Imperial College London

MEng. Individual Project

Imperial College of Science, Technology and Medicine

Department of Computing

Explaining Makespan Scheduling

Draft Report

Author:
Myles Lee

Supervisors:
Dr. Kristijonas Cyras
Prof. Francesca Toni

Second Marker:
Dr. Ruth Misener

April 2019

Abstract

Scheduling arises in countless decision processes and has a wide range of practical applications, such as in healthcare. Scheduling problems are modelled using mathematics, where optimisers can find solutions to large scheduling problems quickly. Such optimisers are often complex with non-accessible formulations of scheduling, resulting in users interpreting solvers and solutions as black-boxes. Users require a means to understand why a schedule is reasonable, which is hindered by black-box interpretations.

We present a tool, *Schedule Explainer* that explains any makeshift schedule easily with clarity. We will explore practical considerations of applied argumentation and extensions to makeshift scheduling, highlighted by the tool.

Contents

Chapter 1

Introduction

1.1 Motivation

Scheduling arises in countless decision processes and its abstract nature results in a wide range of practical applications, such as in healthcare [1]. Scheduling problems are accurately modelled in mathematics, hence scheduling is often interpreted as a class of mathematical optimisation problems. With many mature and developed optimisation solvers [2], solvers can find solutions to large scheduling problems quickly, which would have been impractical using manual optimisation techniques. However, solver scalability and efficiency often leads to complex algorithms. Such complexity combined with mathematical formulations of scheduling, result in users interpreting solvers and solutions as black-boxes. Therefore, decision making is often not transparent.

Transparency of decision making is important. Users require a means to understand why a schedule, whether a solution of a solver or not, is reasonable given a context. For example, hospital managers may seek understanding about the efficiency of their staff and resources. Schedules may need to be robust to unpredictable changes in staff and resources, such as a nurse being unavailable to attend patients due to a personal reasons. In other settings, schedules may need to minimise the number of staff to maximise profits while fulfilling all staff and patients requirements.

Explanations are vital in communication for understanding. With many possible schedules, and many variations of scheduling problems [3], it is impractical to manually-craft explanations for every schedule of every variation. This motivates a tool to generate clear explanations.

1.2 Objectives

The first main objective is to devise and implement a tool that explains an solver's scheduling solutions by human-understandable means, and allows the user to interact with the solver in a human-friendly manner. By using the explanation methodology outlined using argumentation [4], the user can easier understand scheduling in a practical environments such as nurse rostering and

dialysis scheduling. Hence, analysis of such methodology important to understand the applicability of argumentation in practical settings.

The second main objectives is to extend the theoretical or practical capabilities of argumentation for scheduling. Extensions may include interval scheduling, shop scheduling and job-dependent scheduling.

1.3 Challenges

A tool satisfying such objectives are subjected solving the challenges:

1. **Trust:** Users of the tool need to be confident that the explanations generated are true. This requires correctness of algorithms proposed and implemented.
2. **Accessibility:** Explanations generated from this tool are required to be accessible to people without domain-specific knowledge of optimisation or argumentation. The tool should be accessible to computer novices.
3. **Applicability:** Explanations should relate to makespan schedules. The tool should generate clear explanations promptly to users.
4. **Knowledge transfer:** Explanations are constructed using knowledge structures, commonly represented using natural languages or diagrams. A challenge would be to effectively explore the usefulness of using natural languages compared to diagrams.
5. **Background:** Explainable planning is relatively new research area compared to optimisation [5]. Challenges arises from finding related literate on the title.

1.4 Contributions

The main contributions of this report are listed below:

- A new tool *Schedule Explainer* that implements the concepts behind using argumentation for optimisation for makeshift scheduling.
- Theoretical extensions to makeshift scheduling including interval scheduling.
- Algorithmic optimisations of argumentation with scheduling.
- A discussion on applicability of argumentation models against various scheduling models.

Chapter 2

Background

2.1 Optimisation

Optimisation is the process to finding the optimal decision with respect to constraints given a set of possible decisions. The optimal decision is measured by a cost function, that typically determines a numerical value representing how good a decision is. The problem is to either find the optimal decision, either as a minima or maxima using systematic methods. Applications of optimisation occur in many practical situations such as in engineering, manufacturing, science. In engineering applications, accurate modelling of a problem may require discrete decisions and non-linear relationships, resulting in difficulty in finding arbitrary optimal decisions. There are a wide range of applications for optimisation, but in this project we focus with mixed-integer linear programming.

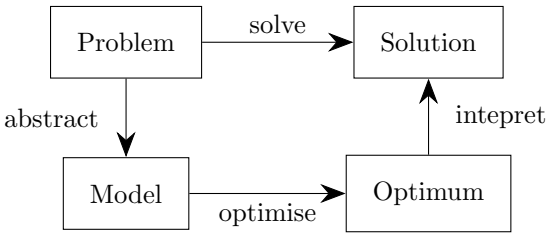


Figure 2.1: Abstraction of problem solving

Real-life problems need to be modelled to be optimised. Problems are formulated using constraints and a cost function. Consider the following linear programming example. A person want to sell some drinks. Each unit of hot chocolate requires 1 litre of milk and 3 bars of chocolate. Each unit of milkshake requires 1 litre of milk and 2 bars of chocolate. The person only has 5 units of milk and 12 bars of chocolate. The person sells a unit of hot chocolate for 6 and a unit of milkshake for 5. What is the best strategy for maximising profit given that all units produced are sold? The problem is abstracted as follows. Let x

and y be the number of hot chocolates and milkshakes produced.

$$\begin{aligned} \max_{x,y} \; & 6x + 5y && \text{subject to:} \\ & x + y \leq 5 && \text{milk resource constraint} \\ & 3x + 2y \leq 12 && \text{chocolate resource constraint} \\ & x, y \geq 0 && \text{non-negative constraints} \\ & x, y \in \mathbb{N} && \text{whole units only} \end{aligned}$$

The problem is sufficiently small to be solved by inspection, but may also be solved using graphical methods or a simplex algorithm. The optimum is $x^* = 2$ and $y^* = 3$, which is interpreted as that the best strategy is producing 2 units of hot chocolate and 3 units of milkshake.

In non-linear optimisation problems, finding global optimums is not trivial. For gradient-based and local-search algorithms, estimated solutions can return local optimums. This is typically not favoured however for large problems, finding global optimums have non-polynomial complexity for arbitrary dimensional problems. In context of the project, it is unfeasible to compute a complete explanation of optimality in polynomial time.

2.1.1 Makespan Scheduling

The simplistic definition of makespan scheduling gives a good foundation for experimenting with argumentation. Makespan schedules are defined by a $m \in \mathbb{N}$ independent machines and $n \in \mathbb{N}$ independent jobs [6]. Let $\mathcal{M} = \{1, ..., m\}$ be the set of machines and $\mathcal{J} = \{1, ..., n\}$ be the set of jobs. Each job $j \in \mathcal{J}$, has an associated processing time $p_j \in \mathbb{R}_{\geq 0}$. All processing times are collectively denoted by a vector $\mathbf{p}$. A machine can only execute at most one job at any time. For a feasible schedule, each job is assigned to a machine non-preemptively. For some $i \in \mathcal{M}$, let C_i be the completion time of the i^{th} machine. Let $C_{\max}$ be the total completion time. Let $\mathbf{x} \in \{0, 1\}^{m \times n}$ be the assignment matrix that allocates jobs to machines. Formally, makespan schedules are modelled as an optimisation problem:

$$\begin{aligned} \min_{C_{\max}, \mathbf{C}, \mathbf{x}} \; & C_{\max} && \text{subject to:} \\ & \forall i \in \mathcal{M}.\; C_{\max} \geq C_i \\ & \forall i \in \mathcal{M}.\; C_i = \sum_{j \in \mathcal{J}} x_{i,j} \cdot p_j \\ & \forall j \in \mathcal{J}.\; \sum_{j \in \mathcal{J}} x_{i,j} = 1 \\ & \forall i \in \mathcal{M},\; \forall j \in \mathcal{J}.\; x_{i,j} \in \{0, 1\} \end{aligned}$$

Definition 1. A schedule S is defined by its assignment matrix $\mathbf{x}$ such that $S = \mathbf{x}$. S will be used to reference a high-level representation of $\mathbf{x}$ but does not specify its formal representation unlike $\mathbf{x}$, which will be used in linear programming and algorithms with its precise definition.

Definition 2. A schedule S is optimal iff S achieves the minimal total completion time.

Definition 3. A machine $i \in \mathcal{M}$ is critical iff $C_i = C_{max}$.

Definition 4. A job $j \in \mathcal{J}$ is critical iff j is allocated to a critical machine $i \in \mathcal{M}$ and $x_{i,j} = 1$.

Definition 5. A schedule satisfies the single exchange property (SEP) iff for any critical machine and any machine $i, i' \in \mathcal{M}$ and for all critical jobs $j \in \mathcal{J}$, $C_i - C_{i'} \leq p_j$

Definition 6. A schedule satisfies the pairwise exchange property (PEP) iff for any critical job and any job $j, j' \in \mathcal{J}$, if $p_j > p_{j'}$, then $C_i + p_{j'} \leq C_{i'} + p_j$.

Definition 7. A schedule S is efficient iff S satisfies SEP and PEP. Note this definition of efficiency succinctly captures necessary optimally conditions, which differs from the property efficiency defined in the paper [4].

Theorem 1. Schedule efficiency is a necessary condition for optimality.

Makespan schedules are often represented using cascade charts. The charts shows graphically the difference in total completion time of schedules where the problem has the parameters $m = 4$ and $n = 13$.

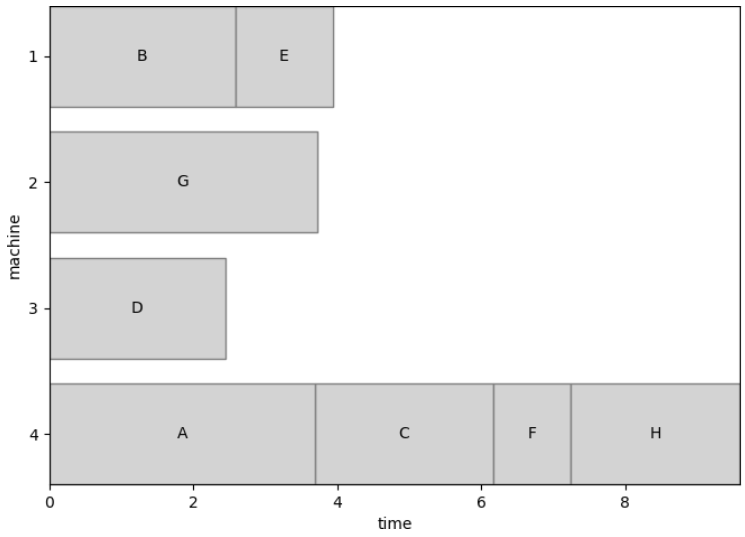


Figure 2.2: An inefficent schedule

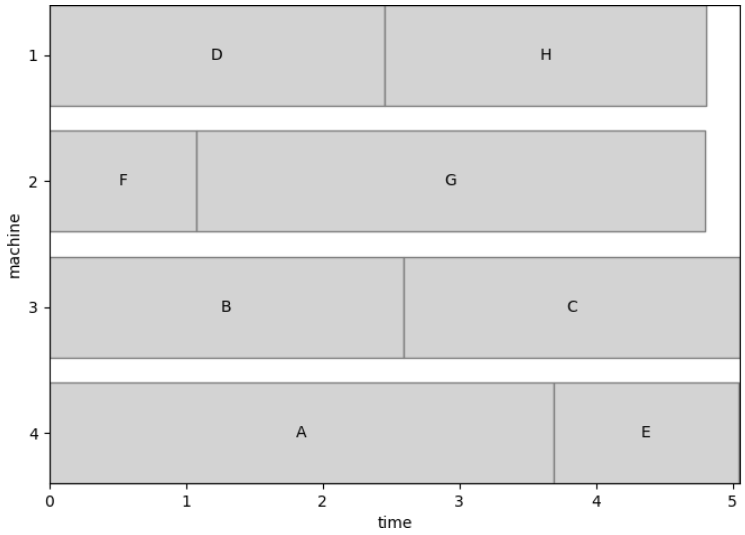


Figure 2.3: An efficient schedule

2.1.2 User Fixed Decisions

To accommodate practical applications of makespan schedules, user positive and negative fixed decisions are introduced as an extension to makespan problems. In a hospital setting, positive fixed decisions capture patients exclusively allocated to a nurse while negative fixed decisions capture unavailable or incompatible nurses and patients [4]. Let $D^-, D^+ \subseteq \mathcal{M} \times \mathcal{J}$ be the negative and positive fixed decisions respectively. Let D be the fixed decisions such that $D = (D^-, D^+)$.

Definition 8. A schedule S satisfies D iff $\forall (i,j) \in D^-.\ x_{i,j} = 0$ and $\forall \langle i,j \rangle \in D^+.\ x_{i,j} = 1$.

