A dynamic approach to selecting timepoints for short-term scheduling with application to multipurpose facilities

Zachariah Stevenson,[†] Ricardo Fukasawa,[†] and Luis Ricardez-Sandoval[*,‡]

[†]*Dept. of Combinatorics and Optimization, University of Waterloo, Waterloo, ON, N2L 3G1, Canada*

[‡]*Dept. of Chemical Engineering, University of Waterloo, Waterloo, ON, N2L 3G1, Canada*

E-mail: laricard@uwaterloo.ca

Abstract

Choosing an efficient time representation is an important consideration when solving short-term scheduling problems. Improving the efficiency of scheduling operations may lead to increased yield, or reduced makespan, resulting in greater profits or customer satisfaction. When formulating these problems, one must choose a time representation for executing scheduling operations over. We propose in this study an iterative framework to refine an initial coarse discretization, by adding key timepoints that may be beneficial. This framework is compared against existing static discretizations using computational experiments on a scientific services facility. Using case studies from other applications in chemical engineering, we compare the performance of our framework against a previously reported time-discretization approach in the literature. The results of these experiments demonstrate that when problems are sufficiently large, our proposed dynamic method is able to achieve a better tradeoff between objective value and CPU time than the currently used discretizations in the literature.

1 Introduction

Scheduling is concerned with how and when to execute operations to optimize a chosen objective such as maximizing profits, or minimizing costs, subject to operational constraints such as deadlines, capacity, or available resource limitations. It is common practice for many industries to make use of scheduling as an optimization problem to guide their progress and meet various economic objectives,[1–5] and proper scheduling can greatly increase efficiency and therefore is of great practical importance. Scheduling operations over a relatively short period of time, such as a day, a shift or a week is often referred to as *short-term scheduling*. We are interested in scheduling in the context of short-term scheduling of a multipurpose plant.

A *multipurpose plant* is a facility which has a set of *units*, such as machines or workers, that may carry out a variety of *tasks*, i.e. the plant may serve more than a single purpose, such as producing several different products. The problem of scheduling a multipurpose plant then can be seen as a variant of the job shop scheduling problem.[6,7] Orders arrive at a multipurpose facility, and each order has a set of samples that comprise it and a sequence of tasks that the samples must undergo in order. This sequence of tasks is called the *path* of an order. The goal is to generate a schedule for the facility, which dictates what samples to assign to which tasks over the time horizon such that an objective is optimized. This schedule must also abide by various operational constraints of the problem such as unit resource limitations and material balance constraints.

Throughout this work, we model these scheduling problems as mixed integer linear programs (MILP), meaning that we restrict the objective function, and constraints of the model to be linear, and allow some variables to be integer. There are several different MILP scheduling models that exist, and one of the key classifications of these models is the time representation that is used.[8]

The time representation determines when operations may be scheduled, and can play a critical role in determining the computational cost of solving the model and the final

solution quality.[8–10] We call each point in time where an operation may be scheduled a *timepoint*. There are two main classes of time representation: continuous time representation and discrete time representation.[11,12]

Continuous time representations allow events to happen at arbitrary points in time during the scheduling horizon, with the selection of where these points should be placed being decided by the model.[13–15] Because the model is able to choose precisely where timepoints should be located, this technique ensures that we obtain the best solutions using this representation.[8] However, one must provide the model a fixed number of timepoints to allocate as input. If the model is allowed to allocate too few timepoints, solution quality may decrease, however if the model is allowed to allocate many timepoints, CPU time may drastically increase. In general, selecting a suitable number of timepoints to allocate for the model may be challenging, particularly for large-scale industrial applications where the number of time points may be relatively large.[16]

Discrete time models instead fix a priori the timepoints at which scheduling decisions may be made to a subset of points during the scheduling horizon.[17,18] We call the set of timepoints that the schedule may use the *time grid*. The difficulty with this representation is in choosing a suitable time grid to provide the model, namely instead of only selecting the number of timepoints and allowing the model to choose the placement, we also give the placement of timepoints as input. There is once again a tradeoff between how coarse or fine the discretization used is, the quality of the resulting schedule and the amount of CPU time required to solve the model. Despite the continuous time formulation allowing the model to use precise points in time, multiple works have concluded that using a discrete time representation results in better performance than using a continuous time representation for scheduling multipurpose facilities.[9,12,19] Moreover, these works have shown that the continuous time formulation version of the problem may be very computationally expensive to solve beyond using only a few timepoints, for large scale applications. However, recent work has shown promise in using the unit-specific event-based continuous time formulation

for large scale applications.[20] In this work we limit ourselves to models using a discrete time representation.

Depending on the specific details of the plant being scheduled (e.g. the number of orders that arrive, the size of the plant, and the time horizon), scheduling industrial-scale multi-purpose facilities may require solving very large models. Therefore, the question naturally arises: how do we solve these *large* scheduling problems more efficiently? This question prompts us to take a look at timepoint representations in more detail. In general, if we knew exactly which timepoints were needed to achieve the optimal solution obtained by using a continuous time representation, we would be able to greatly reduce the size of the model by using only these necessary timepoints. However, it is unlikely that we know this information beforehand, yet we must still choose a time grid to use for a discrete model.

