i \ll \Omega$, due to the fact that with cumulative resources the processing times of activities can overlap.

The edge finding rule for updating bounds was discussed in Section 2.1. Having detected that $\mathsf{LCut}'(T,j)$ must begin after the beginning of i, we now need to find sets $\omega \subseteq \mathsf{LCut}'(T,j)$ that have enough energy to prevent the simultaneous scheduling of i, in other words where $\mathsf{e}_\omega > (\mathcal{C} - c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega)$[39, Section 7]. Equation (2.2) gives the amount of energy from ω which interferes with the scheduling of i, so we can now update lct_i based on this formula (compare to [39, (EF2)]):

$$\mathsf{lct}_i = \min\{\mathsf{update}'(j, c_i), \mathsf{lct}_i\} \tag{3.7}$$

$$\mathsf{update}'(j,c_i) = \min_{\substack{\omega \subseteq \mathsf{LCut}'(T,j) \\ \mathsf{e}_\omega > (\mathcal{C}-c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}} \left\{ \mathsf{lct}_\omega - \left\lceil \frac{\mathsf{e}_\omega - (\mathcal{C} - c_i)(\mathsf{lct}_\omega - \mathsf{est}_\omega)}{c_i} \right\rceil \right\}$$

To calculate all values of update we would have to iterate through every subset of $\mathsf{LCut}'(T,j)$ for every $j \in t$; fortunately, we can skip many of these subsets. By choosing j from T in order of decreasing est_i, we have

$$\mathsf{est}_{j_1} \geq \mathsf{est}_{j_2} \geq \mathsf{est}_{j_3} \geq \ldots \mathsf{est}_{j_n} \\ \mathsf{LCut}'(T,j_1) \subseteq \mathsf{LCut}'(T,j_2) \subseteq \mathsf{LCut}'(T,j_3) \subseteq \ldots \mathsf{LCut}'(T,j_n)$$

So for any given j, we have already computed the update value for some of the subsets of $\mathsf{LCut}'(T,j)$, namely all those subsets which have an earliest start

```
   Requires: ∀i, j ∈ T : LCut(T, j) ≪ i ⟹ prec[i] ≤ lct_j
 1 ADJUSTBOUNDS(tree, order, auxOrder, adjBound, prec, n, cap);
 2 begin
 3     update:= UPDATENEW(n);
 4     for i ∈ auxOrder do
 5         for j ∈ auxOrder, j < i do
 6             if auxBound_j ≤ prec[order_i] then
 7                 if !UPDATECONTAINS(update, c_i) then
 8                     COMPUTEENV^C(tree, c_i);
 9                     CLEARTREE(tree);
10                     upd := −∞;
11                     for l ∈ auxOrder do // by non-decreasing auxBound_l
12                         ENABLENODE(tree, order_l);
13                         Env(l, c_i) ← CALCENVJC(tree, auxBound_l, c_i) ;
14                         diff ← ⌈(Env(l, c_i) − (capacity − c_i) ∗ auxBound_l) / c_i⌉;
15                         upd ← max(upd, diff);
16                         UPDATESET(update, j, c_i, upd);
17                 adjBound_i ← UPDATEGET(update, j, c_i);
18                 break;
19     UPDATEFREE(update);
```

Algorithm 3.6 – ADJUSTBOUNDS(tree, order, auxOrder, adjBound, prec, n, cap) uses the prec array to locate candidates for bounds adjustment. The adjustment of i is based on update(j, c_i); to eliminate extra computations, update is computed only for values of c_i that might actually result in an adjustment.

time greater than est_j. Reusing those earlier computations gives us the following formula for update' (compare to [39, (10)]):

$$\mathsf{update}'(j_l, c) = \begin{cases} \mathsf{diff}'(j_1, c) & \text{when } l = 1 \\ \min\{\mathsf{update}'(j_{l-1}, c),\ \mathsf{diff}'(j_l, c)\} & \text{when } l > 1 \end{cases} \tag{3.8}$$

where (compare to [39, (11)])

$$\mathsf{diff}'(j, c) = \min_{\substack{\Omega \subset \mathsf{LCut}'(T,j) \\ \mathsf{e}_\Omega > (\mathcal{C}-c)(\mathsf{lct}_\Omega - \mathsf{est}_j)}} \left\{ \mathsf{lct}_\Omega - \left\lceil \frac{\mathsf{e}_\Omega - (\mathcal{C} - c)(\mathsf{lct}_\Omega - \mathsf{est}_j)}{c} \right\rceil \right\} \tag{3.9}$$

This still leaves all the sets $\Omega \subseteq \mathsf{LCut}'(T, j)$ that have $\mathsf{est}_\Omega = \mathsf{est}_j$, none of which have been considered before, and any of which might generate a new maximum update. To efficiently find the strongest update provided by any of these subsets, we can first divide the tasks in Ω into two sets, based on a comparison of their last completion time with: [1]

$$\mathsf{minlct}(j, c) = \min \left\{ \mathsf{lct}_\Omega, \Omega \subseteq \mathsf{LCut}'(T, j) \wedge \mathsf{e}_\Omega > (\mathcal{C} - c)(\mathsf{lct}_\Omega - \mathsf{est}_j) \right\} \tag{3.10}$$

[1] Any reader following the parallel explanation in [39] is warned once more that the derivation which follows here differs from that provided in the original. A justification of this difference, along with a more thorough explanation of the intent of this portion of the algorithm, may be found in Section 3.3.2.

There must exist a set Ω_m for which $\mathsf{lct}_{\Omega_m} = \mathsf{minlct}(j,c)$, so we have:

$$\mathsf{diff}'(j,c) \leq \mathsf{lct}_{\Omega_m} - \left\lceil \frac{\mathsf{e}_{\Omega_m} - (\mathcal{C}-c)(\mathsf{lct}_{\Omega_m} - \mathsf{est}_j)}{c} \right\rceil < \mathsf{lct}_{\Omega_m} = \mathsf{minlct}(j,c)$$

and therefore $\mathsf{diff}'(j,c) < \mathsf{minlct}(j,c)$.

There may be sets with $\mathsf{e}_\Omega \leq (\mathcal{C}-c)(\mathsf{lct}_\Omega - \mathsf{est}_j)$ (that are, therefore, not considered in (3.9)) for which $\mathsf{lct}_\Omega > \mathsf{minlct}(j,c)$. For these new sets:

$$\mathsf{lct}_\Omega - \left\lceil \frac{\mathsf{e}_\Omega - (\mathcal{C}-c)(\mathsf{lct}_\Omega - \mathsf{est}_j)}{c} \right\rceil \geq \mathsf{lct}_\Omega \geq \mathsf{minlct}(j,c)$$

Since $\mathsf{diff}'(j,c) < \mathsf{minlct}(j,c)$, these new sets could not influence the minimum in (3.9) if they were included in the calculation. Therefore we can modify (3.9) to get (compare to [39, (12)]):

$$\mathsf{diff}'(j,c) = \min_{\substack{\Omega \subset \mathsf{LCut}'(T,j) \\ \mathsf{lct}_\Omega > \mathsf{minlct}(j,c)}} \left\{ \mathsf{lct}_\Omega - \left\lceil \frac{\mathsf{e}_\Omega - (\mathcal{C}-c)(\mathsf{lct}_\Omega - \mathsf{est}_j)}{c} \right\rceil \right\} \tag{3.11}$$

Formula (3.11) may be restated as (compare to [39, (13)]):

$$\mathsf{diff}'(j,c) = \left\lceil \frac{\mathsf{Env}'(j,c) - (\mathcal{C}-c)\,\mathsf{est}_j}{c} \right\rceil \tag{3.12}$$

where (compare to [39, (14)]):

$$\mathsf{Env}'(j,c) = \min_{\substack{\Omega \subset \mathsf{LCut}'(T,j) \\ \mathsf{e}_\Omega > (\mathcal{C}-c)(\mathsf{lct}_\Omega - \mathsf{est}_j)}} \{\mathcal{C} * \mathsf{lct}_\Omega - \mathsf{e}_\Omega\} \tag{3.13}$$

Once again, so that the algorithm may take advantage of the symmetry of the problem, we can restate equations (3.7), (3.8), (3.12) and (3.13) as:

$$\begin{aligned}
-\mathsf{lct}_i &= \max\{-\mathsf{update}'(j,c_i), -\mathsf{lct}_i\} \\
\mathsf{update}'(j_l,c)- &= \begin{cases} -\mathsf{diff}'(j_1,c) & \text{when } l=1 \\ \max\{-\mathsf{update}'(j_{l-1},c),\ -\mathsf{diff}'(j_l,c)\} & \text{when } l>1 \end{cases} \\
-\mathsf{diff}'(j,c) &= \left\lceil \frac{-\mathsf{Env}'(j,c) - (\mathcal{C}-c)(-\mathsf{est}_j)}{c} \right\rceil \\
-\mathsf{Env}'(j,c) &= \max_{\substack{\Omega \subset \mathsf{LCut}'(T,j) \\ \mathsf{e}_\Omega > (\mathcal{C}-c)(\mathsf{lct}_\Omega - \mathsf{est}_j)}} \{-\mathcal{C} * \mathsf{lct}_\Omega + \mathsf{e}_\Omega\}
\end{aligned} \tag{3.14}$$

So once more, a Θ-tree of negated *lct* values and a list of negated *est* values will allow us to use our *est* update algorithm for adjusting *lct* as well; a fully symmetric adjustment is demonstrated in Algorithm 3.6. The calculation of $\mathsf{Env}(j,c)$ on line 13 is performed in Algorithm 3.8, which must therefore also be symmetric; this symmetry is demonstrated in Section 24. Before proceeding, it is worth noting the formula on line 14, which differs from the equation given by Vilím in the sign of one term [39, Algorithm 3, line 9]. The equation as given here agrees with (3.14); an examination of [39, equation 13] will show that the algorithm as written was in error.

The switching of the two sorted lists also requires an update to the recorded indices in each task. This is handled in the outer loop of the edge-finding algorithm, as is the negation of the bounds for last completion time adjustment, presented as Algorithm 3.7.

```
1  edgeFind(taskList, numTasks, C) begin
2      i ← 0
3      for t ∈ taskList do
4          if status_t ≠ disabled then
5              index_i ← index_t
6              p_i ← p_t
7              c_i ← c_t
8              i ← i + 1
9      prec ← allocate(integer array of size i)
10     estList ← sortByEst(taskList)     // tasks by increasing est
11     lctList ← sortByLct(taskList)     // tasks by decreasing lct
12     for t ∈ tasks do                                  // est adjustment
13         bound_i ← est_t
14         if status_t = enabled then
15             auxBound_i ← lct_t
16         else if status_t = optional then
17             auxBound_i ← ∞
18     buildTree(tree, estList, i, C)
19     detectOrder(tree, estList, lctList, prec, n, C)
20     adjustBounds(tree, estList, lctList, adjBound, prec, n, cap)
21     for i = 0 to numTasks do
22         syncLB(index_i, adjBound_i)
23     for t ∈ tasks do                                  // lct adjustment
24         bound_i ← − lct_t
25         if status_t = enabled then
26             auxBound_i ← − est_t
27         else if status_t = optional then
28             auxBound_i ← −∞
29     buildTree(tree, lctList, i, capacity)
30     detectOrder(tree, lctList, estList, prec, i, C)
31     adjustBounds(tree, lctList, estList, adjBound, prec, i, C)
32     for i = 0 to numTasks do
33         syncUB(index_i, − adjBound_i)
```

Algorithm 3.7 – edgeFind takes an external list of tasks. It then performs edge finding on the upper and lower bounds of the task intervals, and calls the external functions syncUB and syncLB to adjust these bounds for the tasks. setOrder is a helper function; it takes the two sorted lists, attaches them to their proper elements, and sets the index elements of the individual tasks to point back into these lists.

3.3 Clarifications and Corrections

3.3.1 Improving Detection

[39, section 6.2] contains a simple improvement to the detection algorithm. The problem is that both Env and Ect are merely ways of estimating the upper bound of the completion time of a set of tasks, chosen based on their relative ease of computation, while the actual earliest completion time of a set of tasks remains too difficult to calculate. This sort of estimation is bound to limit the ability of edge-finding to recognize some precedences (for an example of an easily recognizable precedence which edge finding cannot detect, see [39, fig. 4]). The result is that the relationship '$\ll$' is known only partially. Knowledge of the relationship may be improved by considering the earliest completion time of a task:

$$\mathsf{ect}_i = \mathsf{est}_i + p_i$$

Clearly, a set of tasks Ω that must end by ect_i would meet the definition of the '$\ll$' relationship: i cannot end before ect_i, so i must end after the end of Ω. Inclusion of this improvement, however, introduces some complications for the implementation; to see why this is, we first consider the meaning of the array prec in the algorithm.

prec Array

In [39, Section 6], the array prec is defined as:

$$\forall i \in T : \{l, l \in T \& \mathsf{lct}_l \leq \mathsf{prec}[i]\} \ll i$$

In other words, when adjusting earliest start time, i ends after the end of the set of all tasks that end on or before the time $\mathsf{prec}[i]$. In [39, Section 5] this set was given another name: $\mathsf{LCut}(T, j)$ (2.7). So we can restate the definition of prec as:

$$\mathsf{prec}[i] \geq \mathsf{lct}_j \iff \mathsf{LCut}(T, j) \ll i \tag{3.15}$$

In the algorithm, prec is used to store the precedences detected during the detection phase (see [39, Algorithm 1, line 8], where $\mathsf{prec}[i]$ is made to equal lct_j, reflecting the detection of a set $\mathsf{LCut}(T, j) \ll i$); these precedences are then retrieved during the adjustment phase.

The improved detection algorithm in [39, Section 6.2] captures the strengthened precedence relationship outline above by updating the prec array thus:

$$\mathsf{prec}[i] \leftarrow \max\{\mathsf{prec}[i], \mathsf{est}_i + p_i\} \tag{3.16}$$

Incorrect Strengthening of Fully Constrained Tasks

Consider once more the example [39, Figure 1], repeated here as Figure 3.3. The unimproved detection algorithm will not detect any interesting subsets which must precede activity B, so after the detection phase $\mathsf{prec}[B] = -\infty$. The application of (3.16) will result in $\mathsf{prec}[B] = \mathsf{est}_B + p_B = 5$. But $lct_B = 5 = \mathsf{prec}[B]$, so by (3.15) we have $LCut(T, B) \ll B$, which means that B must end after the end of B! As a result, est_B is updated to 3, which causes B to be overloaded, and the solution fails. This problem only arises when $\mathsf{ect}_i = \mathsf{lct}_i$; in other words, when i is fully constrained.

Vilím points out that this situation is similar to that encountered in the unary detectable precedences case [38, Section 2.5.4]. In that case, the author has explicitly assumed that the activity i might be included in the preceding set Ω; when est_i is updated there, it is updated against $\mathsf{Ect}_{\Omega\setminus\{i\}}$. Clearly a similar approach must be taken with this improved edge finding detection algorithm, but not an identical one: in the edge finding algorithm, est_i is adjusted based on the value of $\mathsf{update}(j, c_i)$ which is calculated in advance for all $j \in T$ and all $c_i, i \in T$ [39, Section 7]. It is not possible to remove i when making this calculation, because i is not known at the time (as c_i is in no way guaranteed to be unique for each i).