Definition 9. A fixed decision D is satisfiable iff there exists a schedule S such that S satisfies D.

A fixed decision D is satisfiable iff if the following necessary and sufficient conditions hold:

- D^+ and D^- are disjoint.
- $\forall \langle i,j \rangle, \langle i',j' \rangle \in D^+.\ i = i' \vee j \neq j'$
- $\forall j \in \mathcal{J}.\ \exists i \in \mathcal{M}.\ \langle i,j \rangle \notin D^-$

The relaxed definition of D allows D to be not satisfiable, which is not permitted in previous work [4]. This relaxation accommodates for poorly-formulated user problems, allowing explanations over validation of user input, which is more useful to users in a practical setting.

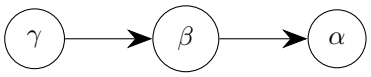
2.1.3 Interval Scheduling

Interval scheduling is a natural extension to makespan scheduling. This has freedom in determining the starting times of its jobs. In practice, rearranging critical jobs may effect the feasibility of a schedule. Interval scheduling is a widely researched area, with many literature proposing algorithms for variants of interval scheduling [7].

Interval scheduling is defined over $m \in \mathbb{N}$ and $n \in \mathbb{N}$ jobs where they are collectively denoted by the sets $\mathcal{M}$ and $\mathcal{J}$ such that $\mathcal{M} = \{1, ..., m\}$ and $\mathcal{J} = \{1, ..., n\}$ respectively. Each job $j \in \mathcal{J}$ must be allocated a machine $i \in \mathcal{M}$ preemptively. In addition, each job must be allocated between the interval $[s_j, f_j)$ such that $s_j < f_j$ where $s_j, f_j \in \mathbb{R}_{\geq 0}$ without loss of generality. Finally, each job is associated with a processing time $p_j \in \mathbb{R}_{\geq 0}$. No jobs are allowed to overlap. Note that $p_j > f_j - s_j$ is possible, meaning that interval schedules can be constructed to be infeasible.

2.2 Argumentation

Argumentation is a method to understand and evaluate reasons for and against potential conclusions. Argumentation is useful in resolving conflicts, clarifying incomplete information and most importantly, with respect to this project, explanations. The precise definition of an argument varies on the literature, however it is commonly agreed that arguments can attack or support other arguments. For an argument α to attack an argument β, α may critically challenge β such that acceptability of β is doubted [8]. This may be to question one of β's premises, by proposing a counter-example. For example, consider an scenario whether to sleep. Let α be "I want to sleep", β be "I have work to do" and γ be "I can work tomorrow". Using human intuition, we can derive that γ attacks β and β attacks α. This is represented graphically below.



If we conclude α to be acceptable, then we must not accept β. Accepting two arguments with conflicts can be interpreted as a contradiction or hypocritical. Hence, argumentation theory have measures of acceptable extensions, to decide whether some set of arguments are acceptable in some notion, with respect to different intuitions. This motivates to use abstract argumentation frameworks.

Note that this example uses implicit background knowledge, also known as enthymemes. In this scenario, one cannot sleep and work at the same time. To our advantage, argumentation is applied to well-defined scheduling problems, so enthymemes are inapplicable in this project.

2.2.1 Abstract Argumentation Frameworks

An abstract argumentation framework (AAF) models the relation of attacks between arguments [9]. Formally, an AAF is a directed graph $(Args, \rightsquigarrow)$ where $Args$ is the set of arguments and $\rightsquigarrow$ is a binary relation over $Args$. For $a, b \in$

$Args$, a attacks b iff $a \rightsquigarrow b$. Attacks are extended over sets of arguments, where $A \subseteq Args \rightsquigarrow b \in Args$ iff $\exists a \in A.\ a \rightsquigarrow b$. An extension E is a subset of $Args$.

Definition 10. An extension E is conflict-free iff $\forall a, b \in E.\ a \not\rightsquigarrow b$.

Definition 11. An extension E is stable iff E is conflict-free and $\forall a \in Args \backslash E.\ E \rightsquigarrow a$

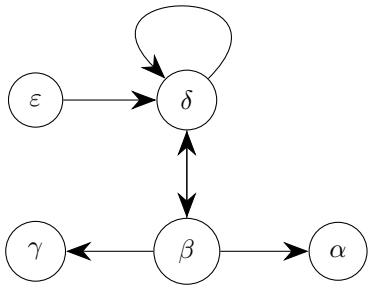


Figure 2.4: An AAF represented graphically

Consider the following example where $Args = \{\alpha, \beta, \gamma, \delta, \varepsilon\}$ and $\rightsquigarrow = \{\langle\beta, \alpha\rangle, \langle\beta, \gamma\rangle, \langle\beta, \delta\rangle, \langle\delta, \beta\rangle, \langle\delta, \delta\rangle, \langle\varepsilon, \delta\rangle\}$ as illustrated above. Then the following statements hold:

- $\varepsilon \rightsquigarrow \delta$.
- $\{\delta, \varepsilon\} \rightsquigarrow \beta$.
- $\{\alpha, \gamma\}$ is conflict-free but not stable.
- $\{\delta\}$ is not conflict-free and not stable.
- $\{\beta, \varepsilon\}$ is conflict-free and stable.

Definition 12. An extension E is admissible iff E is conflict-free and E attacks every argument attacking E.

Definition 13. An extension E defends an argument α iff for all every argument attacking α, E attacks such attacking argument

Definition 14. An extension E is complete iff E is admissible and E contains all arguments E defends.

Definition 15. An extension E is preferred iff E is maximally admissible with respect to $\subseteq$.

2.3 Explanations

Explanation generation problems are a class of explainable planning problems. Many generation tasks appeal to either minimality or simplicity [5]. Explanations may occur over observations. For example, a sequence of states may lead

to an error, an explanation can guide an agent to avoid such error. However, trustworthy and theoretical well-understood algorithms are difficult to explain to non-technical users [10]. The paper highlights concepts to explain given an optimisation context:

- Why did the optimiser do that?
- Why did the optimiser not do this?
- Why does this proposal result to more optimal result?
- Why can't this decision be taken?
- Why do I need to re-plan at this point?
- Will a better result be produced if given n more hours?

Arguably, understanding cannot be captured with a few questions. Future question include the negation, where their answers are not natively their negation.

2.4 Existing Tools

We will look at two scheduling software, Setmore and LEKIN, in terms of explainable planning. We will not look existing tools using argumentation because of relevancy, we use argumentation as a means of explanation in an intermediate process.

2.4.1 Setmore

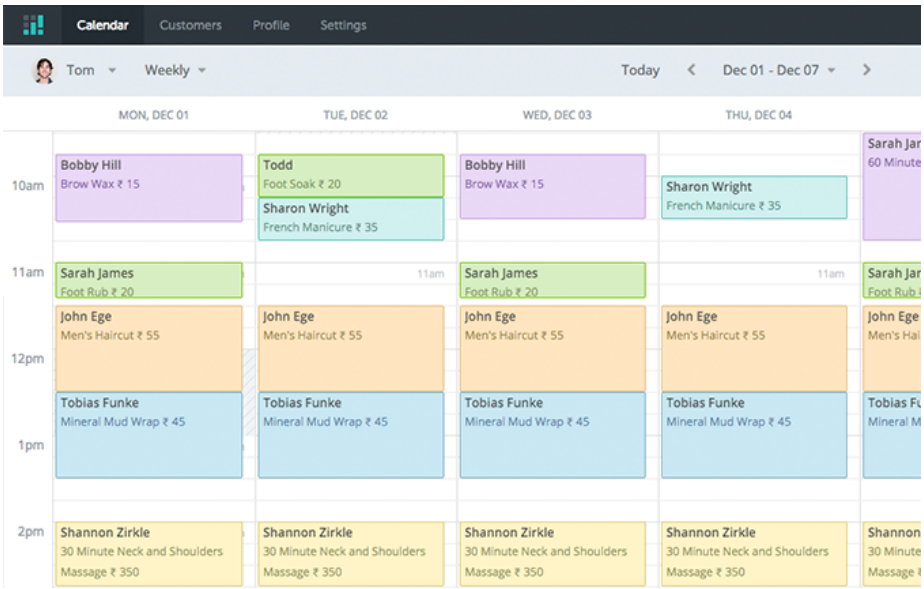


Figure 2.5: Setmore interactive interface [11]

Setmore is a commercial online application that records appointments, schedules and employees. The application is designed for small business such as in health-care [11], where managers can organise appointments on a calender. Makespan schedules are formulae where employees are machines and appointments are jobs. The intuitive interface enables users to quickly glance at appointments and their times, it is graphically clear when two appointments overlap. However, under the condition that one employee is able to attend at most one appointment, the user is presented with an error message. Messages also occur when resources are fully-booked or unavailable. Scheduling error messages alert the user during data input, which may prove problematic when a user wishes to input a large number of appointments. Appointments cannot be over-allocated because each appointment has at most assignment one by interface restrictions.

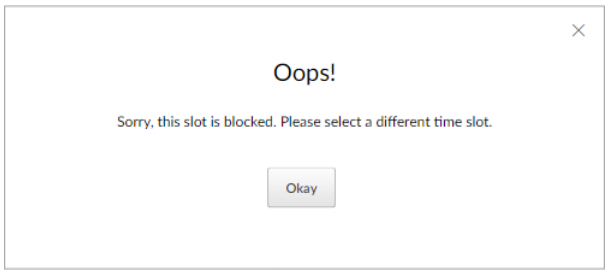


Figure 2.6: Overlapping appointment allocation error message

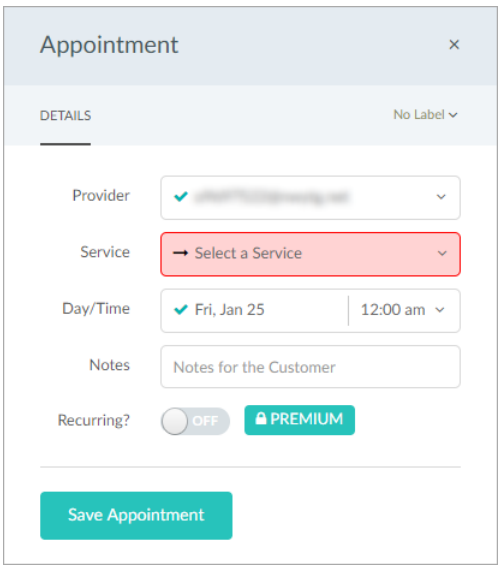


Figure 2.7: Data input of an appointment where there are no possible services, or possible assignments

We will assess the application under its free-trail, which may limit its explanative

functionality. A key observation is that there is no emphasis on the concept of an optimal schedule. Because the tool is designed for small businesses and optimality is not well-defined for arbitrary businesses, the user can graphically inspect and improve a schedule with respect to the user's notion of optimality. Modification of an existing schedule is well-facilitated within its interface, where appointments can be moved or swapped between employees. Explanations for unfeasible schedules are limited to data input verification and validation error messages.

2.4.2 LEKIN

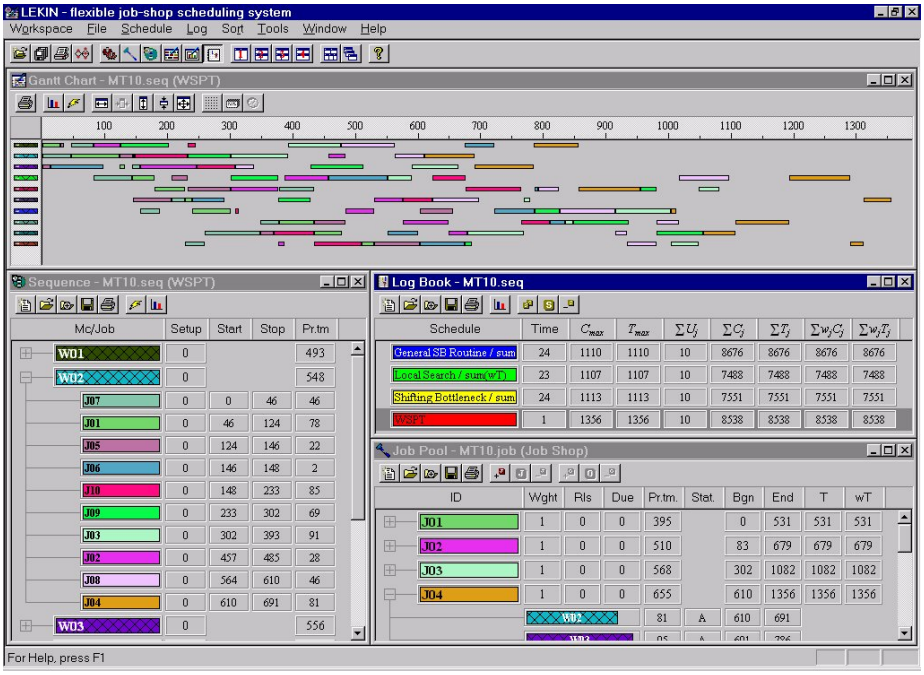


Figure 2.8: LEKIN interactive interface [12]

LEKIN is a academic-oriented scheduling application to teach students scheduling theory and its applications, developed at the Stem School of Business, NYU [12]. The application features numerous optimisation algorithms specialised for scheduling and draws inspiration from academic for rules and heuristics [3]. The tool supports single machines, parallel machines and flexible job shop settings.