As mentioned above, choosing which timepoints to include in the time grids is in general not obvious and can greatly impact the performance of the model. Furthermore, there have been a very limited number of works that have considered how to tailor time grids for individual problem instances.[10,21,22] Here we mean an *instance* of a problem to be a problem with given input data, as opposed to the general problem itself. The purpose of this work is to study this issue of choosing an appropriate time grid. Below we briefly summarize the aforementioned works that consider this issue and how our approach differs from those studies.

Dash et al.[21] studied the traveling salesperson problem with time windows. They considered partitioning their time windows into smaller sub-windows, which may be thought of as discretizing the time windows, and presented cutting planes which are more effective than those found in the literature by exploiting their window division. They then iteratively solved a linear program to find a good partition for the time windows as a pre-processing routine before solving their instance of the traveling salesperson problem with time windows. By using their method, those authors were able to solve several previously unsolved benchmark instances.

Velez and Maravelias[10] showed that they are able to generate non uniform time grids based on the instance input data such that the optimal solution obtained by using these time grids has equal objective value to the optimal solution obtained by using an arbitrarily fine uniform discretization. Furthermore, they show that this method of choosing non uniform time grids leads to a much smaller problem size than that obtained by using a super-fine uniform discretization. However, to ensure that this guarantee holds, their algorithm may include many timepoints making the resulting model very computationally taxing to solve, especially for large problem instances. We will refer to these non uniform discrete (NUD) time grids throughout this work. The NUD time grids allow each process considered to operate using its own set of timepoints. This offers benefits by allowing processes which operate on different time scales to have different time grids, e.g. a process that takes days and a process that takes minutes can both use suitable time grids.

Boland et al.[22] presented a method for iteratively refining the set of timepoints that they considered for solving the service network design problem. They used a different approach than that of Velez and Maravelias, by beginning with only a few timepoints initially and then determining where to add timepoints based on the obtained solution from solving a relaxation of their problem. They are able to show that their method will also terminate with an optimal solution whose objective value is equal to that obtained by using an arbitrarily fine uniform discretization. Moreover, they showed that their method performs well in practice through a computational study.

Though those three works aim to address the issue of time discretization, their proposed approaches are either tailored to their specific application,[21,22] or are computationally expensive for large-scale problems[10].

In this work, we present a generalized framework for iteratively refining time grids for scheduling multipurpose facilities. We propose several heuristics for deciding where to add and remove timepoints from the scheduling model's time grid between iterations of the framework. The proposed approach does not depend much on the particular application,

but rather on the fact that a time-discretized representation of the scheduling process is used. The efficacy of this framework is evaluated through computational experiments comparing its performance against the currently used time grids for scheduling discrete time multipurpose plants in the literature. We also compare our method against the time grid generation algorithm and model of Velez and Maravelias[23]. The results of these experiments show that the proposed method may obtain significant performance improvements over the current time grids in use without a substantial increase in CPU time. Moreover, since the performance of the proposed framework remain relatively stable over a range of problem sizes, this framework provides a stepping stone toward improving instance-agnostic methods for choosing time grids in discrete time based short-term scheduling models.

Note that the focus of this work is on developing heuristic approaches to our scheduling problem with empirical evidence obtained through computational experiments which demonstrate that our heuristics seem to perform well in our test cases. In particular, we base our case studies on an industrial-scale multipurpose plant to validate the proposed method's performance. Consequently, we present our framework with respect to the model that we present in section 2.2 which was used for our primary case study. However, our framework is not dependent on this model, and may be applied to the general model presented by Velez and Maravelias in [23] based on the STN representation which readers may be more familiar with. We present how we applied our framework to this model and the corresponding results in section 4.6.

The structure of this study is as follows. In section 2 we discuss the scheduling problem we solve in more detail, present the model formulation used for our experiments, and discuss other works in the literature which have considered specialized time grids. In section 3 we present the proposed framework for dynamically constructing time grids along with the heuristics used for selecting which timepoints to add and remove. The efficacy of the framework is evaluated using computational experiments and the results are discussed in section 4. Concluding remarks and future work considerations are presented in section 5.

2 Background

2.1 Time Layered Graphs

We now present the definition of a time layered graph to establish notation, before discussing the scheduling problem in more detail and corresponding model we use in section 2.2. Time layered graphs are useful when representing problems which have a network of states, and decisions must be made based on spatial locations in the graph and time, e.g. routing material between states and over a time horizon. The use of these graphs typically arises in transportation routing problems such as bus or flight routing.[22,24–26] They augment the standard static network flow models by adding an extra dimension (time) allowing flows to change over time. This time component will be used in this work to model the scheduling of when to start batch processes in a multipurpose facility. The usage of graphs and graph-centric approaches has become more common in the chemical engineering community in recent years. Graph approaches have been proposed to address problems related to distribution networks[27,28], hydrocarbon generation[29,30], process-network synthesis[31,32], and distributed control[33–35]. These works have drawn from various topics and structures from network theory such as bipartite graphs[29–33,36], community detection[33], cycle detection[36], and graph partitioning[27] and applied these to the aforementioned topics in chemical engineering. Process graphs, or "P-graphs" have even been developed as bipartite graphs with extra structure to model process networks.[37] It is clear that there are many applications in chemical engineering that lend themselves to graphical representations and that there is interest in the community toward these sorts of approaches.