Fortunately it is not necessary, as the unimproved edge finding detection algorithm [39, Algorithm 1] will never set $\mathsf{prec}[i] = \mathsf{lct}_i$, as long as overload checking is being performed in the detection loop (see Section 3.3.5):

Proposition 3.2. *In the unimproved edge finding detection algorithm with overload checking (Algorithm 3.5):*

$$\forall i \in T : \mathsf{prec}[i] \neq \mathsf{lct}_i$$

Proof. lct_i is either unique or it is not. If it is unique, then since tasks are moved from Θ to Λ in decreasing order of last completion time:

$$i \in \Lambda \implies \mathsf{lct}_\Theta < \mathsf{lct}_i \implies \mathsf{prec}[i] < \mathsf{lct}_i$$

If lct_i is not unique, then $\exists j \in T$ such that $\mathsf{lct}_i = \mathsf{lct}_j$. Either i or j will have to be moved into Λ first. Assume that i is the first moved (the argument holds also if j is moved first), and let Θ_1 be Θ before i is moved, while Θ_2 is Θ after the move. It is then possible that $\mathsf{Env}(\Theta_2, \Lambda) > \mathcal{C}\,\mathsf{lct}_j$ (line 9), where $\mathsf{Env}(\Theta_2, \Lambda) = \mathsf{Env}(\Theta_2 \cup \{x\}$ for some $x \in \Lambda$. But if $x = i$, then

$$\mathsf{Env}(\Theta_1) = \mathsf{Env}(\Theta_2 \cup \{i\} = \mathsf{Env}(\Theta_2, \Lambda)$$

However, the inequality $\mathsf{Env}(\Theta_1) > \mathcal{C}\,\mathsf{lct}_i$ (which is equivalent, since $\mathsf{lct}_i = \mathsf{lct}_j$) has already been evaluated on line 8; if this inequality holds, the resource is overloaded, and the edge finding algorithm fails. Therefore, for i and j, either the inequality on line 9 will not be reached, or it will be false. Either of these cases will prevent the algorithm from reaching the assignment of $\mathsf{prec}[i] = \mathsf{lct}_j$ on line 11. □

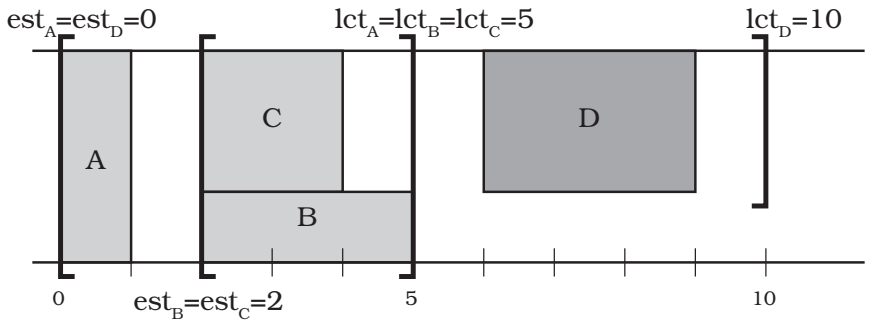


Figure 3.3 – Simple cumulative scheduling problem, revisited. Redrawn from [39, Figure 1].

Therefore, the condition $\mathsf{prec}[i] = \mathsf{lct}_i$ can only be caused by the improvement to the algorithm, in which case it can safely be ignored (see Algorithm 3.5, line 16).

Duplicate Bounds

In the adjustment phase of the algorithm, update of the bounds depends on [39, equation EF2]:

$$\mathsf{LCut}(T, j) \lessdot i \implies \mathsf{est}'_i \leftarrow \max\{\mathsf{update}(j, c_i), \mathsf{est}_i\}$$

Knowledge of the relationship $\mathsf{LCut}(T, j) \lessdot i$ is stored in the array prec, as discussed above. If duplicates in lct_j could be ignored, j could be found simply by iterating through all tasks to find one where $\mathsf{lct}_j = \mathsf{prec}[i]$; with duplicates, care must be taken to find the correct task that meets this equality. In [39, Algorithm 3] tasks are added to Θ in non-decreasing order by lct_j, but this order is arbitrary with duplicate values of lct_j. In this implementation, the order is determined by the sorting of the tasks in the $\mathsf{auxOrder}$ array, which is constant through an iteration, thereby making selection of the correct task possible, as follows.

The calculation of update is non-decreasing; as new tasks are added to Θ and update is recalculated, the new values are taken only if they represent an increase over previously seen values. Abstractly, update is dependent on the energy of the tasks in Θ; the addition of new tasks increases this energy, regardless of whether the tasks have duplicate values for last completion time. For duplicates, the algorithm needs only to ensure that the task that would have been last added to update is the task used in the adjustment procedure. For the addition of tasks, $\mathsf{auxOrder}$ is traversed from the last to the first element; if the search for the correct j begins with the first element, it may correctly terminate when the first matching element is located.

If the improved detection algorithm is used, it is possible that:

$$\exists i \in T : \forall j \in T \setminus \{i\} : \mathsf{prec}[i] \neq \mathsf{lct}_j$$

when the precedence relationship for i has been strengthened based on the new rule. In this case, the correct task j to use in update is given by:

$$\max_{\mathsf{lct}_j}\{j \in T \mid \mathsf{lct}_j \leq \mathsf{prec}[i]\} \tag{3.17}$$

In the case of duplicates, the last such j added to Θ is again the one required; it can be located by iterating through $\mathsf{auxOrder}$ from element 0, and returning the first task that satisfies (3.17).

3.3.2 Definition of minest

Perhaps the most challenging section of [39] is Section 7: Time Bound Adjustment. This is partly due to the fact that the determination of updated bounds is the most complex part of any cumulative edge-finding algorithm; furthermore, in order to reduce this complexity the author manipulates the formulae in ways that do not lend themselves to easy visualization. The author's intent is difficult to divine for an additional reason, however, namely that while the algorithm

given is correct, its derivation is not, due to an incorrect definition of one of the terms.

Recall from Section 2.2.2, that the calculation of potential updates to a task i with a resource demand of c_i is termed $\mathsf{update}(j, c_i)$, which is calculated for all j such that $\mathsf{lct}_j < \mathsf{lct}_i$. Further, for every such j, the correct update is based on the maximum earliest completion time of any subset of the task interval $\mathsf{LCut}(T, j)$. These calculations take advantage of the nested nature of these task intervals; that is:

$$\mathsf{lct}_{j_1} < \mathsf{lct}_{j_2} < \cdots < \mathsf{lct}_{j_n} \implies \mathsf{LCut}(T, j_1) \subset \mathsf{LCut}(T, j_2) \subset \cdots \subset \mathsf{LCut}(T, j_n)$$

By computing $\mathsf{update}(j, c_i)$ in order of increasing lct_j, it is therefore possible to reuse the results of $\mathsf{update}(j_{l-1}, c_i)$ in the computation of $\mathsf{update}(j_l, c_i)$ (for the details of this calculation, the reader is referred to [39, Section 7]). As the subsets of each $\mathsf{LCut}(T, j)$ is searched for new maximum updates, therefore, only those subsets which have not been seen before need be checked, and those subsets are specifically those task intervals which have an upper bound of j. Only task intervals need to be considered [22], so with the upper bound fixed, only the lower bound needs to be varied: the goal of the update calculation is to locate the lower bound which defines the task interval responsible for the strongest update.

In [39, Section 7], this strongest interval is located by dividing the tasks of $\mathsf{LCut}(T, j)$ into two groups: those tasks which must be part of every interval which could cause an update, and the remaining tasks. The division of the tasks is based on whether their earliest start time is greater or less than the quantity $\mathsf{minest}(j, c)$, defined as [39, Section 7]:

$$\mathsf{minest}(j, c) = \min\{\mathsf{est}_\Omega, \Omega \subseteq \mathsf{LCut}(T, j) \wedge \mathsf{e}_\Omega > (\mathcal{C} - c)(\mathsf{lct}_j - \mathsf{est}_\Omega)\} \quad (3.18)$$

When compared with (2.1), it is clear that the inequality in (3.18) is equivalent to $\mathsf{rest}_\Omega > 0$. Put another way, $\mathsf{minest}(j, c)$ is the minimum earliest start time of any subset of tasks of $\mathsf{LCut}(T, j)$ that is energetic enough to interfere with the scheduling of another task that has a resource requirement of c. Consider the example shown in Figure 3.3: the set of tasks $\mathsf{LCut}(T, C) = \{A, B, C\}$ is energetic enough to cause an adjustment to est_D (because $\mathsf{rest}_{\{A,B,C\}} = 5$); furthermore, $\mathsf{est}_A = 0$, which is the least earliest start time of any task in the example, and by (3.18), we have $\mathsf{minest}(C, c_D) = \mathsf{est}_A = 0$.[2] In calculating $\mathsf{update}(j, c)$, $\mathsf{minest}(j, c)$ is used to split the tasks in $\mathsf{LCut}(T, j)$ into two parts, as follows:

$$\alpha(j, c) = \{l, l \in \mathsf{LCut}(T, j) \wedge \mathsf{est}_l \leq \mathsf{minest}(j, c)\}$$
$$\beta(j, c) = \{l, l \in \mathsf{LCut}(T, j) \wedge \mathsf{est}_l > \mathsf{minest}(j, c)\}$$

[2] Note that as $\mathsf{lct}_B = \mathsf{lct}_C = 5$, it is also the case that $\mathsf{minest}(B, c_D) = \mathsf{est}_A = 0$. In practice the sets $\mathsf{LCut}(T, j)$ are generated from a list of tasks sorted by last completion time; in cases such as this, with a duplicate bound, the sort order is arbitrary, with the result that $\mathsf{LCut}(T, j)$ will be a *pseudo-task interval*, with membership determined not by the value of the bound for each task, but rather by the task's position in the sorted array. This is similar to the issue discussed in Section 3.3.1; here, as there, the algorithm will yield the correct answer even given such duplicate bounds. For simplicity, this section will assume a sort order among the example tasks of A, B, C, D; hence $\mathsf{LCut}(T, A) = \{A\}$, $\mathsf{LCut}(T, B) = \{A, B\}$, and $\mathsf{LCut}(T, C) = \{A, B, C\}$.

These two sets are then used to calculate a new version of the energy envelope, given as (compare to [39, (15)]):

$$\mathsf{Env}(j,c) = \mathsf{e}_\beta + \mathsf{Env}(\alpha(j,c)) \tag{3.19}$$

Recall from (2.3) that $\mathsf{Env}(\alpha(j,c))$ is used to calculate the lower bound of the earliest completion time of $\alpha(j,c)$. $\mathsf{Env}(j,c)$ is also used to compute the earliest completion time of a set of tasks (in this case, the set is $\mathsf{LCut}(T,j)$), but with an important difference: by adding e_β, equation (3.19) ensures that all the tasks in β are included in the set of tasks from which the earliest completion time is derived. Since $\mathsf{Env}(\alpha(j,c))$ already includes the energy of the task which defines $\mathsf{minest}(j,c)$, this means that all the tasks with $\mathsf{est}_k \geq \mathsf{minest}(j,c)$ become part of the subset of $\mathsf{LCut}(T,j)$ that is used to calculate $\mathsf{update}(j,c)$, while those with $\mathsf{est}_k < \mathsf{minest}(j,c)$ are included only if they are part of the set of tasks that generates the strongest update.

Returning to our example, we have $\alpha = \{A\}$ and $\beta = \{B,C\}$, and a quick calculation shows:

$$\mathsf{Env}(C,c_D) = \mathsf{Env}_{\{A\}} + \mathsf{e}_{\{B,C\}} = 3 + 7 = 10$$

Using (2.12), we can then compute the update to est_D as:

$$\mathsf{diff}(C,c_D) = \left\lceil \frac{\mathsf{Env}(C,c_D) - (\mathcal{C} - c_D)\,\mathsf{lct}_C}{c_D} \right\rceil = \left\lceil \frac{10 - (3-2)5}{2} \right\rceil = 3$$

This is exactly the update we would expect from a calculation that includes all the tasks in $\{A,B,C\}$, as seen in Figure 2.3. As shown in Figure 2.4, however, a stronger adjustment to est_D is obtained by considering the set $\{B,C\}$ only. Note also that $\{B,C\} \not\subseteq \mathsf{LCut}(T,A)$ and $\{B,C\} \not\subseteq \mathsf{LCut}(T,B)$, so the adjustment based on $\{B,C\}$ will not be detected by this method.

The error in the derivation of $\mathsf{update}(j,c)$ may be seen in the justification of [39, Equation (12)]. This equation is a modification of [39, Equation (11)], repeated here earlier as (2.11); in the original equation, the restriction on the considered sets $\Omega \subseteq \mathsf{LCut}(T,j)$ is $\mathsf{e}_\Omega > (\mathcal{C} - c)(\mathsf{lct}_j - \mathsf{est}_\Omega)$, while in the new equation this condition is replaced with $\mathsf{est}_\Omega \leq \mathsf{minest}(j,c)$. This modification is justified by two assertions: firstly, that the modified condition considers more sets than the original did, and secondly, that none of these new sets will affect the maximum value of the computation. The first of these assertions is, however, incorrect.

Consider again the example above: the set $\{B,C\}$ meets the original condition:

$$\mathsf{e}_{\{B,C\}} = 7 > 3 = (3-2)(5-2) = (\mathcal{C} - c_D)(\mathsf{lct}_C - \mathsf{est}_{\{B,C\}})$$

It fails to meet the new condition, however:

$$\mathsf{est}_{\{B,C\}} = 2 \not\leq 0 = \mathsf{minest}(j,c)$$

Hence, the set $\{B,C\}$ would be considered by the original definition of $\mathsf{diff}(j,c)$, but is ignored by the modified version.

The algorithm provided to compute $\mathsf{minest}(j,c)$ [39, Algorithm 2], when applied to this same example, yields the quite different result $\mathsf{minest}(C,c_D) = \mathsf{est}_C = 2$; when used to cut the tree and calculate $\mathsf{Env}(C,c_D)$, this in turns results

in $\mathsf{update}(C, c_D) = 4$ (verification is left as an exercise for the reader). It is clear that the algorithm computes a $\mathsf{minest}(j, c)$ that differs from the definition. This difference arises from the fact that the algorithm utilizes a variant of the energy envelope to locate the node responsible for $\mathsf{minest}(j, c)$ [39, Section 7]:

$$\mathsf{Env}^c(\Theta) = \max_{\Omega \subseteq \Theta} \{(\mathcal{C} - c)\,\mathsf{est}_\Theta + \mathsf{e}_\Theta\}$$

The key is that Env^c depends on the *maximum* earliest start time of the subsets of Θ, not the minimum. Using Env^c to locate an Ω with $\mathsf{rest}_\Omega > 0$ results in an Ω which has the maximum earliest start time of any such subset. This is the task we actually need to find: the task interval defined by this earliest time is the smallest task interval which needs to be considered when calculating $\mathsf{update}(j, c)$. All of the tasks in this interval are guaranteed to be part of the task interval that generates the strongest update; in the algorithm, these tasks form the set β, which means that their energy will always be included in the computation. The tasks that fall below this interval may or may not be part of the strongest interval; the energy of these tasks is included by way of their energy envelope, which means that only those which are part of the set with the maximum earliest completion time will be included in the calculation.

The replacement of $\mathsf{minest}(j, c)$ with

$$\mathsf{maxest}(j, c) = \max\{\mathsf{est}_\Omega, \Omega \subseteq \mathsf{LCut}(T, j) \wedge \mathsf{e}_\Omega > (\mathcal{C} - c)(\mathsf{lct}_j - \mathsf{est}_\Omega)\}$$

also fixes the problem with the derivation of $\mathsf{update}(j, c)$ discussed above. The condition $\mathsf{est}_\Omega \leq \mathsf{maxest}(j, c)$ does include more sets than the original condition, and these new sets are still not capable of influencing the maximum value, so $\mathsf{diff}(j, c)$ will be correctly computed.

3.3.3 $\alpha - \beta$ Tree Cutting

The division of the Θ-tree into the subtrees α and β, described in the previous section, is accomplished by locating the task responsible for $\mathsf{maxest}(j, c_i)$. Using the provided algorithm [39, Algorithm 2], this task can be located by tracing a root-to-leaf path in $\mathcal{O}(\log n)$ time. The Θ-tree is then "cut": the task responsible for $\mathsf{maxest}(j, c_i)$, designated l, and all leaves to its left become the leaves of the α subtree, while those leaves to the left from the β subtree. The roots of these trees are then used in the calculation of $\mathsf{Env}(j, c_i)$. The procedure for making this cut is not given in [39], although it is asserted that it can be accomplished in $\mathcal{O}(\log n)$ time as well.