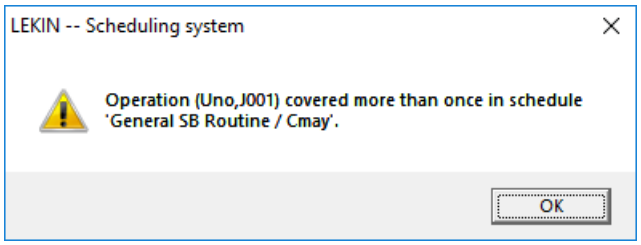


Figure 2.9: Over-allocation of a job to multiple machines

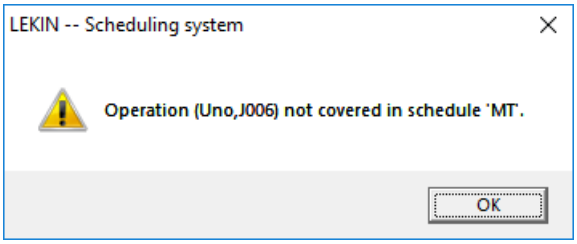


Figure 2.10: A job is not allocated to any machine

The application validates a schedule's feasibility at data input. Infeasible schedule results in error messages. The application computes optimal schedules, but has no functionality to verify its optimality. While our approach is to weaken optimality with efficiency, LEKIN takes the approach to compute common scheduling performance metrics such as makespan completion time, tardiness and weighted metrics over machines. The advantage is that these metrics can be easily and quickly be computed and are intuitive to non-technical users given some background reading. However, these metrics are global across all machines, and give no indication to improving schedules. Non-technical users may run one of the provided optimisers to improve metrics, but no explanation is offered between the assignments of the pre-optimised and the post-optimised schedules.

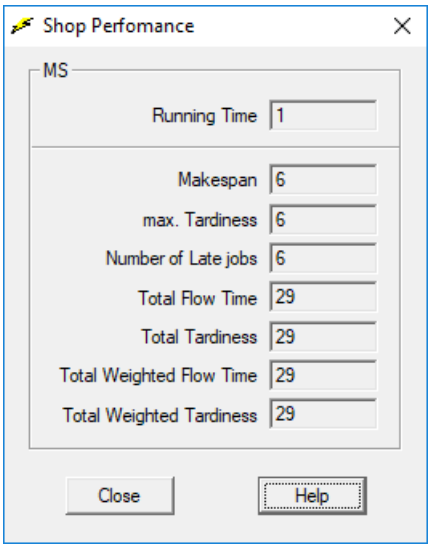


Figure 2.11: Performance metrics of an schedule

2.4.3 Comparison

It is clear that both application offers limited explanations, explicitly by error messages and implicitly by cascade charts. However, both methods require human intuition, which is unfeasible for large schedules. Therefore, for effective knowledge transfer, localised text explanations and cascade charts help users to focus on key machines and jobs.

Chapter 3

Implementation

3.1 Design Decisions

To select a programming language suitable for developing the tool, languages were compared with respect to possible challenges. The language should be compatible with popular optimisation solvers. To make efficient use of time, using an existing interface library between popular solvers is recommended. Interfaces are written for popular languages such as C++, Java and Python. Python was selected for its development speed and support for a wide range of libraries.

There is a balance between program speed and development time. A tool written in C may be fast but time-consuming. The purpose is to demonstrate the concept of argumentation with schedules while allowing analysis of potential future directions and short-comings. Hence, the tool should be sufficiently fast to be responsive, but not necessarily fast as possible.

There are many powerful and efficient solvers such as CPLEX [paper] and GLPK [paper]. To solver large problems, users use commercial over open-source solvers for their superior speed [paper]. However, users may not have access to a commercial solver. To accommodate users, Pyomo is used to interface to many popular solvers.

The tool features a GUI to aid its accessibility. Users such as hospital managers can often use a suitably-designed GUI without training of the tool. In practice, a GUI is easier to demonstrate than a CLI.

3.2 Structure

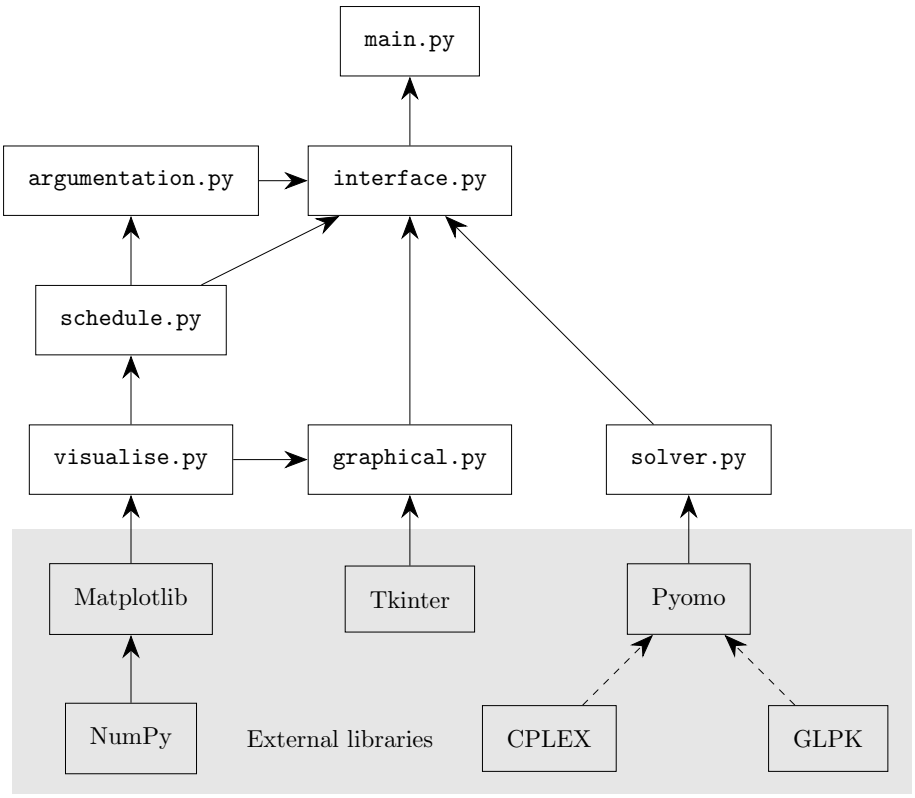


Figure 3.1: The graph illustrates the functional dependency between modules in the code-base of the tool. A solver is required for full functionality of the tool. This could be CPLEX or GLPK.

3.3 Algorithms

3.3.1 Notation

The algorithm uses operators over Boolean tensors to generate frameworks and compute stability. The definitions below share notions with linear algebra over matrices.

Definition 16. Let $\mathbf{0}^{d_1,\ldots,d_n}$ be the zero-valued tensor. The dimensions may be omitted if clear. Matrix subscripts are extended to n dimensions.
For example:

$$\mathbf{0}^{2\times 2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Definition 17. Let $\ominus$ be the element-wise logical negation operator over a tensor.

For example:

$$\ominus \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Definition 18. Let $\oslash$ be the element-wise logical and operator over tensors. For example:

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \oslash \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Definition 19. Let $\oslash$ be the element-wise logical or operator over tensors. For example:

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \oslash \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

3.3.2 Summary

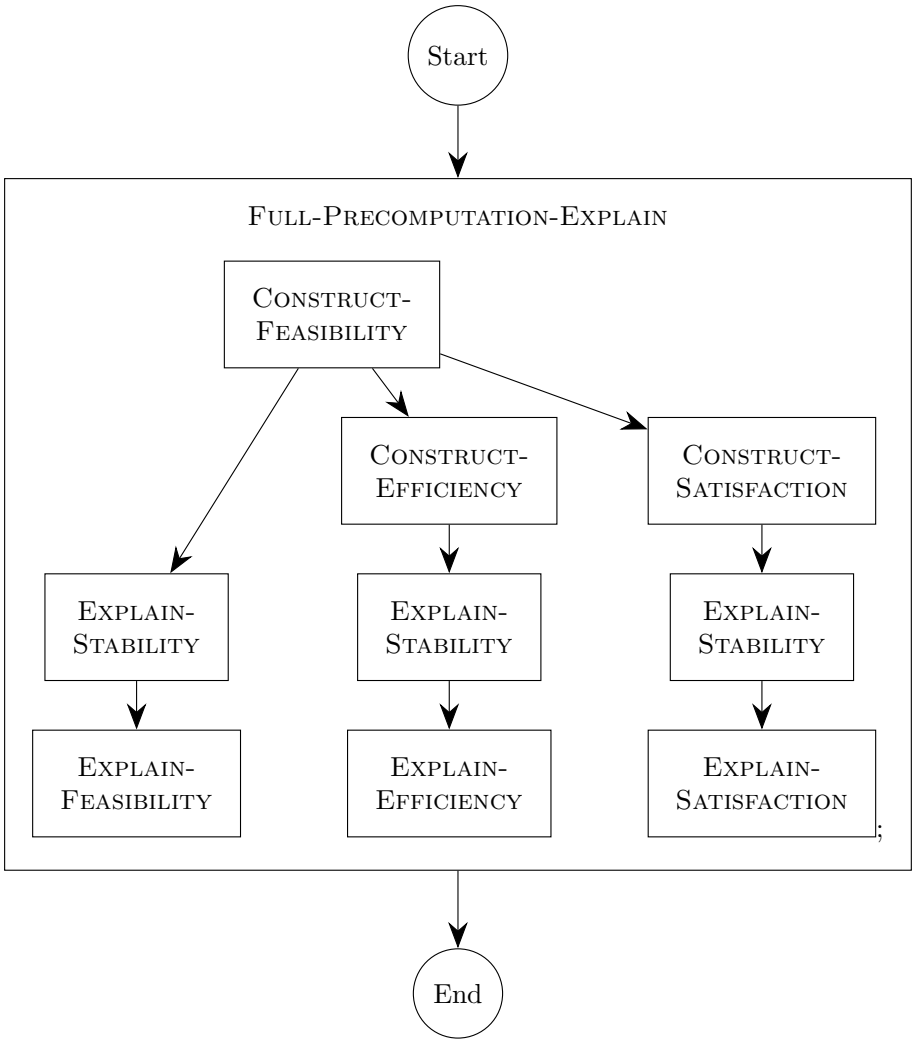


Figure 3.2: The graph summaries the required execution order of sub-functions in the Full-Precomputation-Explain algorithm. Nested rectangles denote nested function calls.

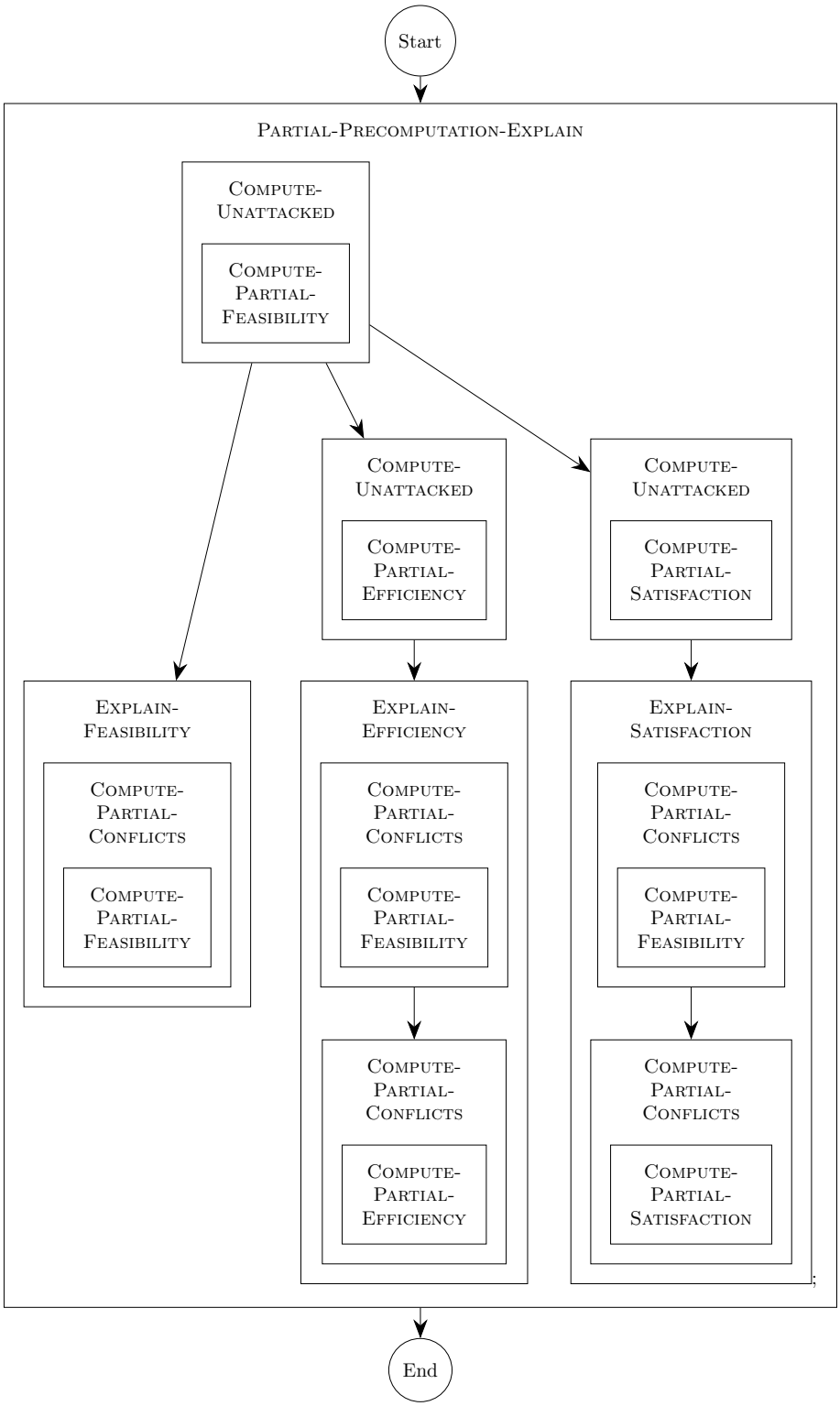


Figure 3.3: The graph summaries the required execution order of sub-functions in the Partial-Precomputation-Explain algorithm.

3.3.3 Framework Construction

The AAFs are constructed using the definitions in paper [4]. The definitions are reprinted for feasibility and fixed decisions only. Take arbitrary $i_1, i_2 \in \mathcal{M}$ and $j_1, j_2 \in \mathcal{J}$.

Definition 20. The feasibility framework $\rightsquigarrow_F$ is defined such that $\langle i_1, j_1\rangle \rightsquigarrow_F \langle i_2, j_2\rangle$ iff $i_1 \neq i_2 \wedge j_1 = j_2$.

Definition 21. The efficiency framework $\rightsquigarrow_S$ is defined such that $\langle i_1, j_1\rangle \rightsquigarrow_S \langle i_2, j_2\rangle \wedge \neg\text{SEP}(i_1, i_2, j_1) \vee \text{PEP}(i_1, i_2, j_1, j_2)$ where:

- Single exchange property (SEP): $\text{SEP}(i_1, i_2, j_1, D)$ iff $C_{i_1} = C_{\max} \wedge x_{i_1,j_1} = 1 \wedge C_{i_1} > C_{i_2} + p_{j_1} \wedge \langle i_1, j_1\rangle \notin D^+ \wedge \langle i_2, j_1\rangle \notin D^-$
- Pair-wise exchange property (PEP): $\text{PEP}(i_1, i_2, j_1, j_2, D)$ iff $C_{i_1} = C_{\max} \wedge x_{i_1,j_1} = 1 \wedge x_{i_2,j_2} = 1 \wedge i_1 \neq i_2 \wedge j_1 \neq j_2 \wedge p_{j_1} > p_{j_2} \wedge C_{i_1} + p_{j_2} > C_{i_2} + p_{j_1} \wedge \langle i_1, j_1\rangle \notin D^+ \wedge \langle i_2, j_2\rangle \notin D^+ \wedge \langle i_2, j_1\rangle \notin D^- \wedge \langle i_1, j_2\rangle \notin D^-$.

The paper [4] defines $\rightsquigarrow_S$ as an optimality framework. This report refers to $\rightsquigarrow_S$ as an efficiency framework, as its stability is determined by necessary but not sufficient conditions for optimality. In addition, the paper does considers efficiency and satisfaction to fixed decisions independently, we extend the notion of efficiency to respect these decisions. With the paper's naive definition of efficiency, the tool would recommend exchanges which violate fixed decisions.

Definition 22. The user fixed decision framework $\rightsquigarrow_D$ is defined such that $\langle i_1, j_1\rangle \rightsquigarrow_S \langle i_2, j_2\rangle \wedge \neg\text{DP}^+(i_1, i_2, j_1, j_2) \vee \text{DP}^-(i_1, i_2, j_1, j_2)$ where:

- Positive decision property: $\text{DP}^+(i_1, i_2, j_1, j_2)$ iff $\langle i_2, j_2\rangle \in D^+$
- Negative decision property: $\text{DP}^-(i_1, i_2, j_1, j_2)$ iff $\langle i_1, j_1\rangle \in D^- \wedge i_1 = i_2 \wedge j_1 = j_2$.

Algorithm 1

```
1:  function CONSTRUCT-FEASIBILITY(m, n)
2:      ⇝_F ← 0^{(m×n)^2}
3:      for j ∈ J do
4:          for i_1 ∈ M do
5:              for i_2 ∈ M do
6:                  if i_1 ≠ i_2 then
7:                      ⇝_{F i_1,j,i_2,j} ← 1
8:                  end if
9:              end for
10:         end for
11:     end for
12:     return ⇝_F
13: end function
```

$\rightsquigarrow_F$ can be constructed trivially in a dense data structure in $\mathcal{O}(m^2n^2)$ computational complexity, because of the complexity of zero-initialising $\rightsquigarrow_F$. This can

be constructed in $\mathcal{O}(m^2 n)$ complexity using a sparse data structure, but results in greatly more complicated code.

Algorithm 2

```
1: function CONSTRUCT-EFFICIENCY(m, n, p, x, D, ⇝_F)
2:     C ← x · p
3:     C_max ← max(C)
4:     ⇝_S ← ⇝_F
5:     for i_1 ∈ M do
6:         if i_1 = C_max then
7:             for j_1 ∈ J do
8:                 if x_{i_1,j_1} = 1 then
9:                     for i_2 ∈ M do
10:                        if SEP(i_1, j_1, i_2, D) then
11:                            ⇝_{S i_1,j_1,i_2,j_1} ← 0
12:                        end if
13:                        for j_2 ∈ J do
14:                            if PEP(i_1, j_1, i_2, j_2, D) then
15:                                ⇝_{S i_1,j_2,i_2,j_2} ← 1
16:                            end if
17:                        end for
18:                    end for
19:                end if
20:            end for
21:        end if
22:    end for
23:    return ⟨⇝_S, C⟩
24: end function
```

The construction of $\rightsquigarrow_S$ is expensive because of the explicit for-loops to iterate over the $\mathcal{M}^2\mathcal{J}^2$ space to compute the edges that satisfy PEP and to copy $\rightsquigarrow_F$. An optimisation by computing SEP outside of the j_2 loop, because PEP is invariant of j_2. We return the value of $\mathbf{C}$ because it will be used later, rather an recompute its value when necessary

Algorithm 3

function CONSTRUCT-SATISFACTION(m, n, D, $\rightsquigarrow_F$)
  $\rightsquigarrow_D \leftarrow \rightsquigarrow_F$
  for $\langle i, j \rangle \in D^-$ **do**
    $\rightsquigarrow_{S i,j,i,j} \leftarrow 1$
  end for
  for $\langle i_1, j_1 \rangle \in D^+$ **do**
    for $i_2 \in \mathcal{M}$ **do**
      for $j_2 \in \mathcal{J}$ **do**
        $\rightsquigarrow_{D i_2,j_2,i_1,j_1} \leftarrow 0$
      end for
    end for
  end for
  return $\rightsquigarrow_D$
end function

If D is assumed to be satisfiable, then D^+ has at most n decisions while D^- has at most $(m-1)n$ decisions. However, if D is not necessarily satisfiable to account for poorly-formulated user problems, so in general D^+ and D^- has at most mn decisions.

3.3.4 Stability

Stability can be computed by checking whether E exists within a all possible stable extensions of some $\langle Args, \rightsquigarrow \rangle$. However, a schedule cannot be reasoned on without understanding whether E may be stable on $\langle Args, \rightsquigarrow \rangle$. Existing solutions require a complication pipeline using answer set solvers. To make the implementation of the tool easier, we adapt the stability computation to schedules into a concise algorithm.

Algorithm 4

function EXPLAIN-STABILITY($\mathbf{x}$, $\rightsquigarrow$, $\bar{\mathbf{u}}$, $\bar{\mathbf{c}}$)
  $\mathbf{u} \leftarrow$ COMPUTE-UNATTACKED($\mathbf{x}$, $\rightsquigarrow$, $\bar{\mathbf{u}}$)
  for $i \in \mathcal{M}$ **do**
    for $j \in \mathcal{J}$ **do**
      $c_{i,j} \leftarrow$ COMPUTE-PARTIAL-CONFLICTS($\mathbf{x}$, $\rightsquigarrow_{i,j}$, $\bar{c}_{i,j}$)
    end for
  end for
  return $\langle \mathbf{u}, \mathbf{c} \rangle$
end function
function COMPUTE-UNATTACKED($\mathbf{x}$, $\rightsquigarrow$, $\bar{\mathbf{u}}$)
  $\mathbf{u} \leftarrow \ominus\, \mathbf{x}$
  for $i \in \mathcal{M}$ **do**
    for $j \in \mathcal{J}$ **do**
      if $x_{i,j} = 1$ **then**
        $\mathbf{u} \leftarrow \mathbf{u} \oslash \ominus \rightsquigarrow_{i,j}$
      end if
    end for
  end for
  $\mathbf{u} \leftarrow \mathbf{u} \oslash \ominus\, \bar{\mathbf{u}}$
  return u
end function
function COMPUTE-PARTIAL-CONFLICTS($\mathbf{x}$, $\rightsquigarrow_{i,j}$, $\bar{c}_{i,j}$)
  $c_{i,j} \leftarrow \mathbf{0}^{m \times n}$
  if $x_{i,j} = 1$ **then**
    $c_{i,j} \leftarrow \mathbf{x} \oslash \rightsquigarrow_{i,j}$
  end if
  $c_{i,j} \leftarrow c_{i,j} \oslash \ominus\, \bar{c}_{i,j}$
  return $c_{i,j}$
end function

The function EXPLAIN-STABILITY returns two tensors, $\mathbf{u}$ encode the unattacked nodes and $\mathbf{c}$ encode the edges are not conflict-free. $\bar{\mathbf{u}}$ and $\bar{\mathbf{c}}$ represent node and edges to ignore from returned values respectively, which are useful in tailoring explanations to particular constraints. By default, $\bar{\mathbf{u}} = \mathbf{0}$ and $\bar{\mathbf{c}} = \mathbf{0}$. The function uses $\mathbf{x}$ rather than its equivalent representation E because $\mathbf{x}$ can be manipulated directly from an optimiser in its tensor form unlike E. This results in improved performance. In addition, it is assumed that $E \subseteq Args$ so $Args$ does not need to be a parameter.

The auxiliary functions COMPUTE-UNATTACKED and COMPUTE-PARTIAL-CONFLICTS are defined such that at most $\mathcal{O}(mn)$ memory is allocated. This will be discussed in the following subsections.

Theorem 2. COMPUTE-CONFLICTS is correct where $\mathbf{x} \approx E$ under S:

$$\text{COMPUTE-CONFLICTS}(\mathbf{x}, \rightsquigarrow, \mathbf{0}) = \mathbf{0} \Leftrightarrow E \text{ is conflict-free on } \langle Args, \rightsquigarrow \rangle$$

Theorem 3. EXPLAIN-STABILITY is correct where $\mathbf{x} \approx E$ under S:

$$\text{EXPLAIN-STABILITY}(\mathbf{x}, \rightsquigarrow, \mathbf{0}, \mathbf{0}) = \langle \mathbf{0}, \mathbf{0} \rangle \Leftrightarrow E \text{ is stable on } \langle Args, \rightsquigarrow \rangle$$

3.3.5 Explanation

Explanations are given in italics. Implementation of algorithms use Python's `print()` to collect explanations over all algorithms in the tool's output.