Non-cutting Algorithm

It is not necessary to actually cut the tree; instead, $\mathsf{e}(\beta)$ and $\mathsf{Env}(\alpha)$ may computed by tracing a single leaf-to-root path. First the node l is located. l is the rightmost leaf in the α-subtree, and as yet no node in the β-subtree has been encountered, giving the values:

$$\begin{aligned} \mathsf{Env}_l(\alpha) &= \mathsf{Env}_l \\ \mathsf{e}_l(\alpha) &= \mathsf{e}_l \\ \mathsf{e}_l(\beta) &= 0 \end{aligned}$$

l is either the left or right child of a parent-node, p. If $l = left(p)$, then p lies in the β-subtree, as do all nodes to the right of p; if $l = right(p)$, p is in the α-subtree along with all nodes to the left of p. So

$$\mathsf{Env}_p(\alpha) = \begin{cases} \max\{\mathsf{Env}_{left(p)} + \mathsf{e}_l(\alpha), \mathsf{Env}_l(\alpha)\} & \text{if } l = right(p) \\ \mathsf{Env}_l(\alpha) & \text{if } l = left(p) \end{cases}$$

$$\mathsf{e}_p(\alpha) = \begin{cases} \mathsf{e}_{left(p)} + \mathsf{e}_l(\alpha) & \text{if } l = right(p) \\ \mathsf{e}_l(\alpha) & \text{if } l = left(p) \end{cases}$$

$$\mathsf{e}_p(\beta) = \begin{cases} \mathsf{e}_l(\beta) & \text{if } l = right(p) \\ \mathsf{e}_l(\beta) + \mathsf{e}_{right(p)} & \text{if } l = left(p) \end{cases}$$

These calculations are then repeated in the parent node of l; on reaching the root of the Θ-tree, $\mathsf{e}_l(\beta)$ and $\mathsf{Env}_l(\alpha)$ will be equal to the corresponding values in the root nodes of the α and β subtrees (see Algorithm 3.8).

```
1  calcEnvJC(tree, bound, c)
2  begin
                // Find the node containing minest or maxlct
3      v ← tree.root
4      E ← 0
5      maxEnv^c ← (tree.C − c) ∗ bound
6      while v is not a leaf node do
7          if Env^c_right(v) +E > maxEnv^c then
8              v ← right(v)
9          else
10             E ← E + e_right(v)
11             v ← left(v)
                      // 'Cut' of tree (see Section 3.3.3)
12     e(α) ← e_v
13     Env(α) ← Env_v
14     e(β) ← 0
15     while v is not root do
16         if v is a left child then
17             e(β) ← e(β) + e_sibling^r(v)
18         else                                   // v is a right child
19             Env(α) ← max{Env_sibling^l(v) + e(α), Env(α)}
20             e(α) ← e_sibling^l(v) + e(α)
21         v ← parent(v)
22     if Env(α) = −∞ then
23         Env(α) ← 0
24     return e(β) + Env(α)
```

Algorithm 3.8 – calcEnvJC(tree, bound, c) makes a root-to-leaf traversal to locate the leaf node which represents minest$(bound, c)$, then a leaf-to-root traversal to 'cut' the tree at that leaf, into α and β subtrees. The return value is $\mathsf{Env}(j, c) = \mathsf{e}_{\beta(j,c)} + \mathsf{Env}(\alpha(j, c))$.

```
1 COMPUTEENV^C(tree, c)
2 begin
3     for n ∈ LEAVES(tree ) do
4         Env^c_n ← (tree.Capacity − c) ∗ bound_n + e_n
```

Algorithm 3.9 – COMPUTEENV$^{\mathsf{C}}$ calculates $\mathsf{Env}^{\mathsf{c}} = (\mathcal{C} - c)\,\mathsf{bound}_j + \mathsf{e}_j$ for all leaf nodes in the Θ-Λ-tree tree. This computation does not affect the values of $\mathsf{Env}^{\mathsf{c}}$ in the leaves Θ-nodes; those values are computed from the node's base $\mathsf{Env}^{\mathsf{c}}$ value when the node is added to Θ.

Symmetry

To demonstrate that the calculation of $\mathsf{Env}(j, c)$ is symmetric for a $-\mathsf{lct}$ sorted Θ-tree, we start by defining:

$$\alpha(j,c) = \{l, l \in \mathsf{LCut}'(T, j)\ \&\ -\mathsf{lct}_l \leq \mathsf{minlct}(j,c)\}$$
$$\beta(j,c) = \{l, l \in \mathsf{LCut}'(T, j)\ \&\ -\mathsf{lct}_l > \mathsf{minlct}(j,c)\}$$

we can see that (compare to [39, (15)]):

$$\begin{aligned} -\mathsf{Env}'(j,c) &= \max_{\substack{\Omega_1 \subseteq \alpha(j,c) \\ \Omega_2 \subseteq \beta(j,c)}} \{-\mathcal{C}\,\mathsf{lct}_{\Omega_1} + \mathsf{e}_{\Omega_1} + \mathsf{e}_{\Omega_2}\} \\ &= -\mathsf{Env}'(\alpha(j,c)) + \mathsf{e}_{\beta(j,c)} \end{aligned}$$

which may be calculated on an *lct*-sorted Θ-tree using the algorithm presented for *est* in Section 3.3.3. To cut the tree we must first calculate $\mathsf{minlct}(j, c)$; to do this we first note that the condition $\mathsf{e}_\Omega > (\mathcal{C} - c)(\mathsf{lct}_\Omega - \mathsf{est}_j)$ in (3.10) is equivalent to (compare to [39, (16)]):

$$(\mathcal{C} - c)\,\mathsf{lct}_\Omega - \mathsf{e}_\Omega < (\mathcal{C} - c)\,\mathsf{est}_j$$

Letting:

$$\mathsf{Env}^{\mathsf{c}\prime}(\Theta) = \min_{\Omega \subseteq \Theta} \{(\mathcal{C} - c)\,\mathsf{lct}_\Omega - \mathsf{e}_\Omega\} \tag{3.20}$$

we see that

$$\begin{aligned} \mathsf{Env}^{\mathsf{c}\prime}(\beta(j,c)) &> (\mathcal{C} - c)\,\mathsf{est}_j \\ -\mathsf{Env}^{\mathsf{c}\prime}(\beta(j,c)) &< (\mathcal{C} - c)(-\mathsf{est}_j) \end{aligned} \tag{3.21}$$

(3.21) enables us to use the algorithm for the computation of $\mathsf{maxest}(j, c)$ on an *est*-sorted Θ-tree [39, Algorithm 2] to compute $\mathsf{minlct}(j, c)$ on an *lct*-sorted Θ-tree.

Θ-tree Calculation of $\mathsf{Env}^{\mathsf{c}}$

$\mathsf{Env}^{\mathsf{c}\prime}(\Theta)$ (3.20) and its counterpart from the earliest start time adjustment phase, $\mathsf{Env}^{\mathsf{c}}(\Theta)$ [39, unlabelled equation following (16)], are similar in structure to $\mathsf{Env}'(\Theta)$ and $\mathsf{Env}(\theta)$, and may be similarly computed using a Θ-tree. Generalizing the calculation to work for either bound, the calculation for a Θ-tree node is:

$$\mathsf{Env}^{\mathsf{c}}{}_i = \begin{cases} (\mathcal{C} - c)\,\mathsf{bound}_i + \mathsf{e}_i & \text{if } i \text{ is a leaf} \\ \max\{\mathsf{Env}^{\mathsf{c}}{}_{\mathsf{left}(i)} + \mathsf{e}_{\mathsf{right}(i)}, \mathsf{Env}^{\mathsf{c}}{}_{\mathsf{right}(i)}\} & \text{otherwise} \end{cases} \tag{3.22}$$

$\mathsf{Env}^{\mathsf{c}}(\Theta)$ is subtly different from $\mathsf{Env}(\Theta)$, and from any of the other value we have so far computed using the Θ-tree, because it depends on the value of c (not c_i). Previous Θ-tree calculations have relied only on values intrinsic to the tree itself (i.e. the contents of the nodes, and which nodes are in Θ or Λ), or which are constant ($\mathcal{C}$, the resource capacity for the problem). During execution, the value of c in (3.22) ranges over the values c in all the tasks, in the loop beginning on line 3 of Algorithm 3.6. Outside of this loop, calculation of $\mathsf{Env}^{\mathsf{c}}(\Theta)$ is meaningless, so it is not possible to compute $\mathsf{Env}^{\mathsf{c}}{}_i$ for the leaf nodes upon the creation of the Θ-tree (the way that other leaf values are handled). Inside the loop, nodes are moved in and out of Θ, with each move requiring an update of $\mathsf{Env}^{\mathsf{c}}(\Theta)$, and it is undesirable to recalculate all values of $\mathsf{Env}^{\mathsf{c}}$ repeatedly.

A new operation on the Θ-tree is required, therefore, which can set $\mathsf{Env}^{\mathsf{c}}{}_i$ values for all the leaves based on the current value of c. As this value must be able to survive the node's subsequent movement in and out of Θ, it is not stored inside the Θ-node; instead it is stored as $\mathsf{base\text{-}Env}^{\mathsf{c}}$ in the outer Θ-Λ-node (see Figure 3.2). When the leaf is later added to Θ, this value will be copied to the $\mathsf{Env}^{\mathsf{c}}$ element of the Θ-node by ADDTOTHETA (Algorithm 3.3, line 21). Note that, at this time, COMPUTEENV$^{\mathrm{C}}$ does not trigger a recomputation of the $\mathsf{Env}^{\mathsf{c}}{}_i$ values of the internal nodes, leaving the Θ-tree in an inconsistent state. This is due to an efficiency concern: $\mathsf{Env}^{\mathsf{c}}$ is used only during the computation of update (Algorithm 3.6, line 7), and that calculation begins by moving all nodes out of Θ and then returning them one at a time. Any initial computation of interior node values would be immediately wiped out by the CLEARTREE operation.

3.3.4 Update Calculation

In [39, Algorithm 3], $\mathsf{update}(j, c)$ is computed for all values of j and c before iteration over the tasks for adjustment. This computation forms the bottleneck for all edge-finding algorithms [22]: the computation of $\mathsf{update}(j, c)$ for one pair of values has a complexity of $\mathcal{O}(n)$, leading to a $\mathcal{O}\left(kn^2\right)$ complexity to compute all the values.

In [39, Algorithm 3], the values of $\mathsf{update}(j, c)$ are computed with a complexity of $\mathcal{O}(kn \log n)$. For a given value of c, a Θ-tree is used to iteratively calculate $\forall j \in T : \mathsf{update}(j, c)$. The tasks in T are sorted in increasing order by $\mathsf{auxBound}$, giving a sequence where:

$$\mathsf{auxBound}_{j_1} \leq \mathsf{auxBound}_{j_2} \leq \cdots \leq \mathsf{auxBound}_{j_n}$$

The calculation of $\mathsf{update}(j_i, c)$ can then use the result of the $\mathsf{update}(j_{i-1}, c)$, so that the calculation for each j can be performed with a complexity of $\mathcal{O}(\log n)$, yielding an overall complexity of $\mathcal{O}(n \log n)$ 1 for each value of c.

As suggested in [22], it is possible in practice to slightly reduce the complexity by dropping the precomputation of $\mathsf{update}(j, c)$ for those values of c which are never used in an adjustment, yielding a complexity of $\mathcal{O}(\Delta n \log n)$, where Δ is the number of distinct task capacities that occur in adjustment operations actually computed. The worst case complexity, which occurs when $\Delta = k$, would not be improved by this modification, but there should be some improvement to the average complexity.

This reduced complexity can be accomplished by paying careful attention to

loop order in the adjustment phase [3]. This order has already been touched on in section 3.3.1, where it was shown that the adjustment of bound_i should be based on:

$$\mathsf{update}(j, c_i) : \max_{\mathsf{auxBound}_j} \{j \in T | \, \mathsf{auxBound}_j \leq \mathsf{prec}[i]\} \tag{3.23}$$

Further, due to the order of the tasks in auxOrder, in an iteration through that array starting from the first element, the value of j which satisfies (3.23) will be the first one encountered which satisfies $\mathsf{auxBound}_j \leq \mathsf{prec}[i]$.

The set of required comparisons can be reduced further by considering the dominance properties of edge-finding, as discussed in [22]. Recall that

$$\mathsf{prec}[i] = \mathsf{auxBound}_j \iff \mathsf{LCut}(T, j) \lessdot i$$

and further:

$$\mathsf{LCut}(T, j) = \{l \in T | \, \mathsf{auxBound}_l \leq \mathsf{auxBound}_j\}$$

It is shown in [22, Property 3] that to calculate the adjustment of a task i, an edge-finder need only consider those sets of tasks $\mathsf{LCut}(T, j)$ where $\mathsf{auxBound}_j < \mathsf{auxBound}_i$. Algorithm 3.6 takes advantage of this fact by iterating through the tasks as indexed in the auxOrder array (in other words, by selecting i in order of auxBound); it can then restrict the search for $\mathsf{update}(j, c_i)$ to the tasks $j \in \mathsf{auxOrder}$ where $j > i$.

It is within this reduced loop that the comparison

$$\mathsf{auxBound}_j \leq \mathsf{prec}[i]$$

is performed. It is only when an appropriate i, j pair is located here that there exists the possibility of an update to the bound of i. In that case, it is determined whether $\mathsf{update}(j, c_i)$ has already been calculated; if it has not,

$$\forall l \in T : \mathsf{update}(l, c_i)$$

is calculated, using the algorithm given as the innermost loop of [39, Algorithm 3].

3.3.5 Overload Checking

The edge-finding algorithm in [39] is a filtering algorithm that could be used in the implementation of a propagator for the cumulative constraint. As discussed in Section 1.2.3, edge-finding is only a *partial* filter for cumulative: an edge-finding filter will always accept more solutions than would satisfy the cumulative constraint. Some of these extraneous solutions can be eliminated if the propagator implements additional partial filtering algorithms, the principle being that each partial filter will help to eliminate some of the solutions that were accepted by the others.

These filters do not need to be combined into a single algorithm; assuming the partial filters to be monotonic, they can be run sequentially in any order, and the result will be the same filtering of the constraint store (although the order may affect execution time). [38] demonstrates this arrangement for the

[3] It should be noted that this is not a topic addressed in [39], which covers in detail the computation of $\mathsf{update}(j, c)$, and then notes that it is then trivial to use the prec and the equation ([39, EF2]) to compute the actual adjustment.

implementation of a global disjunctive scheduling propagator, using a sequence of overload checking, detectable precedences, not-first/not-last, edge-finding, and precedence graph filters to strengthen the store more than any of the individual filters could.

Section 1.2.3 described two other partial filters that are usually combined with edge-finding in implementations of the cumulative constraint: time-tabling and overload checking. Of these two, time-tabling is best implemented separately from edge-finding, while overload checking may be easily performed within the edge-finding algorithm.

Recall from Definition 1.6 that a set of tasks T is said to be e-feasible if no subsets of tasks is overloaded. Using a naive approach, $2^n - 1$ task subsets would have to be checked to verify e-feasibility; however, as shown in [43], it is possible to verify e-feasibility by checking a set of n task intervals. In [40], the task intervals used to verify e-feasibility are $\mathsf{LCut}(T, j)$ for all $j \in T$. So the overload rule can be rewritten as:

$$\forall j \in T : (\mathsf{Env}(\mathsf{LCut}(T, j)) > \mathcal{C}\,\mathsf{lct}_j) \implies \textbf{overload})$$

As we already iterate through all $\mathsf{LCut}(T, j)$ during edge-finding, it seems that overload checking could be usefully performed as part of the edge-finding filter, instead of being implemented as a separate filtering algorithm executed in sequence with edge-finding. There is no need for an additional loop; the overload check is therefore carried out on line 8 of Algorithm 3.5. At this point the bounds have not yet been adjusted, so the only overloads that may be detected are those which existed before any strengthening of the bounds by the edge-finder. As the algorithm is not idempotent, and must be repeated until a fixpoint is reached, any adjustment to the bounds on this iteration will guarantee an additional iteration, during which the new bounds will be checked for e-feasibility.

3.3.6 Optional Activities

Vilím proposes a simple method for modeling optional activities in this algorithm: by temporarily setting the optional activity's bound to be infinite, the optional activity is prevented from influencing any of the required activities. For example, when adjusting the earliest start time of an optional task j, the computation pretends that $\mathsf{lct}_j = \infty$, which means that j can always be scheduled later than any subset of tasks with which it would conflict. In terms of the algorithm, the test at line 9 of Algorithm 3.5 will never be true, so there is no task i for which est_i will be adjusted based on j.

Note, though, that this method requires special checking to determine when an optional activity can be eliminated. Normally the viability of a task is determined through overload checking, but any potential overload involving j will be avoided by simply scheduling j later. Instead, we must check that the newly adjusted *est* does not make it impossible to complete j before the actual lct_j; in other words:

$$(\mathsf{est}'_j + p_j > \mathsf{lct}_j) \implies \mathsf{status}_j \leftarrow \mathit{disabled}$$

This check can be performed when the bound is updated, which occurs in the syncLB and syncUB routines.

A version of SYNCLB suitable for optional activities is presented as Algorithm 3.10. For optional activities, this algorithm only tries to determine if the activity may be disabled; it does not adjust the bounds of optional activities based on scheduling conflicts with non-optional activities, as suggested in [39, Section 9]. Inclusion of this last type of adjustment unfortunately leads to unsound propagation [31].

```
1 if status_t = optional and newEst + p_t > lct_t then
2     status_t ← disabled
3 if status_t = enabled and est_t < newEst then
4     est_t ← newEst
```

Algorithm 3.10 – SYNCLB including optional activity check

3.4 Experimental Results

3.4.1 Correctness

In [22], Mercier and Van Hentenryck present an $\mathcal{O}\left(kn^2\right)$ edge-finder which they demonstrate to be complete, i.e. one that detects all bound updates that would be detected by an edge-finder that checked all subsets of tasks. Vilím shows that the edge-finder in [39] will perform updates *at least* as strong as those performed by the Mercier and Van Hentenryck algorithm. In order to verify this experimentally, the Θ-tree edge-finder implementation described here was run alongside an implementation of the Mercier ad van Hentenryck algorithm.

Tests were run on randomly generated task data, for problems of sizes between $n = 4$ and $n = 100$. In each test, the same test data was run in each edge-finder; the results were compared to determine whether one implementation had performed a stronger update. Over the course of 100,000 tests, the two algorithms performed exactly the same updates in approximately 99.5% of all cases. In all observed cases in which the algorithms made different adjustments, the Θ-tree edge-finder was the source of the stronger update. Manual calculations performed on these differing results failed to find an instance where the stronger update generated by the Θ-tree edge-finder was invalid.

A preliminary analysis of the test cases with different results seems to indicate that the stronger updates generated by the Θ-tree algorithm are due to the detection improvement proposed in [39, Section 6.2], and discussed further in 3.3.1. This improvement detects some precedences that would not be detected by a standard edge-finder, resulting in stronger updates. While the idea behind this improvement seems sound, and no experimental case was found in which it resulted in an incorrect update, it is nonetheless important to bear in mind that no proof of the correctness of this improvement is included in [39]. An implementation that includes this modification is, strictly speaking, no longer an edge-finder, but rather a slightly tighter relaxation of the cumulative constraint.

3.4.2 Complexity and Execution Time

For edge-finding algorithms, problem size is dependent not just on the number of tasks (n), but also on the number of distinct resource requirements among

the tasks (k). Because the problem is discrete, $k \leq \min\{\mathcal{C}, n\}$.

Tests were run on problems randomly generated as follows. First, the resource capacity was fixed to $\mathcal{C} \in \{2, 5, 10, 20\}$. For each $\mathcal{C}$, tests were run for all $1 \leq k \leq \mathcal{C}$. With each value of k, twenty tests were run for each value $n \in [10, 15, 20, \ldots, 100]$. The value of k was ensured by first randomly selecting k unique values in the range $[1..\mathcal{C}]$, and then assigning these values to the tasks by random draw (taking care to verify that each value is used at least once). Once all tests for a given n and k were complete, the two highest and lowest times were discarded, and the remaining times averaged. The results are given in Appendix A.

The results include times for both the Θ-tree edge-finding algorithm and "standard" edge-finding. For the latter, the algorithm from [22] was used. To make the timing comparison more informative, the standard implementation was developed to utilize the same data structures (excepting, of course, the Θ-tree itself) and memory allocation as the Θ-tree implementation described here. The difference in the execution times can be clearly seen in Figure 3.4. Here, mean execution times for test of size n are plotted alongside best fit curves of n^2 and $n \log n$ for the standard and Θ-tree implementations, respectively (the value of k is constant in each plot). It is clear that the lower complexity of the Θ-tree algorithm quickly pays off as problem size increases, resulting in substantially lower execution times. However, it is also worth noting that for problems of $n < 35$ the standard edge-finder outperformed the Θ-tree implementation. In [39], the Θ-tree edge-finder was tested alongside the $\mathcal{O}\left(n^2\right)$ algorithm from [7], which is demonstrated to be incomplete in [22]; nevertheless, even against this lower complexity algorithm, Vilím reports a speedup factor of 1.34 for $n = 20$, and by a factor of over 4 by $n = 100$. These results are clearly stronger than the speed increases shown here; most likely the implementation discussed in this thesis requires further optimization to equal the original author's results.

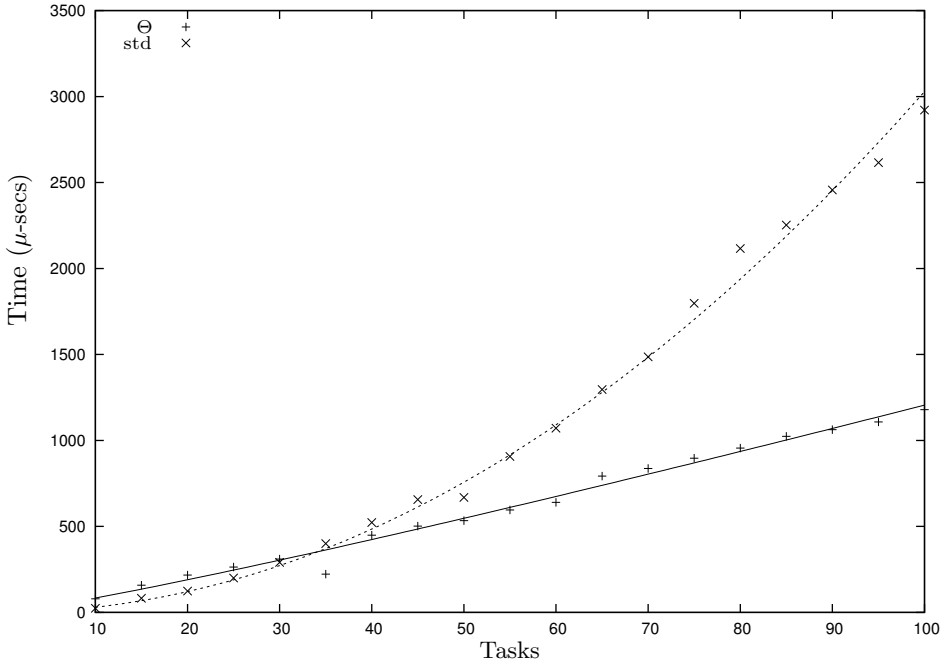
Figures 3.5 and 3.6 show the link between k and the complexity of the Θ-tree algorithm. In Figure 3.6 the value of n is fixed while k varies, and the resulting distribution of times is essentially linear.

3.5 Summary

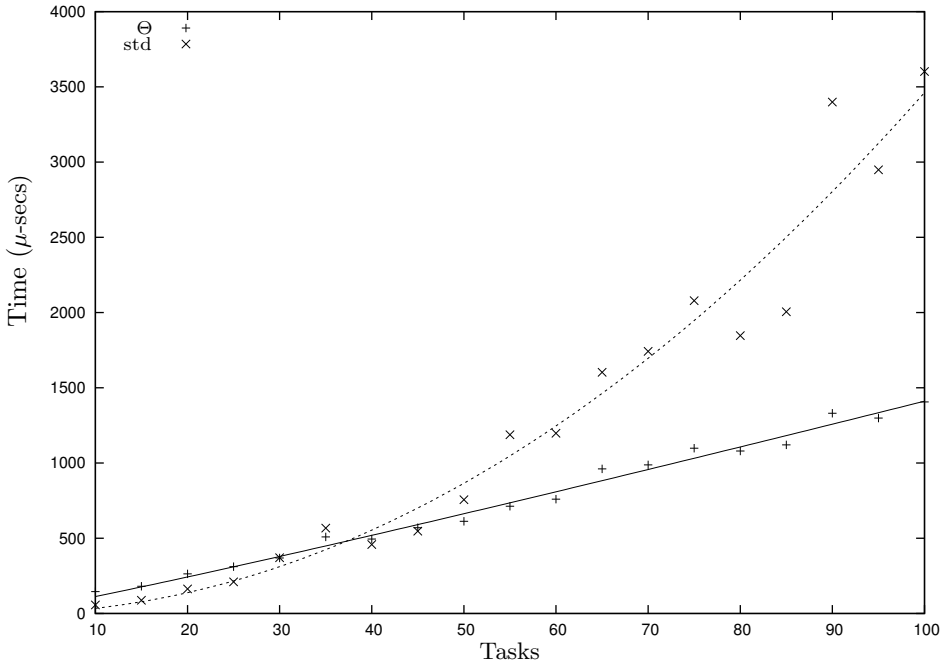
In the course of outlining the implementation of the Θ-tree edge-finding algorithm given in this chapter, several clarification to the description given in [39] have been made, and some errors in that paper have been described. For ease of reference, these clarifications and errors are summarized here:

Section 30 It is noted in [39] that any balanced binary tree may be used to implement the Θ-tree; however, the static structure of the Θ-tree suggests that in practice the nodes may be efficiently stored in an array. Furthermore, if the nodes are populated in the right order, the construction of the Θ-tree becomes a $\mathcal{O}(n)$ operation.

Section 3.1.2 An algorithm is specified here to record the leaf responsible for the lambda values of an interior node, and pass these values up the tree to the root, making it possible to eliminate the $\mathcal{O}(\log n)$ check procedure used in [39, Algorithm ?] to locate the responsible leaf node after the fact.



(a) $\mathcal{C} = 2$, $k = 2$



(b) $\mathcal{C} = 5$, $k = 3$

Figure 3.4 – Comparison of execution times for edge-finding using a Θ-tree implementation versus the "standard" algorithm from [22].

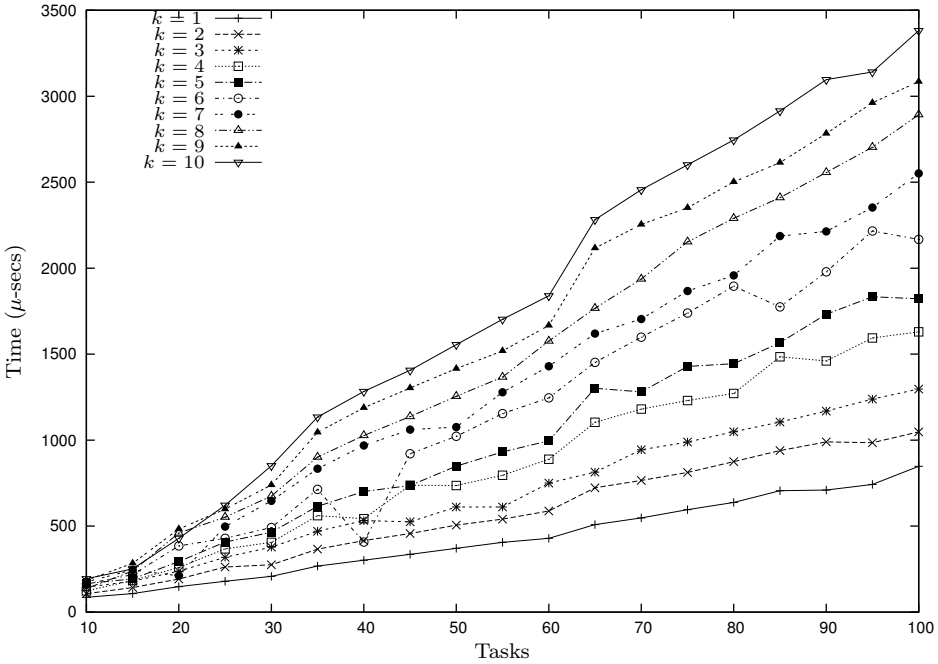


Figure 3.5 – Effect of increasing values of k on execution time for tests where $\mathcal{C} = 20$.

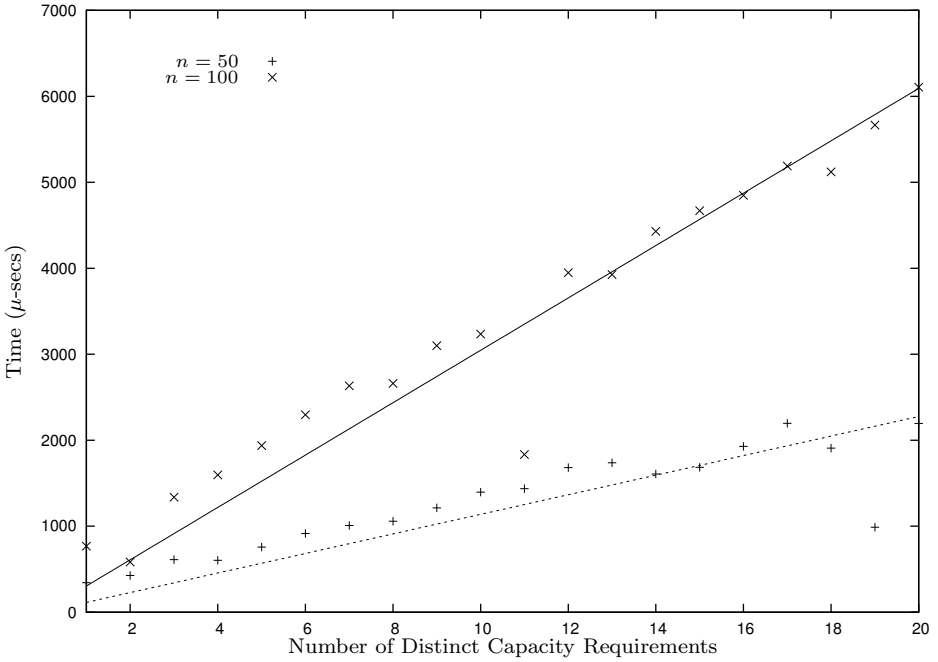


Figure 3.6 – Execution times for Θ-tree edge-finding for tests of $\mathcal{C} = 20$, in terms of k: the number of distinct capacity requirements among the tasks in each test. The number of tasks is fixed to $n = 50$ for the lower set, and $n = 100$ for the upper.

Section 3.2.2 A sign error in [39, algorithm 3] is corrected.

Section 3.3.1 The improvement to order detection outlined in [39, section 6.2] leads to an incorrect strengthening of bounds for fully constrained tasks. To eliminate this problem, it is necessary to ensure that the algorithm does not set $\mathsf{prec}[i] = \mathsf{lct}_i$.

Section 3.3.1 As noted in [39], the algorithm given for the adjustment of bounds remains valid if some of the bounds are not unique; however, this use of pseudo-bound intervals makes determination of loop order in the adjustment phase non-trivial. A loop order that leads to proper adjustments with the minimum number of comparisons is discussed here and in section 3.3.4.

Section 3.3.2 The purpose of $\mathsf{minest}(j, c)$ is elucidated, and the incorrectness of its original definition in [39] and the error in the derivation of $\mathsf{update}(j, c)$ that follows from that definition are explained, and a corrected definition is given.

Section 3.3.3 In [39] ,$\mathsf{Env}(j, c)$ is calculated by cutting the Θ-tree into two smaller trees and performing a calculation using values from the root node of each tree. The author states that this process may be performed in $\mathcal{O}(\log n)$ time, but does not provide an algorithm for doing so. Here an explicit algorithm is given for calculating $\mathsf{Env}(j, c)$ by traversing a single root-to-leaf path

Section 3.3.4 In [39] all values of update are precomputed. Using the improved loop order suggested in [22], it is possible to eliminate some of these relatively expensive operations..

Section 3.3.5 Overload checking, explained in [40] but omitted in [39], is integrated into the edge-finding algorithm, with no addition of complexity.