Algorithm 5

function EXPLAIN-FEASIBILITY($\mathbf{u}$, $\mathbf{c}$)
  if $m = 0$ **then**
    if $n = 0$ **then**
      There are no jobs, so the schedule is trivially feasible.
    else
      There are no machines to allocate to jobs.
    end if
  else
    $\mathbf{y} \leftarrow \mathbf{0}^n$
    $\mathbf{z} \leftarrow \mathbf{0}^{n \times m}$
    for $j \in \mathcal{J}$ **do**
      for $i_1 \in \mathcal{M}$ **do**
        for $i_2 \in \mathcal{M}$ **do**
          if $c_{i_1,j,i_2,j} = 1$ **then**
            $y_j \leftarrow 1$
            $z_{j,i_1} \leftarrow 1$
            $z_{j,j_2} \leftarrow 1$
          end if
        end for
      end for
    end for
    if $u_0^T = \mathbf{0} \wedge \mathbf{y} = \mathbf{0}$ **then**
      All jobs are allocated by exactly one machine.
    else
      for $j \in \mathcal{J}$ **do**
        if $u_{0,j} = 1$ **then**
          Job j is not allocated by any machine.
        end if
        if $y_j \neq \mathbf{0}$ **then**
          Job j is over-allocated by machines $\{i \mid i \in \mathcal{M}, z_{j,i} = 1\}$.
        end if
      end for
    end if
  end if
end function

The paper [4] does not state explanations for trivial cases when $m = 0$ or $n = 0$. The above algorithm handles these cases with additional explanations. A problem with the naive implementation of generating an explanation for each conflict in $\mathbf{c_F}$ results in k^2 explanations for k conflicting machines for a job. This results in superfluous text for the user. To summarise these explanations, the algorithm constructs a pseudo-schedule $\mathbf{z}$ which can be interpreted as $\mathbf{x}$ transposed and rows filtered if $\sum_{i \in \mathcal{M}} x_{i,j} > 1$ for all jobs j. Afterwards, the algorithm prints the non-zero indices of $\mathbf{c}'$, which refer to the machines that

causes over-allocation.
The algorithm features two optimisations. The variable $\mathbf{y}$ represent over-allocated jobs. $y_j = \mathbf{0}$ is faster to compute than its equivalent $z_j = \mathbf{0}$ because y_j is an scalar aggregate over conflicting machines, unlike the vector z_j. Likewise, by the construction of $\mathbf{u_F}$, we can exploit $u_0^T = \mathbf{0} \iff \mathbf{u} = \mathbf{0}$ because $u_0^T = \frac{1}{m}\mathbf{u}^T \cdot \mathbf{1}^m$.

Algorithm 6

```
1: function EXPLAIN-EFFICIENCY(p, C, u, c)
2:     i_1 ← first argmax of C
3:     reasons ← empty list
4:     for j_1 ∈ J do
5:         for i_2 ∈ M do
6:             if u_{i_2,j_1} = 1 then
7:                 reason ← Job j_1 can be allocated to machine i_2.
8:                 append reason to reasons
9:             end if
10:            for j_2 ∈ J do
11:                if c_{i_1,j_1,i_2,j_2} = 1 then
12:                    reason ← Job j_1 and j_2 can be swapped with machines i_1
   and i_2.
13:                    append reason to reasons
14:                end if
15:            end for
16:        end for
17:    end for
18:    sort reasons by ⟨reduction, processing time⟩
19:    if reasons is empty then
20:        All jobs satisfy single and pairwise exchange properties.
21:    else
22:        output reasons
23:    end if
24: end function
```

The algorithm has $\mathcal{O}(mn^2 \log(mn))$ computational complexity, arising from sorting reasons generated. Explanation of efficiency results in at most m^2n^2 lines, which grows quickly for large schedules. To make this easier for the user to understand, we sort the explanations by its reduction, the amount the total completion time will reduce when an single or pairwise exchange occurs. This will highlight the most significant improvements for the user. This justifies the increased complexity with the logarithmic factor.

A key limitation with sorting by reduction, is in the cases of multiple critical machines. In this case, all reductions are zero. This is because single or pairwise exchange results in local optimisations of the same objective value. To find a strictly more optimal schedule, we need to look k steps ahead, where k is the number of critical machines. To solve this, the tool will need to generate instructions of k actions, of single and pair-wise exchanges. For an arbitrary large schedule, this will cause an exponential explosion in k of the explanation length. We continue the assumption that exponential tractable complexity is

not feasible, so therefore, a full explanation for efficiency is not feasible.

An alternative solution is to restrict the explanation space by giving local explanations. Hence in the algorithm, we consider only one critical machine. This reduces the computational complexity by a factor of m, which is significant because efficiency is the most expensive schedule property to explain.

Algorithm 7

```
1: function EXPLAIN-SATISFACTION(D, u, c)
2:     for j ∈ J do
3:         if ∃i ∈ M. ⟨i, j⟩ ∉ D⁻ then
4:             Job j cannot be allocated to any machine.
5:         end if
6:         if D⁻ and D⁺ are not disjoint then
7:             Job j subject to conflicting negative and positive fixed decisions.
8:         end if
9:         if |i ∈ M, ⟨i, j⟩ ∈ D⁺| > 1 then
10:            Job j cannot be allocated to multiple machines.
11:        end if
12:    end for
13:    y ← 0^{m×n}
14:    for i ∈ M do
15:        for j ∈ J do
16:            y ← y ⩒ c_{i,j}
17:        end for
18:    end for
19:    if u = 0 ∧ y = 0 then
20:        All jobs satisfy user fixed decisions.
21:    else
22:        for i ∈ M do
23:            for j ∈ J do
24:                if u_{i,j} then
25:                    Job j must be allocated to machine i.
26:                end if
27:                if y_{i,j} then
28:                    Job j must not be allocated to machine i.
29:                end if
30:            end for
31:        end for
32:    end if
33: end function
```

The variable $\mathbf{y}$ refers to allocations not satisfying D^+. Because of the relaxation that D is not assumed to be satisfiable, we must check the sufficient conditions for this, and generate their explanations if necessary.

Algorithm 8

function FULL-PRECOMPUTATION-EXPLAIN(m, n, $\mathbf{p}$, D, $\mathbf{x}$)
  $\rightsquigarrow_F \leftarrow$ CONSTRUCT-FEASIBILITY(m, n)
  $\langle \mathbf{u_F}, \mathbf{c_F} \rangle \leftarrow$ EXPLAIN-STABILITY($\mathbf{x}$, $\rightsquigarrow_F$, $\mathbf{0}$, $\mathbf{0}$)
  EXPLAIN-FEASIBILITY($\mathbf{u_F}$, $\mathbf{c_F}$)
  $\langle \rightsquigarrow_S, \mathbf{C} \rangle \leftarrow$ CONSTRUCT-EFFICIENCY(m, n, $\mathbf{p}$, $\mathbf{x}$, D, $\rightsquigarrow_F$)
  $\langle \mathbf{u_S}, \mathbf{c_S} \rangle \leftarrow$ EXPLAIN-STABILITY($\mathbf{x}$, $\rightsquigarrow_S$, $\mathbf{u_F}$, $\mathbf{c_F}$)
  EXPLAIN-EFFICIENCY($\mathbf{p}$, $\mathbf{C}$, $\mathbf{u_S}$, $\mathbf{c_S}$)
  $\rightsquigarrow_D \leftarrow$ CONSTRUCT-SATISFACTION(m, n, $\mathbf{x}$, $\rightsquigarrow_F$)
  $\langle \mathbf{u_D}, \mathbf{c_D} \rangle \leftarrow$ EXPLAIN-STABILITY($\mathbf{x}$, $\rightsquigarrow_D$, $\mathbf{u_F}$, $\mathbf{c_F}$)
  EXPLAIN-SATISFACTION($\mathbf{u_D}$, $\mathbf{c_D}$)
end function

The above high-level algorithm generates explanations for feasibility, efficiency and satisfaction while summarising the interaction of construction, stability and explanation functions. The function is named with full precomputation because all frameworks are fully constructed before explanations.

3.3.6 Memory Limitations

A full framework requires at least m^2n^2 bytes space in memory. For, $m = n = 256$ this requires 4GiB. One solution is not to construct frameworks and compute their stability in sequence, but rather inline partial framework construction into frameworks. This reduces the memory complexity to $\mathcal{O}(mn)$, while keeping the same computational complexity. This obviously will be slower to compute, but this method is more scalable. Therefore, we need to modify any function requiring a data object of size m^2n^2 such as $\mathbf{c}$ and $\mathbf{x}$.

The framework construction functions are trivially modified to compute a subgraph from a node, given its indices. For example, CONSTRUCT-FEASIBILITY$(i, j) = \rightsquigarrow_{F_{i,j}}$

Algorithm 9

function PARTIAL-PRECOMPUTATION-EXPLAIN(m, n, $\mathbf{p}$, D, $\mathbf{x}$)
  function $\rightsquigarrow'_F(i, j)$
    return CONSTRUCT-PARTIAL-FEASIBILITY(m, n, i, j)
  end function
  function $\mathbf{c'_F}(i, j)$
    return CONSTRUCT-PARTIAL-CONFLICTS($\mathbf{x}$, $\rightsquigarrow'_F$, $\mathbf{0}$)
  end function
  function $\rightsquigarrow'_S(i, j)$
    return CONSTRUCT-PARTIAL-EFFICIENCY(m, n, $\mathbf{p}$, $\mathbf{x}$, D, i, j)
  end function
  function $\mathbf{c'_S}(i, j)$
    return CONSTRUCT-PARTIAL-CONFLICTS($\mathbf{x}$, $\rightsquigarrow'_S$, $\rightsquigarrow'_F$)
  end function
  function $\rightsquigarrow'_D(i, j)$
    return CONSTRUCT-PARTIAL-SATISFACTION(m, n, D, i, j)
  end function
  function $\mathbf{c'_D}(i, j)$
    return CONSTRUCT-PARTIAL-CONFLICTS($\mathbf{x}$, $\rightsquigarrow'_D$, $\rightsquigarrow'_F$)
  end function
  $\mathbf{u_F} \leftarrow$ COMPUTED-UNATTACKED($\mathbf{x}$, $\rightsquigarrow'_F$, $\mathbf{0}$)
  EXPLAIN-FEASIBILITY($\mathbf{u_F}$, $\mathbf{c'_F}$)
  $\mathbf{u_S} \leftarrow$ COMPUTED-UNATTACKED($\mathbf{x}$, $\rightsquigarrow'_S$, $\mathbf{u_F}$)
  EXPLAIN-EFFICIENCY($\mathbf{u_S}$, $\mathbf{c'_S}$)
  $\mathbf{u_D} \leftarrow$ COMPUTED-UNATTACKED($\mathbf{x}$, $\rightsquigarrow'_D$, $\mathbf{u_F}$)
  EXPLAIN-SATISFACTION($\mathbf{u_D}$, $\mathbf{c'_D}$)
end function

Chapter 4

Schedule Properties and Argumentation

Schedule properties such as efficiency are modelled using frameworks to explain the satisfaction of properties. In particular, the definitions of efficiency and fixed decision frameworks extend from the definition of the feasibility framework. This is generalised to reason about arbitrary number of properties, using a commonly-extended framework. Using this extended reasoning, we apply argumentation to a variant interval scheduling to illustrate extensions of argumentation with scheduling.

4.1 Extensions

We use stability over other notions of good extensions to accurately model schedule constraints.

4.2 Frameworks

In order to reason about arbitrary number of properties, we inductively grow the property space over a commonly-extended framework, denoted by $\rightsquigarrow_0$. A property P is modelled by the framework $\rightsquigarrow_P$, such that $\rightsquigarrow_P = \rightsquigarrow_0 \cup \rightsquigarrow_P^+$. In other words, P may require adding edges in $\rightsquigarrow_0$ for $\rightsquigarrow_P$ to be modelled correctly. To be correct, we must preserve the stability of some extension E on $\langle Args, \rightsquigarrow_P \rangle$ if E is also stable on $\langle Args, \rightsquigarrow_0 \rangle$ and $P(S)$ is true. Let $\rightsquigarrow_1 \in Args^2$ and $\rightsquigarrow_2 \in Args^2$ be arbitrary frameworks.

Definition 23. A framework $\rightsquigarrow$ fully-models a schedule property P iff for all extensions E and corresponding schedules S, E is stable on $\langle Args, \rightsquigarrow \rangle \Leftrightarrow P(S)$

Definition 24. A framework $\rightsquigarrow$ models a schedule property P iff for all extensions E and corresponding schedules S, E is conflict-free on $\langle Args, \rightsquigarrow \rangle \Leftrightarrow P(S)$

Lemma 1. E is conflict-free on $\langle Args, \rightsquigarrow_1 \rangle$ and on $\langle Args, \rightsquigarrow_2 \rangle$ iff E is conflict-free on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$.

Proof. To prove the forward implication, assume E is conflict-free on $\langle Args, \rightsquigarrow_1 \rangle$ and on $\langle Args, \rightsquigarrow_2 \rangle$. To aim for a contradiction, assume E is not conflict-free on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$. Then there exists $e_1, e_2 \in E$ such that $e_1(\rightsquigarrow_1 \cup \rightsquigarrow_2)e_2$. Then $e_1 \rightsquigarrow_1 e_2$ or $e_1 \rightsquigarrow_2 e_2$. Both cases lead to a contradiction, so E is conflict-free on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$.
To prove the backward implication, assume E is conflict-free on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$. To aim for a contradiction, assume E is not conflict-free on $\langle Args, \rightsquigarrow_1 \rangle$. Then there exists $e_1, e_2 \in E$ such that $e_1 \rightsquigarrow_1 e_2$. Then $e_1(\rightsquigarrow_1 \cup \rightsquigarrow_2)e_2$, which contradicts the most recent assumption. Therefore, E is conflict-free on $\langle Args, \rightsquigarrow_1 \rangle$, and also conflict-free on $\langle Args, \rightsquigarrow_2 \rangle$ by similar argument. □

Lemma 2. If E is stable on $\rightsquigarrow_1$ and E is conflict-free on $\rightsquigarrow_2$, then E is stable on $(\rightsquigarrow_1 \cup \rightsquigarrow_2)$.