Section 3.3.6 The adjustment of the bounds of optional activities discussed in [39], which leads to unsound propagation, is eliminated. Instead, a simple check is added to determine if previously optional activities can now be disabled.

Chapter 4

Parallel Edge-Finding

4.1 Parallelism in the Algorithm

The first step in designing a parallel algorithm is the identification of tasks that may be performed concurrently [20]. The Θ-tree edge-finding algorithm has several symmetric sections, some of which may be executed concurrently.

4.1.1 Bounds

The first to be considered is bound symmetry: the adjustment of the earliest start time and last completion time of the tasks is perfectly symmetric in each iteration of the filter. There is almost no task dependency between these two sections; the sole exception is the sorted bounds list. Adjustment of each bound uses a Θ-tree with leaves sorted on that same bound; these two trees can be constructed in local memory for the two different tasks (the serial algorithm has to use these same two Θ-trees to adjust the tasks, so this does not introduce any new overhead in the parallel implementation). However, after the tree is constructed, the tasks are removed from the tree in the order of the other bound, requiring the existence of a task list sorted by this other bound. To reduce the size of the serial section of the algorithm, each bound adjustment section can be responsible for sorting one of the two bound lists: the earliest start time adjustment section will sort the estList, and the last completion time adjustment section will sort the lctList. After this list is sorted, the corresponding tree can be built; before proceeding with the detection phase, though, the parallel section will have to wait until the other sorted bound list is completed.

Note that, even in the serial version, the sorted bound lists are not adjusted as updates are made (i.e., the estList used in the last completion time adjustment section is based on the earliest start time of the tasks *before* the execution of the earliest start time adjustment section). As discussed in Section 3.1.1, the algorithm is not idempotent, so it will have to execute again in any case to discover whether a fixpoint has been reached; updates to the bounds from the previous iteration may result in additional pruning during this new iteration. For the serial algorithm, this ensures that the last completion time adjustment phase can proceed without waiting for the adjusted bounds from the earliest start time adjustment phase, and *vice versa*.

4.1.2 Update Precomputations

Within the adjustment of a bound, the process can be divided into two segments with limited data dependency. There is an expensive precomputation of $\mathsf{update}(j, c)$ for every task j and every distinct capacity requirement c. This precomputation may be carried on while the order detection phase of the filter is proceeding, but this latter task will then have to wait for the precomputation to complete before moving on to the bounds adjustment phase.

The update precomputation, is perfectly parallel over the capacity requirements c, making it relatively easy to parallelize. This is one of the most expensive parts of the algorithm—profiling of the serial algorithm suggests that approximately $1/3$ of all execution time is spent on the calculation of the updates—so it seems like an area that could offer rewarding speedup in a parallel implementation.

There are several factors that will serve to limit the amount of speedup that can be gained by parallelizing the update precomputation, however. For example, computing the update values for one value of c requires the use of a Θ-tree which is modified over the course of the computation—values of Env^c are computed in all nodes, which are dependent on c; further, the tree is emptied of all nodes at the start of the algorithm, and nodes are then added back in one at a time. So the Θ-tree will have to be private to each thread processing an update task.

Parallelizing the update computations also eliminates an optimization used in the serial algorithm. There are kn possible values of update, one for every task and every value of c, but not all of these values are used in the algorithm. The computation is optimized within a particular c value, using the value of $\mathsf{Env}(j_1, c)$ to compute the values of $\mathsf{Env}(j_2, c)$, so if an update is required for one combination of c and j, it is computed for all values of j at the same time. However, not all values of c are used in an update calculation, and in the serial algorithm this fact is used to eliminate unneeded computations.

Unfortunately, this determination cannot be made until after the order detection phase is completed, so it means that updates cannot be precomputed, instead they must be computed on demand. This computation takes place so late in the algorithm that it would be difficult to benefit from parallelization at that point. Instead, the parallel implementation will compute update for all values of c, while another thread performs the detection phase. After the detection phase is complete, the main thread will have to wait for the update computations to finish before it can proceed to the adjustment phase.

Alternately, it is possible to use a manual synchronization to allow some parts of the adjustment phase to begin while the update precomputations continue. Each adjustment depends on a value of c; once update is computed for that value of c, that adjustment can proceed. This data dependency is illustrated in figure 4.1.

4.1.3 Loops

In most instances, the most straightforward method for converting a serial algorithm to a parallel one is to identify loop structures for which the individual iterations may be spread among different processors. This distribution is trivial when there is no data dependency between loop iterations, in other words

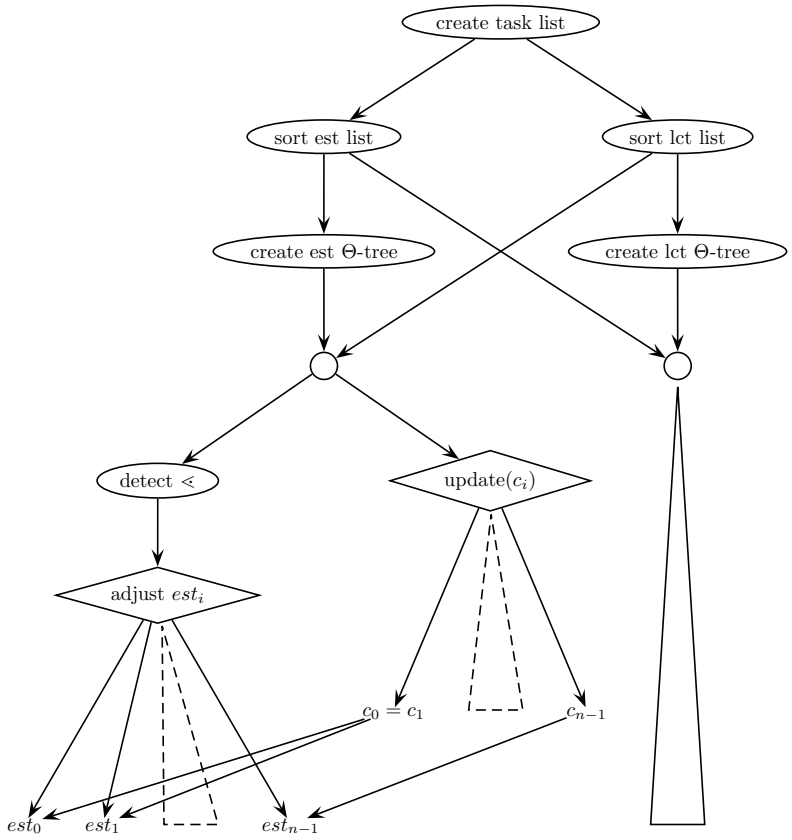


Figure 4.1 – Task-dependency tree for Θ-tree edge-finding. The algorithms for earliest start time and last completion time adjustment are symmetric, and once the sorted task lists are created, independent; past this initial interdependence, only the earliest start time portion of the algorithm is shown. Also, note that while there is an earliest start time adjustment process for every i, the number of update processes may be smaller, as in this case where $c_0 = c_1$, and consequently the adjustment of est_0 and est_1 are each dependent on the completion of the same update calculation.

when the calculations in each loop do not depend on the results of calculations from earlier loops. Even loops with some intra-loop data dependency may be parallelized in many cases, given sufficient care.

Both the detection and adjustment phases of the Θ-tree edge-finding algorithm are comprised mostly of loop structures, but it is hard to see how these might be parallelized effectively. Each iteration involves removing one task from Θ, and a particular strength of the algorithm is the speed at which it can perform these single task deletions. A perfectly parallel loop would require not only copying the Θ-tree structure into each thread, but also making an $\mathcal{O}(n)$ adjustment to the tasks in Θ before starting the computations, resulting in an overall time complexity of $\mathcal{O}\left(kn^2\right)$—a complexity which can be achieved using a

much simpler serial algorithm. Unless a smarter way of parallelizing these loops can be discovered, it seems unlikely that this approach will yield worthwhile results.

In theory, it would be possible to overcome this data dependency using a two-stage parallel pipeline. For example, in the detection phase (presented in Algorithm 3.5), the main loop begins on line 5. In the current algorithm, the Θ-tree is modified at the end of this loop (line 13) when the current task j is moved to the set Λ, but with a slightly restructured loop this modification could be performed at the beginning of the loop. Once j had been moved to Λ, the processor responsible for the j iteration could signal to the $j+1$ processor that the Θ-tree structure had been modified; that processor would then start the $j+1$ iteration. Each iteration would then perform its own calculations using the Θ-tree it formed at the start of the loop; in other words, every iteration would be required to have a local copy of the Θ-tree, which it modified and then passed on the subsequent iteration. This sort of loop interaction might make sense in a message-passing implementation, but is (at best) awkward in a shared-data-space implementation.

Furthermore, there is another alteration to the Θ-tree structure in each iteration: on line 12, the task i is removed from Λ, indicating that the strongest precedence $\mathsf{LCut}(T, j) \lessdot i$ has been detected. This introduces a more difficult dependency among the loops: the simple solution, in which each iteration waits until the previous iteration has performed all such removals it is required to, effectively eliminates the pipeline, as each iteration now has to wait until the entire previous iteration is complete.

To preserve the pipeline, the purpose of this portion of the algorithm must be considered more closely. Recall that Λ is the set of all tasks i for which no precedence has yet been detected. Once a single precedence is detected for i, it is removed from Λ. This removal is justified by the order of the detection loop, which processes tasks in decreasing order of lct_j, thereby assuring that the first precedence detected is the strongest. The algorithm would still be correct if tasks were not removed from Λ as long as weaker precedences were not allowed to overwrite stronger ones; however, many unnecessary comparisons would then be made.

Consider a parallel implementation (Algorithm 4.1) where iteration $j+1$ proceeds without waiting for iteration j to complete. Upon detection of $\mathsf{LCut}(T, j+1) \lessdot i$, the current process would check a global prec array to see if a stronger precedence had already been detected (i.e. if $\mathsf{prec}[i] > j+1$); if it had not, the current precedence would be recorded and i would be removed from a global variable that tracked possible candidates for the set Λ. To eliminate as many extra calculations as possible, each iteration would begin by checking this global Λ pool, and removing any appropriate tasks from the Λ it had just inherited from the previous iteration.

In practice, the significant amount of overhead introduced by this solution makes it extremely unlikely that there would be any parallel speedup. Without extremely large problem sizes, in fact, there would almost certainly be a substantial performance degradation. The adjustment loop is, if anything, less promising; therefore it seems that this sort of loop parallelization is unlikely to yield worthwhile speedups, and it has not been investigated further here.

```
1  pipelinedDetectOrder(j, tree, order, auxOrder, prec, n, C)
2  begin
3      t ← auxOrder[j]
4      receive(j - 1, tree)            // block until received
5      moveToLambda(tree, rank_t + n − 1)
6      for k = 1 to n-1 do
7          if !LambdaPool[k] then
8              removeFromLambda(tree, k)
9      send(j+1, tree)                 // non-blocking
10     if Env_root > C * auxBound_t then
11         FAIL                        // resource is overloaded
12     while Env^Λ_root > C * auxBound_t do
13         i ← resp(Env^Λ)_root
14         if prec[i − (n − 1)] < auxBound_t then
15             prec[i − (n − 1)] ← auxBound_t
16             removeFromLambda(tree, i)
17             removeFromLambdaPool(i)
```

Algorithm 4.1 – pipelinedDetectOrder handles a single iteration of the detection loop, for task j. It must block execution until it receives a Θ-tree from iteration $j-1$; after then modifying that tree it sends it on to iteration $j+1$. To eliminate extra checks for tasks $i \in \Lambda$ for which a precendence has already been detected, a global pool of Λ candidates is maintained. On detecting a precedence, the current task i is removed from that pool.

4.2 Choice of Parallel Platform

Parallel programming platforms may be broadly divided into two groups, categorized by their communication model: message-passing platforms, and shared-data-space platforms [20].

In the message-passing paradigm, each processing node has an address space local to the node that cannot be accessed by the other nodes. Nodes interact through the passing of messages, both to synchronize their activities and to distribute and synchronize data. As data transfer rates are much slower than processor speeds, the distribution of data tends to form the main overhead in implementations using this paradigm; on the other hand, message-passing works quite well for clusters, where the processing nodes may be distributed over several machines. Programming using message-passing is relatively low-level; the programmer must explicitly partition or copy the data, and the division of the parallel tasks must also be specified. As a result, serial code generally requires significant transformation to operate in a message-passing parallel environment. Examples of message-passing platforms are the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM).

In the shared-data-space paradigm, there is a global address space shared by all processors. Additionally, each process maintains a local address space for non-shared data. The global space is logically uniform, although it may be physically distributed among the processors; nonetheless, shared-data-space platforms require some form of hardware support for the logical uniformity of the global space. Communication takes place primarily through modification of

the shared data. When several processes are able to write to the same shared memory address concurrently, a mutual exclusion scheme is required; similar methods may be provided to enable synchronization of tasks. Shared-data-space platforms can provide relatively high-level parallelization, and it is often possible to create parallel implementations that differ very little from the serial implementations. Popular shared-data-space platforms include POSIX threads and OpenMP.

The Θ-tree edge-finding algorithm in [39] has a relatively even mixture of global and local data (see Section 4.3 for details), and could be implemented using either of the above paradigms. The shared-data-space paradigm was chosen here, primarily due to the relatively small problem size. Any parallel implementation introduces overhead not present in serial implementations, such as initialization of the various processes, inter-process communication times, and processor idle times caused by inter-process synchronization. This overhead increases with the number of parallel processes; therefore it is important from a performance standpoint to use enough processes to effectively parallelize the computation, but no more. Given that reasonable problem sizes in cumulative scheduling tend to be of the order of $n = 100$ and lower, there is little profit in designing a parallel implementation which uses a large number of processes. Shared-data-space platforms can be easily configured on multiprocessor machines, which are increasingly available, and which provide sufficient parallel computing capability to effectively parallelize this algorithm.

The platform chosen here is OpenMP, a shared-data parallel platform with an emphasis on the development of code that may be executed in either parallel or serial mode [24]. Parallel processes in OpenMP are explicitly forked off an initial process, using processor directives. These directives may be simply ignored when executing on a machine that does not have access to OpenMP, resulting in serial execution. In addition to the compiler directives, OpenMP defines several procedures, but these can be replaced with a simple library of stubs or removed during compilation. As a result, it is relatively easy to write programs in OpenMP that can take advantage of multiple local processors when possible, but continue to function as serial programs the rest of the time.

4.3 Data Structures

One major area of modification required in the parallel algorithm was the data arrangement. The data structures used in the serial version are shown in figure 3.2. In this arrangement, there is a significant intermixing of data which will be shared across the threads, and that which must be private to either a thread or a team.

For example, the task list is mostly constant through the algorithm. It is constructed once at the beginning, and then used as-is by all parts of the algorithm. Most of the data in the tasks remains static—it is accessed by the algorithm, but not modified. The bound and auxBound elements are used to store earliest start time and last completion time in the two parts of the algorithm, and these are switched at the midway point, but this can be worked around by simply storing earliest start time and last completion time in the tasks and accessing them directly. However, the order and auxOrder elements are more difficult. These are used to record the indices of that task in the two

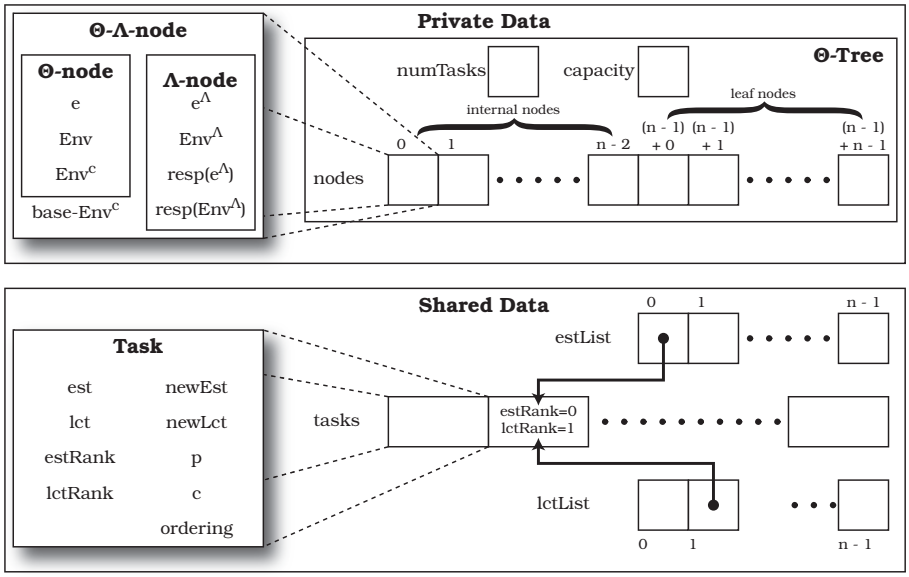


Figure 4.2 – Data structures used in the parallel implementation. See Figure 3.2 for the serial implementation.

sorted bound lists (estList and lctList), to make it possible to move from a task in the auxOrder list to the same task in the order list. These values are significantly different depending on which bound is being adjusted.

To make the parallel algorithm efficient, it was necessary to redistribute the data based on whether is would be shared by the threads or not. The order and auxOrder lists were eliminated; in the parallel version, estList and lctList are passed directly to the detection and adjustment routines, with only a change in the order of the parameters indicating which is currently the sort order of the Θ-tree. The corresponding values in the tasks were changed to *estRank* and *lctRank*; this also requires that the detection and adjustment functions be able to know which of these values to use, so an *ordering* variable was introduced which indicates which bound was used to create the current sort order.

The Θ-tree was mostly independent already, so it required only minor modification to the way that leaf values were computed to allow for the altered task structure. The node array in the tree is dynamically allocated, however, so it was necessary to create a function to manually copy the allocated memory so that the tree could be copied for use in the various threads.

The creation of local data that is a copy of formerly global data is normally accomplished in OpenMP through the use of a data-sharing clause, such as **firstprivate**. When applied to a parallel construct, this attribute not only makes one or more variable local to the task, but also initializes the local copy with the data that is currently held in the global copy [24]. Unfortunately, since the problem size varies, all the arrays in the Θ-tree edge-finding algorithm are dynamically allocated, which limits the usefulness of this clause directive—only the pointer variable to the array is copied, not the dynamically allocated contents. Therefore, array contents had to be copied manually in those cases when it was impossible to defer allocation until after the start of the parallel

section.

4.4 Implementation

Three parallel implementations were tested. All were based on a serial implementation of the algorithm, and reused most of the modules from that implementation, once the modifications to the data structures outlined in the previous section were completed. These modifications were tested first in the serial implementation, verifying that it produced the same results as the original version. This modified version was then used as the base case for computing the speedup of the parallel versions.

4.4.1 Bound Parallel Version

In the first version, outlined in Algorithm 4.2, the computation is divided into two parts, one for adjusting earliest start time and the other last completion time. This division is accomplished using the OpenMP **sections** construct, a worksharing construct in which several blocks of code are distributed to exactly one processor apiece, and each block is executed once [24]. The blocks in this case are the earliest start time and last completion time sections; while these sections consist mostly of identical code, their setup differs in several particulars (e.g. the negation of the bounds for the adjustment of last completion time, the alternating use of estList and lctList, etc.), and making these differences explicit in the code was deemed the simplest way to proceed.

The sole required point of synchronization of these two sections is the requirement for each to wait for the other to finish sorting the task list before proceeding with the detection phase. This synchronization is accomplished by means of a pair of OpenMP *lock variables*, one each for estList and lctList. After initialization, an OpenMP lock variable may take one of two states, *unlocked* or *locked*. A process may attempt to acquire a lock by *setting* it; if the lock variable is unlocked, the process acquires it, thereby changing the state to locked. Any other process attempting to acquire the locked variable will block execution. When the process that acquired the lock has finished with the lock, it releases it, and the variable becomes unlocked again; a process waiting for that same lock (if one exists) will then be woken up, will acquire the lock itself, and so forth [24].

In this instance, the two locks are set to locked at the start of the algorithm. When the earliest start time section has finished sorting estList, it unlocks the corresponding lock variable; when that same section has finished initializing the tree, it attempts to acquire the lock for the lctList, which will only be available if that list has been sorted by the other processor. Having acquired that lock, the section may proceed with the detection phase.

4.4.2 Update Precomputation Parallel Version

In the remaining two versions, these parallel section were augmented with an parallelization of the update precomputation. This parallelization was formed with the **task** construct, available in OpenMP version 3.0. In OpenMP, a parallel task differs from a parallel section: sections, as mentioned above, are

```
1  boundParallelEF(taskList, numTasks, C) begin
3      // Initialize Tasks...
4      ParallelSection
5          orderType ← estOrderType
6          estList ← sortByEst(taskList)   // tasks by increasing est
7          unset_lock(estLock)
8          prec ← allocate(integer array of size i)
9          buildTree(tree, estList, i, C)
10         set_lock(lctLock)
11         detectOrder(tree, estList, lctList, prec, i, C)
12         for t ∈ estList do
13             if ! updateContains(c_t) then
14                 calculateAllUpdate(tree, c_t)
15         for i ∈ lctList do
16             for j ∈ lctList do
17                 if lct_j ≤ prec[estRank_i] then
18                     adjBound_i ← updateGet(update, j, c_i)
19         for i ∈ estList do
20             syncLB(index_i, adjBound_i)
21     ParallelSection
22         orderType ← lctOrderType
23         lctList ← sortByLct(taskList)   // tasks by decreasing lct
24         unset_lock(lctLock)
25         prec ← allocate(integer array of size i)
26         buildTree(tree, lctList, i, capacity)
27         set_lock(estLock)
28         detectOrder(tree, lctList, estList, prec, i, C)
29         for t ∈ lctList do
30             if ! updateContains(c_t) then
31                 calculateAllUpdate(tree, c_t)
32         for i ∈ estList do
33             for j ∈ estList do
34                 if − est_j ≤ prec[lctRank_i] then
35                     adjBound_i ← updateGet(update, j, c_i)
36         for i ∈ lctList do
37             syncUB(index_i, − adjBound_i)
```

Algorithm 4.2 – boundParallelEF divides the edge-finding algorithm into two parallel sections, one for each bound to be adjusted. The two sections must synchronize once, to ensure that the bound list sorted by the other section is complete.

```
1  updateParallelEF(taskList, numTasks, C) begin
3      // Initialize Tasks...
4      ParallelSection
5          orderType ← estOrderType
6          estList ← sortByEst(taskList)   // tasks by increasing est
7          unset_lock(estLock)
8          prec ← allocate(integer array of size i)
9          buildTree(tree, estList, i, C)
10         set_lock(lctLock)
11         for t ∈ estList do
12             if ! updateContains(c_t) then
13                 ParallelTask
14                     copyThetaTree(tree, localTree)
15                     calculateAllUpdate(localTree, c_t)
17         // Wait for all tasks to complete
18         detectOrder(tree, estList, lctList, prec, i, C)
19         for i ∈ lctList do
20             for j ∈ lctList do
21                 if lct_j ≤ prec[estRank_i] then
22                     adjBound_i ← updateGet(update, j, c_i)
23         for i ∈ estList do
24             syncLB(index_i, adjBound_i)
25     ParallelSection
27         // Repeat for lct adjustment section...
```

Algorithm 4.3 – updateParallelEF adds a parallelization of the update precalculation.

each executed by a different processor immediately, whereas tasks are placed in a queue. Available processes execute the tasks in this queue as they become available [24]. This distinction is advantageous here due to the unknown number of update calculations that need to be performed; recall that $\mathsf{update}(j, c)$ is computed for every *distinct* value of c in the set of tasks. Using the OpenMP task queue, the main thread simply loops through all the tasks. For each task j, if c_j has not already been queued for precomputation, it is queued; if it has already been queued, it is ignored. This queueing can take place at the beginning of the parallel section (earliest start time or last completion time adjustment), and the master thread for that section may then proceed with the detection and adjustment phases.

In the simpler, second version (Algorithm 4.3), the tasks are synchronized with the end of the order detection phase by means of an OpenMP **taskwait** directive. This construct causes the current process to halt execution until all child threads spawned by the process have finished. In this case, the master thread for each section blocks before starting the adjustment phase; once all precomputations are complete, the adjustment phase continues.

Clearly this synchronization introduces a certain amount of inefficiency; the adjustment of the bound for task i only requires that the computation of $\mathsf{update}(j, c_i)$ is complete, not that all precomputations are complete. Furthermore, some of these precomputations are never required for any adjustment.

```
1  updateParallelEFSynchronized(taskList, numTasks, C) begin
3     // Initialize Tasks...
4     ParallelSection
5        orderType ← estOrderType
6        estList ← sortByEst(taskList)   // tasks by increasing est
7        unset_lock(estLock)
8        prec ← allocate(integer array of size i)
9        buildTree(tree, estList, i, C)
10       set_lock(lctLock)
11       for t ∈ estList do
12          set_lock(estUpdateLock [t])
13       for t ∈ lctList do
14          if ! updateContains(c_t) then
15             ParallelTask
16                copyThetaTree(tree, localTree)
17                calculateAllUpdate(localTree, c_t)
18                for k ∈ lctList do
19                   if c_k = c_t then
20                      unset_lock(estUpdateLock [k])
22       // Wait for all tasks to complete
23       detectOrder(tree, estList, lctList, prec, i, C)
         // Adjust Bounds:
24       for i ∈ lctList do
25          for j ∈ lctList do
26             if lct_j ≤ prec[estRank_i] then
27                set_lock(estUpdateLock[lctRank_i])
28                adjBound_i ← updateGet(update, j, c_i)
29                unset_lock(estUpdateLock[lctRank_i])
30       for i ∈ lctList do
31          syncLB(index_i, adjBound_i)
32    ParallelSection
34       // Repeat for lct adjustment section...
```

Algorithm 4.4 – In updateParallelEFSynchronized, the adjustment phase for task i is free to commence once update(j, c_i) is computed.

In the third and final implementation (Algorithm 4.4), each section uses a set of locks (one for each task). These are initially all locked; once update(j, c) is computed for a value of c, the lock corresponding to each task with the same c value is released. This allows the master thread to start the adjustment procedure before all the update tasks are complete. Inside the adjustment function, the master thread contends for the lock of each task it is trying to adjust.

4.5 Results

Runtimes for the various implementations are shown in table 4.1. Tests were performed with task sets of increasing size. For each task set size, 100 task

number of tasks	serial	bounds	bounds+update	bounds+update(sync)
20	227	233	467	437
40	927	645	670	647
60	1908	1190	881	866
80	3795	2228	1222	1178
100	5743	3253	1579	1557
150	13953	7863	3005	2897
200	24458	13788	4456	4481
250	37973	20981	6766	6502
300	61743	33866	9655	10041
350	81808	45089	13040	12863
400	105951	58693	16417	16369
450	132934	74941	20198	20258
500	167094	90628	24869	24745

Table 4.1 – Runtimes (in μ-seconds) for serial and parallel edge-finding implementations.

sets were randomly generated. These task sets were then run through each of the different implementations, and the average time to completion is reported. Tests were performed on an 8 processor, i386 machine running SunOS 5.10. For the parallel update versions, thread count was limited to 16 threads.

Surprisingly, measurable speedup was observed even at small problem sizes (see figure 4.3). The simple bounds parallel approach achieved a speedup of nearly 2 for problems of 100 or more tasks. However, the two parallel update versions performed far better, with speedups approaching 7 on large problem sizes. Interestingly, these two implementations did not differ significantly from each other. The most likely explanation is that the added overhead of the locks, and the time spent in contention for them, balanced out with the time gained by starting the adjustment phase earlier.

4.6 Conclusion

The parallel version of the edge-finder performed better than expected. Both the parallel bounds version and the simple parallel update version seem to be justified enhancements. The manually synchronized update version did not result in any improvement, making it probably not a worthwhile path at the moment. It is possible that it could be improved using non-blocking testing for the locks in the adjustment stage, or similarly by spawning threads only to handle updates that might actually be required, similar to the approach taken in the serial version. However, the speedup provided by the simple versions shown here is very encouraging.

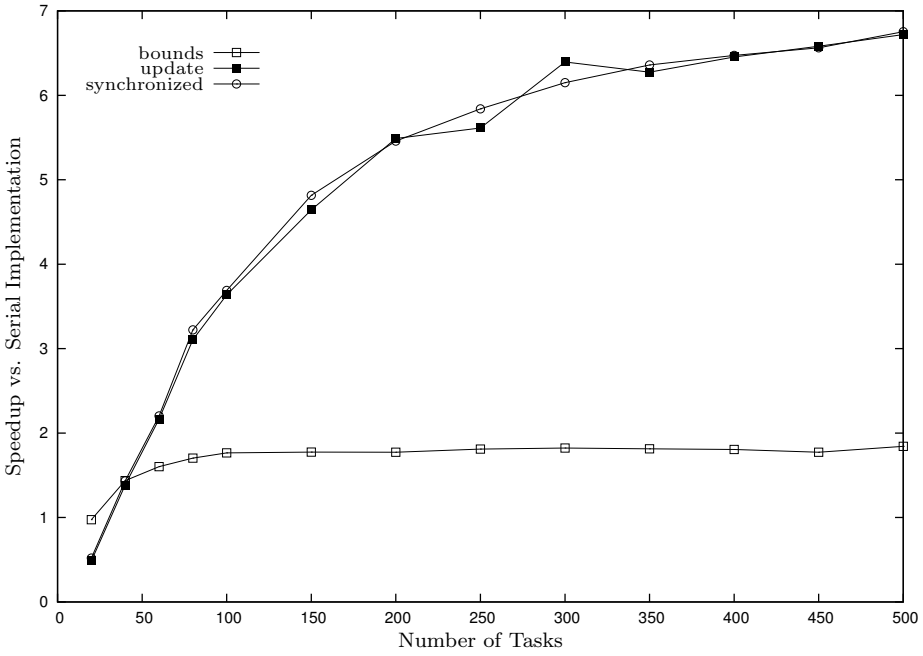


Figure 4.3 – Speedup of parallel implementations versus the serial version.

Chapter 5

Max Energy

5.1 Introduction

The cumulative edge-finding algorithm introduced in Chapter 2 makes adjustments to the earliest start time and last completion time of tasks. There exist several other algorithms for cumulative scheduling that make similar adjustments, such as Not-First/Not-Last [34] and Energetic Reasoning [7]. Less common are cumulative scheduling algorithms that filter variable duration and capacity requirements. One of these latter algorithms, presented in [40], is the max energy filtering algorithm. This algorithm uses a modified version of overload checking to determine maximum feasible duration and resource requirement for tasks, and is implemented again using the Θ-tree.

5.1.1 Maximum Energy

The overload rule for cumulative resources was given as Definition 1.6; in Section 3.3.5 the rule was reformulated as:

$$\forall j \in T : (\mathsf{Env}(\mathsf{LCut}(T, j)) > \mathcal{C}\,\mathsf{lct}_j) \implies \textbf{overload}) \tag{5.1}$$

This overload check was implemented as part of the edge-finding detection loop, but of course it could be easily implemented as a stand-alone filter using a Θ-tree as follows: the tasks $j \in T$ are added to a Θ-tree in increasing order of lct_j, thereby assuring that the contents of the tree always represent $\mathsf{LCut}(T, j)$ for the current task j; for each task, an $\mathcal{O}(\log n)$ operation yields $\mathsf{Env}_{root} = \mathsf{Env}(\mathsf{LCut}(T, j))$; e-feasibility is then verified using (5.1) (see [40, Algorithm 1.1]). This is not a new idea: a similar $\mathcal{O}(n \log n)$ algorithm for overload checking using balanced binary trees is given in [43].