Proof. Assume E is stable on $\rightsquigarrow_1$ and E is conflict-free on $\rightsquigarrow_2$. By definition of stability, $\forall a \in Args \setminus E \; \exists e \in E \; e \rightsquigarrow_1 a$. Then $\forall a \in Args \setminus E \; \exists e \in E \; e(\rightsquigarrow_1 \cup \rightsquigarrow_2)a$. So every argument not in E is attacked by some argument in E. E is conflict-free on $\rightsquigarrow_1$ because E is stable on $\rightsquigarrow_1$. Since E is conflict-free on $\rightsquigarrow_1$ and on $\rightsquigarrow_2$, we use Lemma 1 to show that E is also conflict-free on $(\rightsquigarrow_1 \cup \rightsquigarrow_2)$. Therefore E is stable on $\rightsquigarrow_1 \cup \rightsquigarrow_2)$. □

Lemma 3. If E is stable on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$, then E is conflict-free on $\langle Args, \rightsquigarrow_1 \rangle$.

Proof. Assume E is stable on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$. By definition of stability, E is conflict-free on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$. By Lemma 1, E is conflict-free on $\langle Args, \rightsquigarrow_1 \rangle$. □

Lemma 4. If E is stable on $\langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle$ and $\forall a \in Args \setminus E \; (\exists e \in E \; e \rightsquigarrow_2 a) \implies (\exists e \in E \; e \rightsquigarrow_1 a)$, then E is stable on $\langle Args, \rightsquigarrow_1 \rangle$.

Proof.

$$
\begin{aligned}
& E \text{ is stable on } \langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle \\
& \quad \wedge \forall a \in Args \setminus E \; (\exists e \in E \; e \rightsquigarrow_2 a) \implies (\exists e \in E \; e \rightsquigarrow_1 a) \\
\implies & E \text{ is conflict-free on } \langle Args, \rightsquigarrow_1 \cup \rightsquigarrow_2 \rangle \\
& \quad \wedge \forall a \in Args \setminus E \; \exists e \in E \; e(\rightsquigarrow_1 \cup \rightsquigarrow_2) \\
& \quad \wedge \forall a \in Args \setminus E \; (\exists e \in E \; e \rightsquigarrow_2 a) \implies (\exists e \in E \; e \rightsquigarrow_1 a) \\
\implies & E \text{ is conflict-free on } \langle Args, \rightsquigarrow_1 \rangle \\
& \quad \wedge \forall a \in Args \setminus E \; \exists e \in E \; (e \rightsquigarrow_1 a \vee e \rightsquigarrow_2 a) \\
& \quad \wedge \forall a \in Args \setminus E \; (\exists e \in E \; e \rightsquigarrow_2 a) \implies (\exists e \in E \; e \rightsquigarrow_1 a) \\
\implies & E \text{ is conflict-free on } \langle Args, \rightsquigarrow_1 \rangle \\
& \quad \wedge \forall a \in Args \setminus E \; ((\exists e \in E \; e \rightsquigarrow_1 a) \vee (\exists e \in E \; e \rightsquigarrow_2 a)) \\
& \quad \wedge \forall a \in Args \setminus E \; (\exists e \in E \; e \rightsquigarrow_2 a) \implies (\exists e \in E \; e \rightsquigarrow_1 a) \\
\implies & E \text{ is conflict-free on } \langle Args, \rightsquigarrow_1 \rangle \\
& \quad \wedge \forall a \in Args \setminus E \; ((\exists e \in E \; e \rightsquigarrow_1 a) \vee (\exists e \in E \; e \rightsquigarrow_1 a)) \\
\implies & E \text{ is conflict-free on } \langle Args, \rightsquigarrow_1 \rangle \\
& \quad \wedge \forall a \in Args \setminus E \; \exists e \in E \; e \rightsquigarrow_1 a \\
\implies & E \text{ is stable on } \langle Args, \rightsquigarrow_1 \rangle
\end{aligned}
$$

□

Theorem 4. Let $P_0, ..., P_K$ be schedule properties with K properties. If $\rightsquigarrow_0$ fully-models P_0, and $\forall k \in [\![1, K]\!]$ $\rightsquigarrow_k$ models P_k, and for all extensions E, $\forall a \in Arg \setminus E\ \forall k \in [\![1, k]\!]\ \big((\exists e \in E\ e \rightsquigarrow_k a) \implies (\exists e \in E\ e \rightsquigarrow_0 a)\big)$, then $\left(\bigcup_{k=0}^{K} \rightsquigarrow_k\right)$ fully-models $P_{[\![0,K]\!]}$. $P_{[\![i,j]\!]}$ is defined as an aggregate schedule property where for all schedules S, $P_{[\![i,k]\!]}(S) \Leftrightarrow \forall k \in [\![i, j]\!]\ P_k(S)$.

Proof. Take arbitrary $K \in \mathbb{N}$. To prove forward implication:

1. $\rightsquigarrow_0$ fully-models P_0 — given
2. $\forall k \in [\![1, K]\!]$ $\rightsquigarrow_k$ models P_k — given
3. $\forall a \in Arg \setminus E\ \forall k \in [\![1, K]\!]\ \big((\exists e \in E\ e \rightsquigarrow_k a) \implies (\exists e \in E\ e \rightsquigarrow_0 a)\big)$ — given
4. E is stable on $\langle Args, \bigcup_{k=0}^{K} \rightsquigarrow_k \rangle$ — assumption
5. $\forall a \in Arg \setminus E\ \left(\left(\exists e \in E\ e \left(\bigcup_{k=0}^{K} \rightsquigarrow_k\right) a\right) \implies (\exists e \in E\ e \rightsquigarrow_0 a)\right)$ — by equivalence of 3
6. E is stable on $\langle Args, \rightsquigarrow_0 \rangle$ — lemma 4, 4, 5
7. $P_0(S)$ — 1, 6
8. Take arbitrary $k \in [\![1, K]\!]$
    9. E is conflict-free on $\langle Args, \rightsquigarrow_k \rangle$ — lemma 3, 4
    10. P_k(S) — 2, 9
11. $\forall k \in [\![1, K]\!]$ $P_k(S)$ — 8, 10
12. $P_{[\![0,K]\!]}(S)$ — 7, 11

To prove backward implication:

1. $\rightsquigarrow_0$ fully-models P_0 — given
2. $\forall k \in [\![1, K]\!]$ $\rightsquigarrow_k$ models P_k — given
3. $P_{[\![0,K]\!]}(S)$ — assumption
4. $P_0(S)$ — 3
5. E is stable on $\langle Args, \rightsquigarrow_0 \rangle$ — 1, 4
6. Recursively over $k \in [\![1, K]\!]$
    7. $P_k(S)$ — 3
    8. E is conflict-free on $\langle Arg, \rightsquigarrow_k \rangle$ — 2, 7
    9. E is stable on $\langle Arg, \bigcup_{k'=0}^{k} \rightsquigarrow_{k'} \rangle$ — lemma 2, 5, 8
10. E is stable on $\langle Arg, \bigcup_{k=0}^{K} \rightsquigarrow_k \rangle$ — 6, 9

□

4.3 Interval Scheduling

Makespan schedules are extended to discrete time-indexed interval scheduling. Let T be the exclusive upper-bound of indexed time where $\mathcal{T} = \{0, ..., T-1\}$. The assignment matrix $\mathbf{x} \in \mathcal{M} \times \mathcal{J} \times \mathcal{T}$ is extended such that $x_{i,j,t} = 1$ iff job j is starts work on machine i at time t. Each job j has a start time $s_j \in \mathcal{T}$ and finish time $f_j \in \{0, ..., T\}$, where j must be completed within the $[s_j, f_j)$ interval. The objective is to minimise the total completion time.

$$\min_{\mathbf{x}} C_{\max} \text{ subject to:}$$

$$\forall i \in \mathcal{M}\ \forall j \in \mathcal{J}\ \forall t \in \mathcal{T}\ C_{\max} \geq x_{i,j,t}(t + p_j)$$

$$\forall j \in \mathcal{J}\ \sum_{i \in \mathcal{M}} \sum_{t \in \mathcal{T}} x_{i,j,t} = 1 \qquad (\alpha)$$

$$\forall i \in \mathcal{M}\ \forall t \in \mathcal{T}\ \sum_{j \in \mathcal{J}} \sum_{t' = \max\{t - p_j + 1, 0\}}^{t} x_{i,j,t'} \leq 1 \qquad (\beta)$$

$$\forall i \in \mathcal{M}\ \forall j \in \mathcal{J}\ \forall t \in \{0, ..., s_j - 1\}\ x_{i,j,t} = 0 \qquad (\gamma)$$

$$\forall i \in \mathcal{M}\ \forall j \in \mathcal{J}\ \forall t \in \{f_j - p_j + 1, ..., T - 1\}\ x_{i,j,t} = 0 \qquad (\delta)$$

$$\forall \langle i, j \rangle \in D^-\ \forall t \in \mathcal{T}\ x_{i,j,t} = 0 \qquad (\varepsilon)$$

(α) models feasibility, that all jobs must be allocated. (β) models that machines cannot process multiple jobs at the same time. (γ) and (δ) model the restriction of start and end times respectively. (ε) model negative fixed decisions. Positive fixed decisions are not modelled because for each D^+, there exists an D^- that equivalently restricts the allocation space of $\mathbf{x}$. For instance, $\forall \langle i, j \rangle \in D^+\ \forall i' \in \mathcal{M}\ \forall j' \in \mathcal{J}\ i \neq i' \wedge j \neq j' \implies \langle i', j' \rangle \in D^-$ captures all satisfiable instances of D.

Chapter 5

Evaluation

5.1 Measures of Success

The success of the project can be measured by comparing the project results with objectives. We will also review the design choices in constructing the tool. The review will highlight the practical outcomes from using argumentation to explain scheduling.

Arguably, the most important outcome of this project is that the tool is functionality correct. This means the tool is required to explain schedules for feasibility, efficiency and satisfaction with respect to user decisions.

One objective is to implement an accessible tool. To measure accessibility, we can refer to the tractability complexity from explanations. The length of explanations either in the number of words or characters may be correlated with understandability. Alternatively, we can conduct a survey targeted towards potential users. Because the understandability of explanations are difficult to measure, we will attempt to quantity this using multiple choice questions. To carefully evaluate explanations, one would need to refer to cognitive science, which is beyond the scope of this project. In addition, we can measure performance to reflect responsiveness and scalability of the tool. This may be achieved by using profiling utilities to measure performance metrics such as time to generate explanations and memory consumption.

We can also use the survey to ask how well our tool performs with respect to applicability and knowledge transfer. We can also infer the practical limitations from the tool to measure the applicability.

5.2 Theoretical Results

Algorithm	Computational	Memory	Tractability
Construct-Feasibility	$\mathcal{O}(m^2n^2)$	$\mathcal{O}(m^2n^2)$	
Construct-Efficiency	$\mathcal{O}(m^2n^2)$	$\mathcal{O}(m^2n^2)$	
Construct-Satisfaction	$\mathcal{O}(m^2n)$	$\mathcal{O}(m^2n^2)$	
Compute-Unattacked	$\mathcal{O}(m^2n^2)$	$\mathcal{O}(mn)$	
Compute-Partial-Conflicts	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	
Explain-Stability	$\mathcal{O}(m^2n^2)$	$\mathcal{O}(m^2n^2)$	
Explain-Feasibility	$\mathcal{O}(mn^2)$	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$
Explain-Efficiency	$\mathcal{O}(mn^2 \log(mn^2))$	$\mathcal{O}(mn^2)$	$\mathcal{O}(mn^2)$
Explain-Satisfaction	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$
Full-Precomputation-Explain	$\mathcal{O}(m^2n^2 \log(mn^2))$	$\mathcal{O}(m^2n^2)$	$\mathcal{O}(mn^2)$
Partial-Precomputation-Explain	$\mathcal{O}(m^2n^2 \log(mn^2))$	$\mathcal{O}(mn^2)$	$\mathcal{O}(mn^2)$

Figure 5.1: Computational, memory and tractability complexity of algorithms using argumentation

Algorithm	Computational	Memory	Tractability
Naive-Explain-Feasibility	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$
Naive-Explain-Efficiency	$\mathcal{O}(mn^2 \log(mn^2))$	$\mathcal{O}(mn^2)$	$\mathcal{O}(mn^2)$
Naive-Explain-Satisfaction	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$	$\mathcal{O}(mn)$
Naive-Explain	$\mathcal{O}(mn^2 \log(mn^2))$	$\mathcal{O}(mn^2)$	$\mathcal{O}(mn^2)$

Figure 5.2: Computational, memory and tractability complexity of algorithms without using argumentation

Using an naive explanation approach only improves the computational complexity of verifying the feasibility property.