If the duration and resource requirements of the tasks are allowed to vary, however, a new question presents itself: what is the maximum amount of energy a set of tasks could be allowed without causing an overload? In this light, we can reformulate (5.1) to reveal the maximum feasible energy envelope of a set $\mathsf{LCut}(T, j)$ [40, Equation (8)]:

$$\overline{\mathsf{Env}}(\mathsf{LCut}(T, j)) = \mathcal{C}\,\mathsf{lct}_j$$

Starting with this value in the root of the Θ-tree, it is possible to propagate down into the leaves the values of $\overline{\mathsf{Env}}$ and $\bar{\mathsf{e}}$, which represent the maximum

feasible energy envelope and maximum feasible energy of the tasks. Recall that the relationship between the energy envelope of a node and its child nodes was given as [39, (4),(5)]:

$$\begin{aligned} \mathsf{e}_v &= \mathsf{e}_{left(v)} + \mathsf{e}_{right(v)} \\ \mathsf{Env}_v &= \max\{\mathsf{Env}_{left(v)} + \mathsf{e}_{right(v)}, \mathsf{Env}_{right(v)}\} \end{aligned}$$

The downward propagation of $\overline{\mathsf{Env}}$ and $\bar{\mathsf{e}}$ essentially reverses this relationship: given maximum feasible values in the parent node, maximum feasible values for each child node are deduced as such [40, (11) - (14)]:

$$\begin{aligned} \overline{\mathsf{Env}}_{right(v)} &= \overline{\mathsf{Env}}_v \\ \overline{\mathsf{Env}}_{left(v)} &= \overline{\mathsf{Env}}_v - \mathsf{e}_{right(v)} \\ \bar{\mathsf{e}}_{right(v)} &= \min\{\overline{\mathsf{Env}}_v - \mathsf{Env}_{left(v)}, \bar{\mathsf{e}}_v - \mathsf{e}_{left(v)}\} \\ \bar{\mathsf{e}}_{left(v)} &= \bar{\mathsf{e}}_v - \mathsf{e}_{right(v)} \end{aligned} \tag{5.2}$$

After this propagation is completed, the maximum energy for any task $i \in \mathsf{LCut}(T, j)$ is given by [40, (15)]:

$$\bar{\mathsf{e}}_i \leq \min\{\bar{\mathsf{e}}_v, \overline{\mathsf{Env}}_v - \mathcal{C}\,\mathsf{est}_i\}$$

Performing all of these propagation steps yields an algorithm with a time complexity of $\mathcal{O}\left(n^2\right)$ (this is given as [40, Algorithm 1.2]); however, it is possible to reduce the complexity to $\mathcal{O}\left(n \log n\right)$ by observing that many several propagation steps may be skipped in each iteration, effectively postponing them until they are required later in the algorithm. The delayed propagation must be performed when a new task is added to Θ; at that point, the values are propagated downwards from all nodes lying on a path from the root node to the node corresponding to the task to be added. Up until this time, any values that had been propagated downward into interior nodes may be left there; if further propagation results in new potential values for these interior nodes, the minimum value is retained and the other discarded.

The rules for this new propagation algorithm are given as [40, (16) - (19)], and have been implemented here as Algorithm 5.1.

```
1 pushDown2(v) begin
2     r ← 2 * v + 2
3     l ← 2 * v + 1
4     Env̄_r ← min(Env̄_v, Env̄_r)
5     Env̄_l ← min(Env̄_v − e_r, Env̄_l)
6     ē_r ← min(Env̄_v − Env_l, ē_v − e_l, ē_r)
7     ē_l ← min(ē_v − e_r, ē_l)
```

Algorithm 5.1 – pushDown2(v) adjusts the $\overline{\mathsf{Env}}$ and $\bar{\mathsf{e}}$ values of the left and right children of node v. Note that pushDown2 does not propagate these changes any further down the tree; algorithms 5.3 and 5.5 use pushDown2 to perform calculations based on a single node, and then propagate the altered values as needed.

5.2 Θ-tree modifications

The tree used in the max energy filter is a relatively straightforward Θ-tree, as opposed to the significantly more involved Θ-Λ-tree used in the edge-finding algorithm. Each node only needs to track four values: Env, e, $\overline{\mathsf{Env}}$, and $\bar{\mathsf{e}}$. The first two of these values are handled in a nearly identical fashion to their counterparts in the edge-finding algorithm: values are calculated for the leaf nodes (corresponding to individual tasks), and then propagated up through the tree using (2.4) and (2.5). $\bar{\mathsf{e}}$ and $\overline{\mathsf{Env}}$ propagate from the root node down to the leaves; this a wholly new operation, and is handled in Algorithms 5.1 and 5.3.

The remainder of the Θ-tree operations are shown in Algorithm 5.2. The added feature is the initialization of $\bar{\mathsf{e}}$ and $\overline{\mathsf{Env}}$ to ∞, for both leaf and interior nodes.

The other change is the handling of energy envelope for "empty" nodes—that is, nodes that are acting as structural placeholders for tasks that have been removed from Θ. In the edge-finding algorithm, the correct calculation of Env^{Λ} requires that these nodes have $\mathsf{Env} = -\infty$ [40, Section 6.1]. In the max energy algorithm, this use of $-\infty$ leads to some incorrect calculations of $\bar{\mathsf{e}}$ in right child nodes during the downwards propagation in pushDown2, based on (5.2). For correct propagation of these values, the energy envelope of empty nodes must be set to 0.

5.3 Algorithm

5.3.1 Push Down Path

As outlined in Section 5.1.1, the $\mathcal{O}(n \log n)$ version of the max energy algorithm does not immediately perform the downwards propagation of $\bar{\mathsf{e}}$ and $\overline{\mathsf{Env}}$ values. Instead, these values are propagated only on a path from the root node to the node corresponding to the newly added task j. The means for determining this path are not provided in [40], but are instead left to the reader. An algorithm for tracing the path is not particularly challenging; it is, however, different from any method of navigating the Θ-tree discussed thus far, and therefore merits a brief discussion.

There are two instances of root-to-leaf traversal of the Θ-tree in the edge-finding algorithm: the location of the node responsible for $\mathsf{Env}(\Theta, \Lambda)$ [39, end of Section 6.1], and the location of $\mathsf{minest}(j, c)$ [39, Algorithm 2]. In each of these cases, the location of the leaf node is not known in advance, but some property of the task is known. A comparison is made at each interior node, and based on the results of this comparison the traversal moves into either the left or right child node.

The current case is quite different. Here the location of the leaf node is known: the task to be added to Θ has an index that may be translated directly into the index of the corresponding node. The only determination that needs to be made at each interior node, therefore, is whether the leaf node in question lies in the left or right subtree of the current node, which can be simply determined through an enumeration of the leaf nodes. The leaves are numbered from left to right; a middle value is calculated as half the maximum number of possible leaves for a tree of that size. If the target leaf has a value greater than half the

```
1  buildTree(estList, n, C) begin
2      size ← 2^⌈log2(n)⌉ + n − 1
3      tree ← allocate(array of size nodes)
4      tree.numTasks ← n
5      tree.capacity ← C
6      tree.size ← size
7      for i = tree.size − 1 to tree.size − tree.numTasks do
8          computeLeafVals(tree,estList, i, C)
9      for i = tree.size − tree.numTasks − 1 to 0 do
10         computeNodeVals(tree, i, C)
11     return tree
12 computeLeafVals(tree, estList, i, C) begin
13     t ← order[nodeToTask(i)]
14     e_i ← c_t * p_t
15     Env_i ← C * est_t + e_i
16     ē_i ← ∞
17     Ēnv_i ← ∞
18 computeNodeVals(tree, i, C) begin
19     if left(i) does not exist then
20         clearNode(tree, i)
21     else if right(i) does not exist then
22         tree[i] ← tree[left(i)]
23     else
24         e_i ← e_left(i) + e_right(i)
25         Env_i ← max(Env_left(i) + e_right(i), Env_right(i))
26         ē_i ← ∞
27         Ēnv_i ← ∞
28 clearNode(tree, i) begin
29     e_i ← 0
30     Env_i ← 0
31     ē_i ← ∞
32     Ēnv_i ← ∞
```

Algorithm 5.2 – Operations for a max energy Θ-tree, similar to the Θ-tree operations presented in Algorithms 3.1 and 3.2. Note that in clearNode, the $\mathsf{Env} = 0$ for an empty node, as opposed to the $-\infty$ value used in the edge-finding Θ-tree.

middle value, it is in the right subtree, otherwise it is in the left.

After moving to the left or right child node, the subtree rooted at the new node will have half the leaves of the previous tree. If the leaf node was last in the left subtree, then it retains its previous leaf number; if it was in the right subtree, its new leaf number is obtained by subtracting the previous middle value; the middle value itself is simply divided in two. This process is repeated until the leaf node is reached, with the pushDown2 procedure executed on each node encountered along the way (see Algorithm 5.3).

```
1  ADD(tree, j) begin
2      v ← root
3      midpoint ← 2^(depth−1)
4      leafnum ← j − (tree.size − tree.numTasks)
5      while v is not a leaf do
6          midpoint ← midpoint/2
7          PUSHDOWN2(v)
8          ē_v ← ∞
9          Env̄_v ← ∞
10         if leafnum ≥ midpoint then
11             leafnum ← leafnum − midpoint
12             v ← 2 ∗ v + 2
13         else
14             v ← 2 ∗ v + 1
       Ensures : v == j
15     ENABLENODE(tree, j)
```

Algorithm 5.3 – ADD(Θ, j) activates the leaf node j. Before doing so, it must push down any stored values for $\bar{e}$ or $\overline{Env}$ in the nodes on a path from root to j, and then clear the stored values.

5.3.2 Inactive Nodes

The procedure PUSHDOWN2 is not provided in [40], but is instead described in terms of the functions that it implements; specifically, PUSHDOWN2 implements [40, Equations (16) - (19)]. This set of equations describes the propagation of maximum energy and energy envelope values from the root of theΘ-tree down to its leaves. Implementation of the procedure itself is trivial, and is presented as Algorithm 5.1.

In practice, the structure of the Θ-tree remains static throughout the algorithm, and leaf nodes corresponding to tasks not currently in the set Θ simply hold values that cannot affect the calculation of interior nodes above them. While these placeholder values prevent the improper *upwards* propagation of values of tasks not in Θ, there is nothing to prevent PUSHDOWN2 from propagating values downward into these nodes. As seen in Algorithm 5.5, Θ is built over the course of the algorithm through the addition of tasks in order of non-decreasing last completion time, so lct_Θ increases as the algorithm moves forward. As the maximum energy available depends on lct_Θ, this maximum energy also increases.

If PUSHDOWN2 propagates values based on the initial, low maximum energy values into nodes not yet included in Θ, then these values will not be overwritten by subsequent execution. The problem is that PUSHDOWN2 cannot automatically overwrite $\bar{e}$ and $\overline{Env}$ values in the children nodes; instead it compares the new value generated by downward propagation to the existing value in those nodes, and uses the minimum of these values. Any values propagated into these nodes before the nodes were added to Θ will always be less than the correct values propagated later, when the Θ-tree represents a higher energy set.

To prevent this, these incorrect values may simply be overwritten when the

node is added to Θ, setting:

$$\overline{\mathsf{e}}_v = \infty$$
$$\overline{\overline{\mathsf{Env}}_v} = \infty$$

Note that these are the same values used to initialize all the nodes of the tree [40, following Equation (19)]; however, this initialization will be subsequently overwritten by calls to pushDown2, and hence must be refreshed when each leaf node is added to Θ. Nodes are never removed from Θ, so it is correct to consider all the values generated by propagation after the node is added, and there is no need to reset the leaf node values again.

```
1 setEnergyMax(v)begin
2     t ← nodeToTask(v)
3     ē_v ← min(ē_v, Env̄_v − C est_t)
4     c̄_t ← min(⌊ē_v / p_t⌋, c̄_t)
5     p̄_t ← min(⌊ē_v / c_t⌋, p̄_t)
```

Algorithm 5.4 – setEnergyMax(v) calculates the maximum values for c and p for the task corresponding to the node v.

```
1  maxEnergy(taskList, n, C)begin
2      estList ← sortByEst(taskList )
3      lctList ← sortByLct(taskList )
4      tree ← buildTree(estList, n, C)
5      clearTree(tree)
6      for j ∈ lctList in non-decreasing order of lct_j do
7          add(tree, j)
8          if Env_root > C lct_j then
9              return FAIL
10         Env̄_root ← C lct_j
11     v ← 0
12     while v < tree.size − tree.numTasks − 1 do
13         pushDown2(tree[v])
14         v ← v + 1
15     while v ≥ 0 do
16         setEnergyMax(tree[v])
17         v ← v + 1
18     return SUCCESS
```

Algorithm 5.5 – maxEnergy(taskList, n, C) computes the maximum c and p values for the tasks in taskList. It fails if it detects an overload situation as defined by e-feasibility.

5.4 Combining Edge-Finding and Max Energy

The primary loop orders in edge-finding and max energy are different: edge-finding order detection occurs in non-increasing order by lct_j [39, Algorithm 1],

while max energy is performed in non-decreasing order over the same bound [40, Algorithm 1.3]. So instead of running the two filters inside the same loop, they are run in sequence. The two filters can be implemented to share a Θ-tree, saving on both setup time and space requirements.

The results of edge-finding depend on the required capacity and resource demand of each task, values that are referred to in the max energy filter as the minimum bounds for c and p. The maximum bounds for c and p ($maxC$ and $maxP$, respectively) are the variables that the max energy filter adjusts; however, these values are irrelevant in the edge-finding filter. On the other hand, the max energy filter is dependent on the earliest start time and last completion time of the tasks, values that may be strengthened by the edge-finding filter. Running the max energy filter using the bounds adjusted by the edge-finding filter should therefore result in better propagation.

Each of these filters performs the same overload check, so one of these checks could be eliminated. Since earliest start time and last completion time may have been adjusted between the two overload checks, it is possible for the second to fail in some cases where the first did not. In the interests of of detecting failure as early as possible (thus eliminating unnecessary and costly filtering), both of these overload checks have been left in the algorithm.

As discussed in Section 5.2, the treatment of energy envelope for empty nodes differs between the two algorithms. This difference is handled by storing the four values used by max energy in a separate subnode (the same way that Θ and Λ values were separated in the edge-finding algorithm). The use of a new subnode results in a minor duplication of data (the values of energy and energy envelope stored in the Θ-node and the max-node will be nearly always identical), but it also makes it simpler to isolate the effects of the calculations of each filter, ensuring correct results.

Chapter 6

Conclusion

The application of the Θ-tree to cumulative scheduling problems brings to cumulative scheduling filters a substantial reduction in time complexity, but comes at the cost of a significantly more complicated algorithm. Without a prior understanding of edge-finding, parts of the algorithm presented in [39] can be quite opaque; it would be unfortunate if this opacity were to discourage potential implementors from tackling this quite beautiful algorithm. It is the author's hope that this thesis can serve those uninitiated in edge-finding as both an introduction to the theory as well as a clarification of some of the more challenging sections of the original paper.

Of the latter, the update calculation section [39, Section 7]is perhaps most in need of further explanation. Towards this end significant space has been spent here attempting to shed light on both the idea underlying this portion of the algorithm (including a revision to the proof in the original paper), and to dealing with those parts of the update procedure omitted from the original.

Additionally, several new contributions to the algorithm are presented here, perhaps the most significant being the non-cutting algorithm for calculating the values of the α and β subtrees during the calculation of $\mathsf{Env}^{\mathsf{c}}$.

The results of the parallel version of the edge-finding algorithm were encouraging, and perhaps merit further investigation. Also, the combination of the edge-finding and max energy algorithms allows for some handling of variable task resource requirements and durations in the filter, helping to broaden the range of real-world applications for this scheduling algorithm.

6.1 Future Work

The most interesting additional work in this area is probably the application of similar methods to other relaxations of the cumulative constraint. Recently, Schutt and Wolf have proposed an algorithm for cumulative not-first/not-last filtering that uses an innovative modification of the Θ-tree to achieve $\mathcal{O}\left(n^2 \log n\right)$ complexity [33]. As the authors note there, the existence of a $\mathcal{O}\left(kn \log n\right)$ cumulative edge-finding algorithm suggests that the complexity of cumulative not-first/not-last might be reducible to that same point. There is also a stronger variant of the edge-finding rule, commonly called *extended* edge-finding. Mercier and Van Hentenryck [22] provide an extended edge-finding algorithm with the

same complexity as their (non-extended) edge-finding algorithm ($\mathcal{O}\left(n^2k\right)$); this suggests that, using the Θ-tree, it might be possible to develop an $\mathcal{O}\left(kn\log n\right)$ algorithm for extended edge-finding as well.

Appendix A

Experimental Data

Number of Tasks (n)	Distinct Resource Req.s (k)			
	1		2	
	Θ	Std	Θ	Std
10	95	23	79	24
15	115	40	158	82
20	170	85	217	124
25	202	128	264	200
30	231	171	310	291
35	305	234	223	400
40	335	305	449	523
45	357	354	502	656
50	404	497	534	669
55	431	477	596	907
60	467	632	640	1071
65	584	836	793	1296
70	612	953	837	1486
75	662	1036	897	1797
80	699	1221	956	2116
85	404	1387	1024	2253
90	787	1342	1063	2457
95	817	1600	1108	2616
100	867	1789	1179	2921

(a)

Number of Tasks (n)	Distinct Resource Requirements (k)									
	1		2		3		4		5	
	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std
10	95	28	101	22	145	57	164	60	148	48
15	114	42	149	70	181	88	198	83	246	155
20	162	72	227	152	263	163	308	207	360	259
25	183	100	236	116	311	210	357	259	438	376
30	215	146	297	251	369	371	436	458	515	575
35	281	197	343	266	509	567	562	504	685	769
40	296	243	440	441	495	458	627	661	764	959
45	331	292	486	564	570	547	715	977	846	1299
50	363	361	539	648	613	756	794	1185	927	1416
55	441	600	559	693	713	1188	847	1387	1005	1745
60	464	720	558	803	760	1198	908	1641	1068	1933
65	523	674	721	1109	961	1603	1098	1829	1346	2468
70	560	800	719	1064	988	1742	1216	2395	1393	2499
75	603	746	847	1548	1098	2079	1279	2284	1538	3242
80	633	1018	924	1635	1080	1847	1353	2884	1652	3670
85	672	1134	974	2125	1121	2005	1449	3156	1710	3990
90	759	1275	921	1696	1331	3399	818	4059	1810	4445
95	813	1390	1062	2450	1299	2949	1602	4017	1937	5360
100	779	1487	1101	2576	1406	3602	1692	4365	1034	6127

(b)

Table A.1 – Edge-finding runtimes for $\mathcal{C} = 2$ and $\mathcal{C} = 5$.

Number of Tasks (n)	Distinct Resource Requirements (k)																			
	1		2		3		4		5		6		7		8		9		10	
	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std
10	86	17	107	26	145	54	123	31	166	58	164	54	176	37	137	85	185	65	194	57
15	108	31	143	57	180	95	184	76	193	89	218	112	245	115	239	81	284	155	251	121
20	148	48	192	85	238	117	259	111	294	172	385	300	211	265	460	266	483	309	429	279
25	179	78	262	198	319	245	368	319	407	301	429	373	497	421	552	480	599	530	621	623
30	208	124	275	195	378	445	406	306	465	416	492	355	647	798	675	722	739	829	851	1020
35	268	164	366	273	470	466	561	561	615	559	714	606	834	903	901	897	1046	1204	1134	1367
40	302	221	416	371	532	606	543	422	701	731	407	824	969	1238	1027	1228	1188	1598	1283	1713
45	336	279	457	468	525	471	738	1070	736	925	921	1251	1061	1549	1137	1533	1303	1949	1406	2095
50	371	343	506	573	611	756	736	1009	849	1139	1022	1600	1076	1486	1255	1975	1416	2271	1555	2664
55	406	423	541	678	611	742	796	1213	932	1379	1154	2083	1278	2213	1367	2371	1519	2604	1703	3252
60	429	500	588	831	750	1308	889	1575	996	1534	1246	2603	1430	3067	1576	3321	1667	3508	1839	3841
65	509	591	723	974	814	1092	1104	2003	1302	2169	1452	2441	1620	2829	1767	2769	2117	4194	2282	4534
70	548	744	766	1122	944	1354	1181	2138	1281	2094	1599	2947	1704	2942	1937	3625	2254	4665	2456	5273
75	596	750	812	1282	989	1412	1230	2154	1429	2393	1739	3570	1867	3714	2153	4682	2351	5165	2601	5981
80	638	1004	875	1498	1049	1606	1271	2194	1444	2662	1895	4306	1958	3893	2290	5507	2502	5922	2745	6691
85	706	991	939	1721	1105	1833	1485	3447	1567	2966	1775	3868	2187	5348	2410	6194	2614	6304	2914	7731
90	710	1213	990	1920	1169	2109	1461	3268	1732	4049	1979	4830	2214	5557	2557	6956	2783	7933	3097	8822
95	742	1270	985	2047	1239	2389	1594	4021	1834	4290	2216	6125	2353	6396	2703	7822	2962	8952	3141	9177
100	848	1631	1048	1944	1297	2647	1630	3789	1822	4233	2167	5490	2551	7494	2893	9175	3085	8665	3381	10384

Table A.2 – Edge-finding runtimes for $\mathcal{C} = 10$.

Number of Tasks (n)	Distinct Resource Requirements (k)																			
	1		2		3		4		5		6		7		8		9		10	
	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std
10	87	17	110	28	127	29	122	35	153	50	116	19	174	61	144	31	178	53	163	45
15	111	32	144	66	169	80	169	80	203	90	244	86	251	134	314	207	245	116	283	154
20	151	49	183	80	252	138	258	125	315	179	370	178	362	225	461	232	464	284	520	330
25	181	81	268	207	327	270	345	254	404	348	446	318	516	420	430	285	538	396	507	349
30	211	114	262	173	350	286	391	276	476	448	519	438	602	632	624	601	721	728	835	897
35	271	165	373	348	473	461	561	582	633	608	715	697	808	864	799	687	966	999	1102	1252
40	313	221	442	527	466	388	542	523	368	795	824	903	904	1113	1033	1297	1096	1301	1234	1556
45	343	276	427	404	611	771	604	516	757	857	915	1205	1008	1384	1058	1388	1213	1593	1396	2035
50	371	349	470	533	670	840	705	771	830	1024	918	1062	1108	1729	1314	2117	1332	1879	1531	2445
55	404	424	514	572	724	1183	817	1282	987	1549	1004	1522	1201	2115	1447	2613	1459	2309	1649	2968
60	433	500	541	675	738	1102	816	1263	1015	1585	1236	2312	1305	2504	1573	3145	1527	2657	1781	3549
65	589	886	775	1238	958	1660	1095	1647	1095	1298	1439	2535	1623	2902	1935	3652	1904	3173	2210	4235
70	611	887	765	1071	978	1558	1093	1634	1436	2863	1477	2477	1765	3374	2053	4176	2047	3737	2273	4129
75	587	844	859	1601	1030	1787	1199	1893	1444	2758	1459	2390	1721	3030	2190	4707	2273	4377	2377	4620
80	683	996	892	1767	1144	2190	1346	2831	1562	3429	1815	3895	1845	3424	2083	4066	2412	5074	2626	6254
85	657	953	987	2028	1039	1728	1369	2537	1606	3270	1994	4953	2082	4803	2391	5589	2560	5899	2597	4961
90	684	1316	905	1493	1227	2239	1443	2861	1766	4467	2088	5535	1161	5864	2446	6268	2774	7216	2914	7324
95	734	1328	972	1945	1293	2964	1513	3181	1930	5236	2196	6131	2278	5764	2697	7639	2723	6947	3092	8284
100	768	1477	583	2543	1337	3164	1597	3580	1938	5463	2296	6720	2632	8510	2660	7228	3100	8859	3234	9245

Number of Tasks (n)	Distinct Resource Requirements (k)																			
	11		12		13		14		15		16		17		18		19		20	
	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std	Θ	Std
10	202	77	196	70	279	148	310	161	274	120	336	189	466	234	652	450	513	370	615	441
15	282	154	276	119	285	135	329	161	272	120	336	189	465	232	652	449	514	370	614	439
20	503	400	513	367	569	357	574	347	507	332	487	226	638	346	760	622	564	401	612	442
25	637	550	750	671	721	622	680	545	717	598	816	680	639	557	950	923	801	740	831	793
30	626	582	888	938	846	943	915	844	887	952	819	859	894	817	999	1054	906	841	943	860
35	1060	1192	1185	1275	1264	1494	1446	1821	1442	1596	1382	1574	1670	1855	1528	1770	1444	1535	1696	2080
40	1365	1740	1325	1586	1581	2097	1572	2041	1779	2526	1497	1696	1825	2273	1837	2558	1902	2735	1959	2710
45	1436	1810	1681	2614	1738	2650	1607	2294	1685	2273	1929	2847	2197	3391	1908	2827	988	3132	2195	3398
50	1698	3000	1709	3066	1993	3402	1956	3130	1951	2991	1498	4003	2186	3406	2273	4019	2275	3379	2587	4135
55	1788	3364	1943	3612	2094	3891	2217	4090	2255	3971	2334	4101	2456	4312	2744	5089	2903	5988	1449	4981
60	1867	3490	2030	3433	2220	4494	2207	4472	2407	4707	2649	5486	2606	5090	3045	6156	2922	5827	2806	5691
65	2490	4914	2471	4517	2777	5329	2905	5291	2998	5454	3362	6467	3278	5889	1852	6647	3556	6508	3991	8282
70	2423	4605	2770	5254	2959	6187	3117	7005	3168	6293	3465	7043	3657	7490	3890	8967	4012	8728	2537	8471
75	2741	6029	2865	6030	3166	7049	3349	6921	3595	7939	3687	7362	4011	8894	4042	9728	4373	10409	4600	10583
80	2955	6561	3074	7017	3448	8557	3661	9359	3744	9115	3998	9977	4199	10010	4302	10929	4371	10973	4825	12187
85	3145	7541	3350	8673	3660	9847	3738	8766	4111	10076	4097	10805	4574	12073	4938	13646	4357	10927	5193	13952
90	3341	8802	3642	10131	3858	11106	3848	9115	4028	10691	4148	10440	2357	13633	2498	14990	3926	13610	5553	15923
95	3440	9243	3817	11157	3975	11616	4047	10064	4401	12123	4820	14524	5007	14462	5011	14277	5450	16750	5714	16761
100	1834	11899	3949	11976	3926	11555	4429	13410	4669	14411	4848	14776	5190	16495	5122	15648	5666	17421	6107	18833

Table A.3 – Edge-finding runtimes for $\mathcal{C} = 20$.

Appendix B

Solution to the sudoku puzzle

Figure B.1 – Solution to the sudoku puzzle on page 3

6	9	3	7	8	4	5	1	2
4	8	7	5	1	2	9	3	6
1	2	5	9	6	3	8	7	4
9	3	2	6	5	1	4	8	7
5	6	8	2	4	7	3	9	1
7	4	1	3	9	8	6	2	5
3	1	9	4	7	5	2	6	8
8	5	6	1	2	9	7	4	3
2	7	4	8	3	6	1	5	9

Bibliography

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[2] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *INFORMS Journal on Computing*, 3(2):149–156, 1991.

[3] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, England, 2003.

[4] Kenneth R. Baker and Dan Trietsch. *Principles of Sequencing and Scheduling.* John Wiley & Sons, Hoboken, New Jersey, 2009.

[5] Philippe Baptiste and Claude Le Pape. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Proceedings of the fifteenth workshop of the UK planning special interest group*, volume 335, pages 339–345, 1996.

[6] Philippe Baptiste, Claude Le Pape, and W. P. M. Nuijten. Satisfiability tests and time-bound adjustmentsfor cumulative scheduling problems. *Annals of Operations Research*, 92:305–333, 1999.

[7] Philippe Baptiste, Claude Le Pape, and W. P. M. Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems.* Springer-Verlag, 2001.

[8] Peter Brucker. *Scheduling Algorithms.* Springer, Berlin Heidelberg, Fifth edition, 2007.

[9] Peter Brucker and Sigrid Knust. *Complex Scheduling.* GOR Publications. Springer, Berlin Heidelberg, 2006.

[10] Jacques Carlier and Eric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78(2):146–161, 1994.

[11] Yves Caseau and François Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of the eleventh international conference on Logic programming*, pages 369–383. MIT Press, 1994.

[12] Yves Caseau and François Laburthe. Cumulative scheduling with task intervals. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 363–377. The MIT Press, 1996.

[13] Rina Dechter. *Constraint Processing.* Morgan Kaufmann, San Francisco, 2003.

[14] Rina Dechter. Tractable structures for constraint satisfaction problems. In Rossi et al. [27], chapter 7, pages 209–244.

[15] Andries P. Engelbrecht. *Computational Intelligence: An Introduction.* John Wiley & Sons, second edition, 2007.

[16] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible sudoku grids. [PDF Document], 2005. Retrieved from: http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf.

[17] Eugene C. Freuder and Alan K. Mackworth. Constraint satisfaction: An emerging paradigm. In Rossi et al. [27], chapter 2, pages 13–27.

[18] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming.* Springer, Berlin, 2003.

[19] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[20] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing.* Addison Wesley, second edition, 2003.

[21] George F. Luger. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* Addison-Wesley, 6$^{\text{th}}$ edition, 2009.

[22] Luc Mercier and Pascal Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20:143–153, 2008.

[23] W. P. M. Nuijten. *Time and resource constrained scheduling : a constraint satisfaction approach.* PhD thesis, Technische Universiteit Eindhoven, 1994.

[24] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2008. Version 3.0. Accessed from http://www.openmp.org/mp-documents/spec30.pdf.

[25] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization.* Prentice-Hall Englewood Cliffs, NJ, 1982.

[26] Jean-Charles Règin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1, pages 362–367. AAAI Press, 1994.

[27] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming.* Foundations of Artificial Intelligence. Elsevier, Amsterdam, 2006.

[28] Gordon Royle. Minimum sudoku. Retrieved from: http://mapleta.maths.uwa.edu.au/~gordon/sudokumin.php.

[29] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency.* Springer Verlag, 2003.

[30] Christian Schulte. Course Notes: Constraint Programming (ID2204). [PDF Document], 2009. Retrieved from: http://www.ict.kth.se/courses/ID2204/notes/ID2204-CN.pdf.

[31] Christian Schulte. Personal communication, February 2010.

[32] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In Rossi et al. [27], chapter 14, pages 495–526.

[33] Andreas Schutt and Armin Wolf. A new $O(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 445–459. Springer Berlin / Heidelberg, 2010.

[34] Andreas Schutt, Armin Wolf, and Gunnar Schrader. Not-First and Not-Last Detection for Cumulative Scheduling in O (n^3 log n). *Lecture Notes in Computer Science*, 4369:66, 2006.

[35] Guido Tack and Christian Schulte. View-based propagator derivation, August 2009.

[36] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Rossi et al. [27], chapter 6, pages 169–208.

[37] Petr Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In Jean-Charles Régin and Michel Rueher, editors, *Proceedings of CP-AI-OR 2004*, volume 3011 of *Lecture Notes in Computer Science*, pages 335–347, Nice, France, April 2004. Springer-Verlag.

[38] Petr Vilím. *Global Constraints in Scheduling.* PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Univerzita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic, August 2007.

[39] Petr Vilím. Edge finding filtering algorithm for discrete cumulative resources in $O(kn \log n)$. In I.P. Gent, editor, *CP2009*, number 5732 in Lecture Notes in Computer Science, pages 802–816, Berlin/Heidelberg, 2009. Springer-Verlag.

[40] Petr Vilím. Max energy filtering algorithm for discrete cumulative resources. In *Proceedings of CP-AI-OR 2009*, number 5547 in Lecture Notes in Computer Science, pages 294–308, Berlin/Heidelberg, 2009. Springer-Verlag.

[41] Petr Vilím, Roman Barták, and Ondřej Čepek. Unary resource constraint with optional activities. In *Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2004.

[42] Petr Vilím, Roman Barták, and Ondřej Čepek. Extension of $O(n \log n)$ filtering algorithms for the unary resource constraint to optional activities. *Constraints*, 10(4):403–425, October 2005.

[43] Armin Wolf and Gunnar Schrader. $O(n \log n)$ overload checking for the cumulative constraint and its application. In M. Umeda, Armin Wolf, O. Bartenstein, U. Geske, D. Seipel, and O. Takata, editors, *INAP 2005*, volume 4369 of *LNCS*, pages 88–101. Springer, 2006.

[44] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.