5.3 Practical Results

I will be able to demonstrate the tool, and its functionality. This will be reflected in the report through a user documentation guide.

5.3.1 Full versus Partial Precomputation

We will compare two algorithmic approaches to AAF construction with the naive approach without argumentation. The algorithms are implemented, then profiled for elapsed time and maximum allocated memory. The time and memory are measured with the Python's `cProfile` and `memory-profiler` modules respectively. The tool was executed on Department of Computing's virtual machines, with the specification of dual-core CPU at 2GHz with 2GiB RAM.

Time comparison:

- Elapsed time measurements are noisy.
- For less than 100 jobs, all approaches have approximately equal timings.

- From profiling, the tool takes 0.4 seconds on average to startup.
- Partial precomputation is 3% faster than full precomputation on average, excluding startup time.
- Naive is 18% faster than partial precomputation on average, excluding startup time.
- Both graphs hints at quadratic complexity.

Memory comparison:

- For less than 40 jobs, all approaches have approximately equal memory usage.
- From profiling, the tool uses 52MiB on average to startup.
- For a large number of jobs, partial-precomputation is 7 times more efficient than full precomputation excluding startup memory.
- Naive and partial-precomputation have less than 1% memory usage difference.
- Both graphs hints at quadratic complexity.

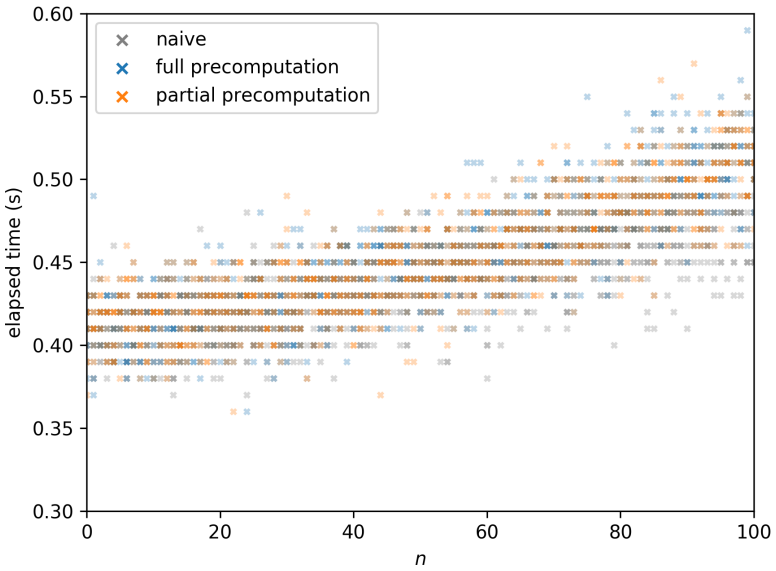


Figure 5.3: Elapsed time comparison where $m = 10$ and $0 \leq n \leq 100$ with 1000 samples.

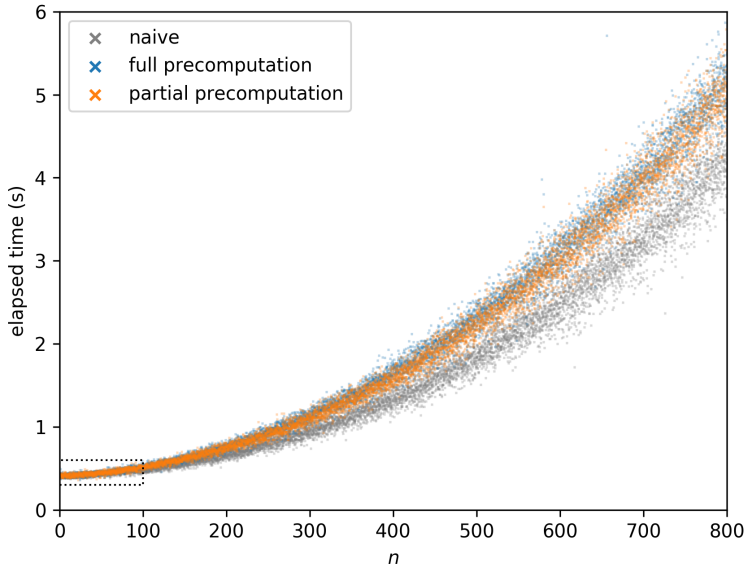


Figure 5.4: Elapsed time comparison where $m = 10$ and $0 \leq n \leq 800$ with 8000 samples.

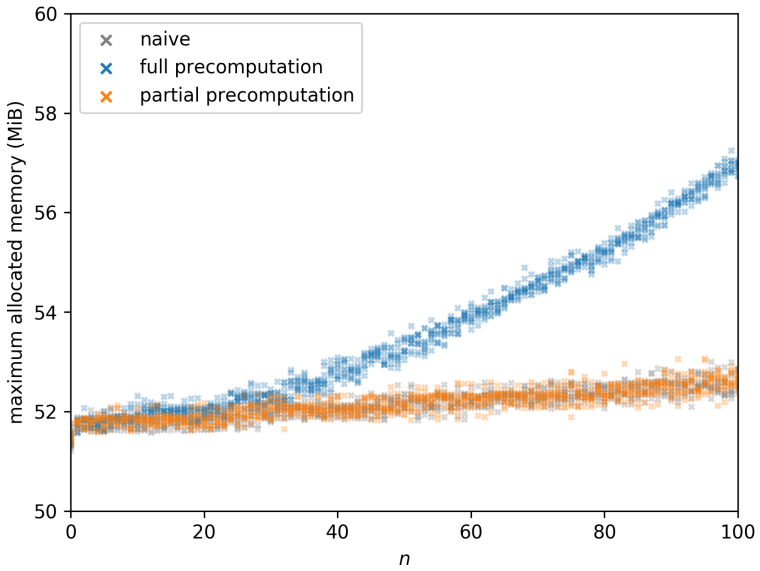


Figure 5.5: Maximum allocated memory comparison where $m = 10$ and $0 \leq n \leq 100$ with 1000 samples.

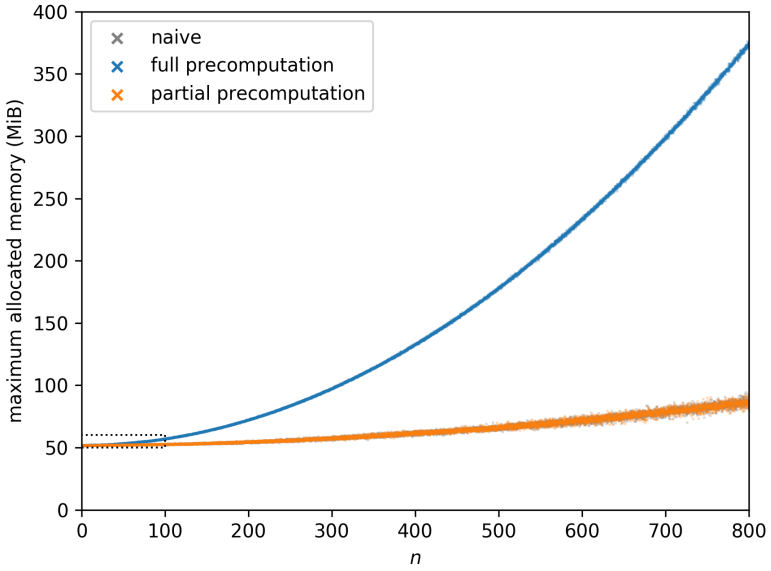


Figure 5.6: Maximum allocated memory comparison where $m = 10$ and $0 \leq n \leq 800$ with 8000 samples.

Appendix A

User Guide

The user guide is a tutorial for non-technical users to learn how to use the tool. The tool has been extensively tested on Linux, the tool has full functionality on Linux and basic functionality on Windows.

A.1 Installation

The tool has the following required dependencies:

- Python 3
- Tkinter
- NumPy
- Matplotlib
- Pillow

Tkinter is the default GUI package for Python and Matplotlib depends on NumPy so Tkinter and Matplotlib do not need to be installed explicitly. The tool has the following optional dependencies:

- Pyomo
- An optimiser, either CPLEX or GLPK

The optional dependencies are used to optimise schedules, these are not strictly required as users can input their own schedules. GLPK is recommended because of its licence and open-source development. The repository is found at `gitlab.doc.ic.ac.uk/mtl115/aes`. Please refer to the package's websites for troubleshooting. Alternatively, contact the author for assistance.

A.1.1 Linux Installation

The tool was tested on Ubuntu 18.04.1. Python is pre-installed so the following packages can be installed as below:

```
apt install python3-pip
apt install python-glpk
apt install glpk-utils
pip3 install matplotlib
pip3 install pillow
pip3 install pyomo
```

A.1.2 Windows Installation

The tool was tested on Windows 10. First, download Python from the official website, then setup the path environment variable so `python` can be executed on the command prompt. Afterwards, install the required dependencies as below:

```
python -m pip install matplotlib
python -m pip install pillow
python -m pip install pyomo
```

A.2 Usage

A.2.1 Getting Started

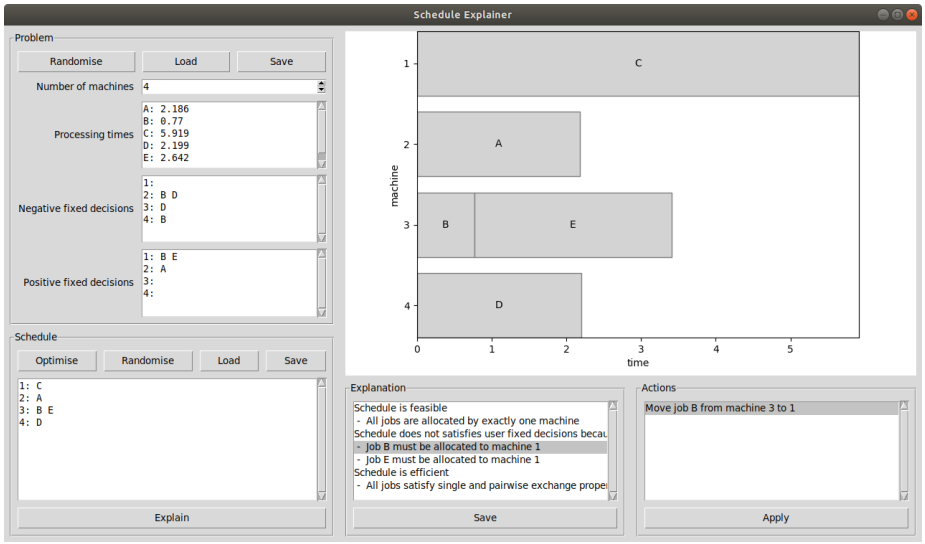


Figure A.1: Tool GUI

To start the tool, run `python3 main.py -g` on Linux or `python main.py -g` on Windows in the `src` directory supplied in the repository.

The makespan problem consists of the number of machines and job processing times. The tutorial will use a hospital setting, where nurses and patients are represented as machines and jobs respectively. Consider the following example where there are two nurses, Alice and Bob. and two patients, Charlie and Dave. Charlie's and Dave's appointment takes 15 and 10 minutes respectively.

To enter the example in the tool, nurses and patients are indexed. Hence, A represents Alice, B represents Bob for nurses and 1 represents Charlie and 2 represents Dave for patients. Numbers are used to index machines and letters are used to index jobs. The problem is to minimise the total completion time, which intuitively is the longest time any nurse has to work.

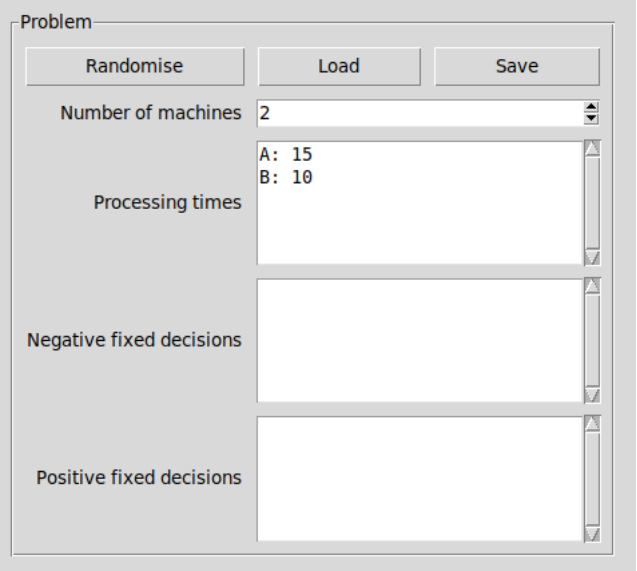


Figure A.2: Example problem input

Each line in the processing time textbox represents one job. The first line can be interpreted as: job A has processing time of 15 units, with following lines having similar interpretations. Negative fixed decisions represent jobs that cannot be assigned to machines. Positive fixed decisions represents jobs that much be assigned to a machine. Note that for all multi-line inputs, each line ends with a new line character.

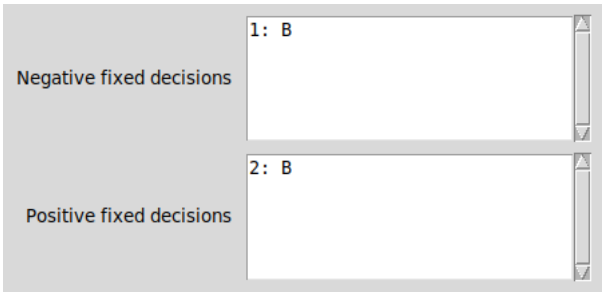


Figure A.3: Example fixed decisions input

Each line in each fixed decisions textbox represents one decision. The first line of negative fixed decisions can be interpreted as: machine 1 cannot be allocated to job B. The first line of positive fixed decisions can be interpreted as: machine

2 cannot be allocated to job B. In context of the example, this means Alice cannot be with Dave and Bob must be with Dave.

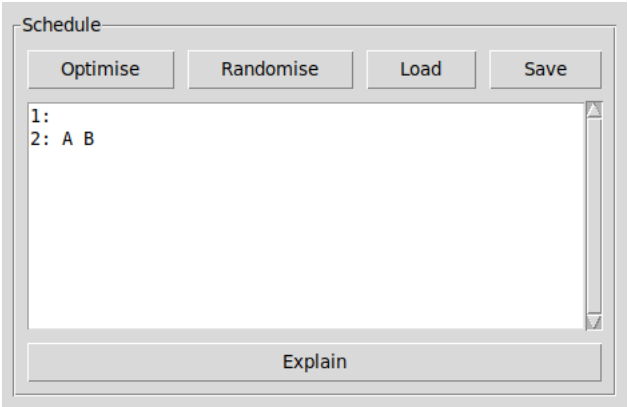


Figure A.4: Example schedule input

After defining the makespan problem, enter the above schedule. The schedule can be interpreted as: machine 1 has no allocated jobs; machine 2 have two allocated jobs, A and B. The `Optimise` button finds the optimal schedule using a solver, which is by default GLPK. To specify a solver, starting the tool with `python3 main.py -g -S SOLVER_NAME` where `SOLVER_NAME` is GLPK or CPLEX, for instance. Note that for large problems, optimisation may take a long time, so a solver time limit can be enforced by starting the tool with `python3 main.py -g -t TIME_LIMIT` where `TIME_LIMIT` is in seconds. The `Randomize` button generates some feasible schedule, which may violate fixed decisions. To explain the schedule, click the `Explain` button.

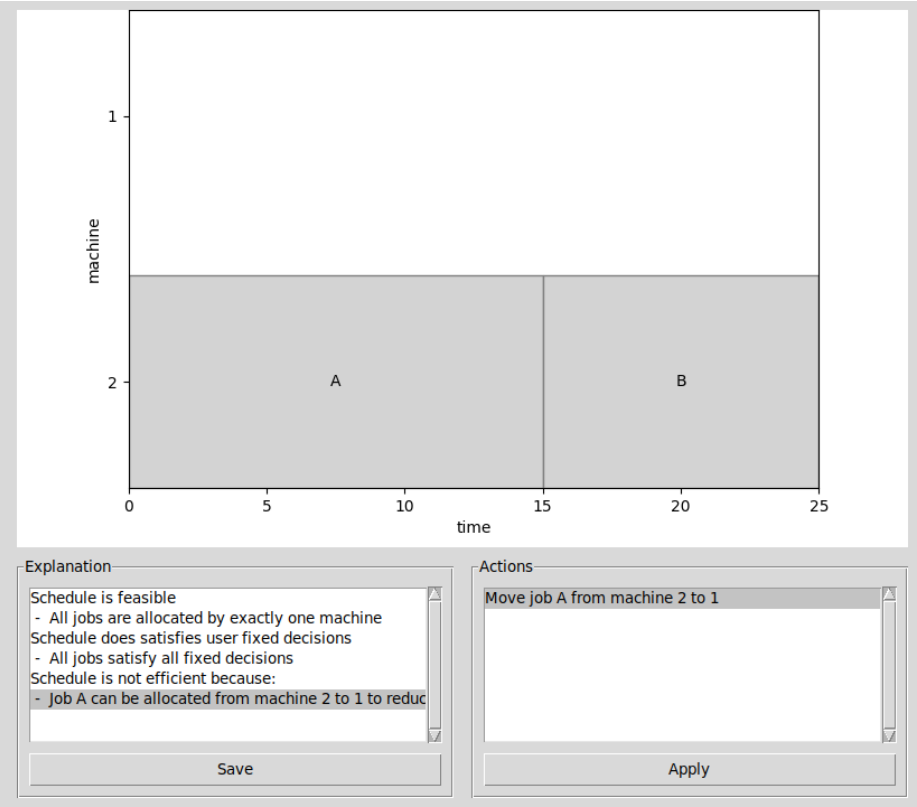


Figure A.5: Example explanation output

The explanation reasons on three concepts: feasibility, satisfaction of fixed decisions and efficiency. Feasibility ensures that each job is allocated once. Satisfaction of fixed decisions ensures the schedule does not violate negative and positive fixed decisions specified in the problem. Efficiency regards suggestions to improve the total completion time. To improve the schedule, select a line in the explanation listbox to address, then select a line in the actions listbox. An line of explanation may have many different approaches to address the problem or schedule. Click on the `Apply` button to improve the schedule.

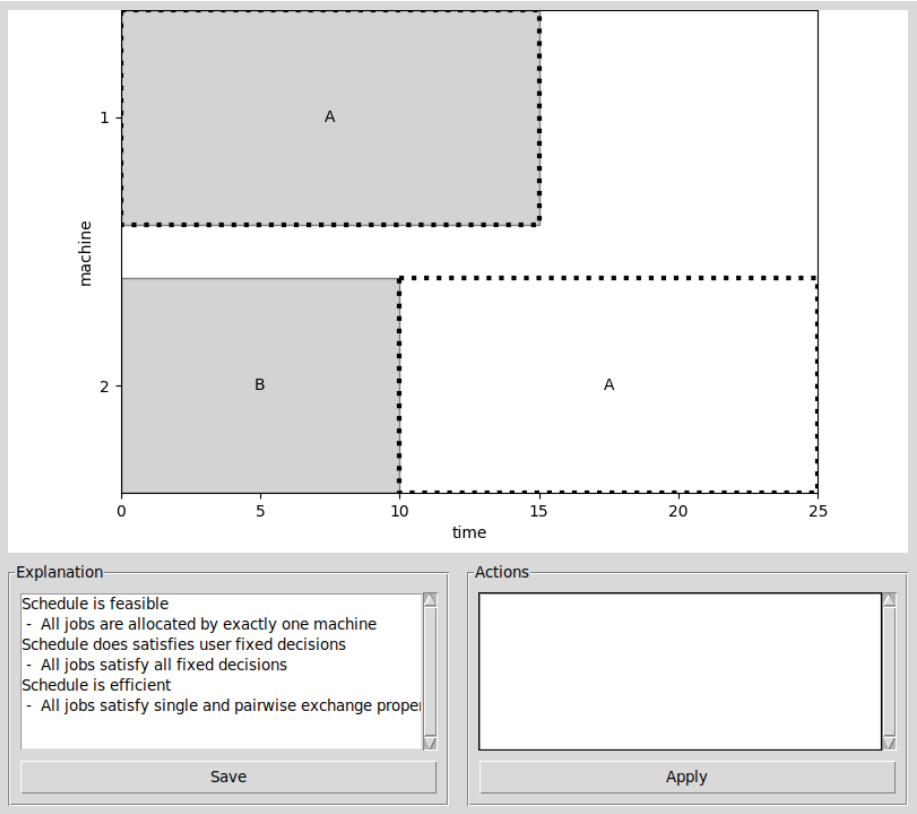


Figure A.6: Example explanation output

The example schedule only required one action to make the schedule efficient. However, many iterative actions may be required to reach an efficiency schedule. No further actions show that the schedule is feasible, satisfies fixed decisions and is efficient. The dot-highlighted boxes in the cascade chart illustrate newly and removed allocations compared to before the applying the action.

A.2.2 Command Line Examples

Help Command

```
> python3 main.py -h
usage: main.py [-h] [-g] [-e] [-p PROBLEM | -r [M]]
    [-O | -s SCHEDULE | -R] [-o OUTPUT] [--partial]
    [-t TIME_LIMIT] [-S SOLVER_NAME]

Explains makespan schedules using abstract argumentation
frameworks

optional arguments:
-h, --help            show this help message and exit
-g, --graphical       displays graphical user interface
-e, --explain         generate explanation
```

```
-p PROBLEM, --problem PROBLEM
-r [M], --random_problem [M]
    creates random problem with jobs and fixed decisions
    where M is the number of machines
-O, --optimise        uses SOLVER_NAME to find most efficient
schedule
-s SCHEDULE, --schedule SCHEDULE
-R, --random_schedule
-o OUTPUT, --output OUTPUT
    output filename for selected problem, schedule or
    explanation
--partial             use partial framework construction to
favour memory over CPU
-t TIME_LIMIT, --time_limit TIME_LIMIT
    maximum time for optimisation in seconds, use negative
    time_limit for infinite limit, default is unlimited
time
-S SOLVER_NAME, --solver SOLVER_NAME
    optimisation solver for schedule, default is 'glpk'
```

Random problem

```
> python3 main.py -r
4;
A: 1.822
B: 2.994
C: 6.444
D: 2.578
E: 2.386
;
1: B
2: A
3: D
4: B
;
1:
2: B
3: C
4:
```

Formally, this represents $m = 4$, $n = 5$, $\mathbf{p} = \begin{bmatrix} 1.822 & 2.994 & 6.444 & 2.578 & 2.386 \end{bmatrix}^T$, $D^- = \{\langle 1,2\rangle, \langle 2,1\rangle, \langle 3,4\rangle, \langle 4,2\rangle\}$ and $D^+ = \{\langle 2,2\rangle, \langle 3,3\rangle\}$.

Random schedule given previous problem

```
> python3 main.py -p example.problem -R
1: A
2: B C E
3: D
4:
```

Formally, this represents $\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Explantation given previous problem and schedule

```
> python3 main.py -p example.problem -s example.schedule -e
Schedule is feasible
-  All jobs are allocated by exactly one machine
Schedule does not satisfies user fixed decisions because:
-  Job C must be allocated to machine 3
-  Job D must not be allocated to machine 3
Schedule is not efficient because:
-  Job C can be allocated from machine 2 to 4 to reduce by 5.38
-  Jobs C and D can be swapped with machines 2 and 3 to reduce by 3.87
-  Job C can be allocated from machine 2 to 1 to reduce by 3.56
-  Job C can be allocated from machine 2 to 3 to reduce by 2.8
-  Job E can be allocated from machine 2 to 1 to reduce by 2.39
-  Job E can be allocated from machine 2 to 3 to reduce by 2.39
-  Job E can be allocated from machine 2 to 4 to reduce by 2.39
```

A.3 Known Limitations

- Holding down space for a button that requires significant computation results in permanent depressed visual of the button. This is a issue with Tkinter.

Bibliography

[1] E. K. Burke, P. De Causmaecker, G. V. Berghe, and H. Van Landeghem, *The State of the Art of Nurse Rostering.* Kluwer Academic Publishers, 2004.

[2] J. L. Gearhart, K. L. Adair, R. J. Detry, J. D. Durfee, K. A. Jones, and N. Martin, *Comparison of open-source linear programming solvers.* 2013.

[3] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems.* 2019.

[4] K. Cyras, D. Letsios, R. Misener, and F. Toni, *Argumentation for Explainable Scheduling.* 2018.

[5] S. Sohrabi, J. A. Baier, and S. A. McIlraith, *Preferred Explanations: Theory and Generation via Planning.* AAAI Press, 2011.

[6] P. Brucker, *Scheduling Algorithms.* Springer, 2007.

[7] A. W. Kolen, J. K. Lenstra, C. H. Papadimitriou, and F. C. Spieksma, *Interval scheduling: A survey*, vol. 54. Wiley Online Library, 2007.

[8] D. Walton, *Argumentation theory: A very short introduction.* Springer, 2009.

[9] P. M. Dung, *On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games*, vol. 77. 1995.

[10] M. Fox, D. Long, and D. Magazzeni, *Explainable Planning.* 2017.

[11] "Free online appointment scheduling calender software." `https://www.setmore.com`, 2019. Accessed on 25/01/2019.

[12] "Scheduling system." `http://web-static.stern.nyu.edu/om/software/lekin`, 2010. Accessed on 25/01/2019.