We assume that there is an underlying directed network $G = (V, A)$ where our node set V is comprised of a set of "states" (e.g. physical locations such as stations or holding depots, or tasks), and our directed arc set A indicates how we may transition from state to state. For each state u, there is a discrete set of (state-specific) integer timepoints $\varepsilon(u)$, indicating at which times we may leave state u. We denote the k'th timepoint of $\varepsilon(u)$ as $\varepsilon(u, k)$. We

denote the set of all sets of timepoints as $\varepsilon = \{\varepsilon(u) : u \in V\}$. Furthermore, we denote the *next* timepoint in the timepoint set of state u after or at time t as $n(u,t) = \min\{t' : t' \in \varepsilon(u), t' \geq t\}$ if it exists, otherwise ∞. At each point in time, t, we assume that there is an integer travel time associated with traveling from state u, to state v, denoted $\tau(u,v,t)$. Note that restricting ourselves to integral points in time for timepoints and integer travel times is without loss of generality (as long as the original time data is rational) as we may discretize time as finely as needed.

With respect to any graph $G = (V, A)$, we denote the set of arcs leaving a node $q \in V$ as $\delta_G^+(q)$, and the set of arcs entering q as $\delta_G^-(q)$. Similarly, we denote the set of nodes that can be reached by traversing a single arc from node $q \in V$ as $N_G^+(q) = \{v \in V : \exists(q,v) \in A\}$, and the set of nodes which can reach q by traversing a single arc as $N_G^-(q) = \{v \in V : \exists(v,q) \in A\}$. When the context of which graph we are discussing is clear, we will omit the subscript to make the notation less cumbersome.

To obtain our time layered graph $G^* = (V^*, A^*)$ we perform the following construction. The set of nodes $V^* = \{(u,t) : u \in V, t \in \varepsilon(u)\}$ is the set of all pairs of states and timepoints for each given state. Arcs either start and end at two distinct states, representing the transition from one state to another, or start and end at the same state, representing staying in the same state. We denote the former set of arcs as A_L^* ("leaving arcs"), and use the next timepoint function $n(v,t)$ and travel function $\tau(u,v,t)$ to add arcs from (u,t) to $(v, n(v, t+\tau(u,v,t)))$ for each neighbor $v \in N_G^+(u)$. The latter set of arcs are denoted as A_H^* ("holding arcs") and is made up of arcs that go from (u,t) to $(u, n(u,t+1))$. The full set of arcs of G^* are $A^* = A_L^* \cup A_H^*$.

In the proceeding discussions, when we refer to a time layered graph we will assume that we also have access to the underlying graph G, travel function τ and timepoint sets ε. Moreover, we will use G to denote a graph with no time components and G^* to denote a time layered graph obtained from G (with travel function τ and timepoint sets ε).

2.2 Problem Description

Using the notion of time layered graphs from section 2.1, we may now discuss our scheduling problem and corresponding model. Assume that we start with a time layered graph $G^* = (V^*, A^*)$ with underlying network $G = (V, A)$, timepoint sets ε, and travel function τ, as described in section 2.1. We have a set I of orders that must be scheduled. Each order $i \in I$ is comprised of a discrete set of samples and a path $\Pi(i) = (\Pi(i,1), \Pi(i,2), \ldots, \Pi(i,m))$ of tasks (which correspond to the "states" described in section 2.1), which order i must be routed through, in order. The "samples" of an order correspond to the amount of material that must be processed for a given order. Note that the samples of a single order may be split up into discrete batches and processed at different times, as long as each individual sample follows the order's path. We denote the number of samples of order i that arrive at task u at time $t \in \varepsilon(u)$ as $\alpha(i, u, t)$.

Tasks of the underlying graph, $u \in V$, have a number of properties. We denote the number of units that may carry out task u as $\rho(u)$, and the capacity of each unit as $\kappa(u)$. These units can be considered to be any entities which carry out a given task, for example machines or personnel. Note that all of these units are considered identical, and we will refer to units that perform task u as u-units. Moreover, the previously discussed travel function τ corresponds to the processing times of task u, which we allow to depend both on the time that u is being used and the subsequent task v that will process the products after u. These properties restrict how order samples may flow through the network; for instance, sending m samples from task u to a different task v at time t requires that $\lceil m/\kappa(u) \rceil$ u-units are available to be utilized at time t. Note that we assume that packing samples into units is not an issue as we allow samples from a single order to be processed by different units and units to transport samples from different orders, as long as there is sufficient capacity. We assume that arcs that start and end at the same state, which represent samples waiting at a task, have unlimited capacity and do not require any unit allocation as the samples are just being *held* at the task.

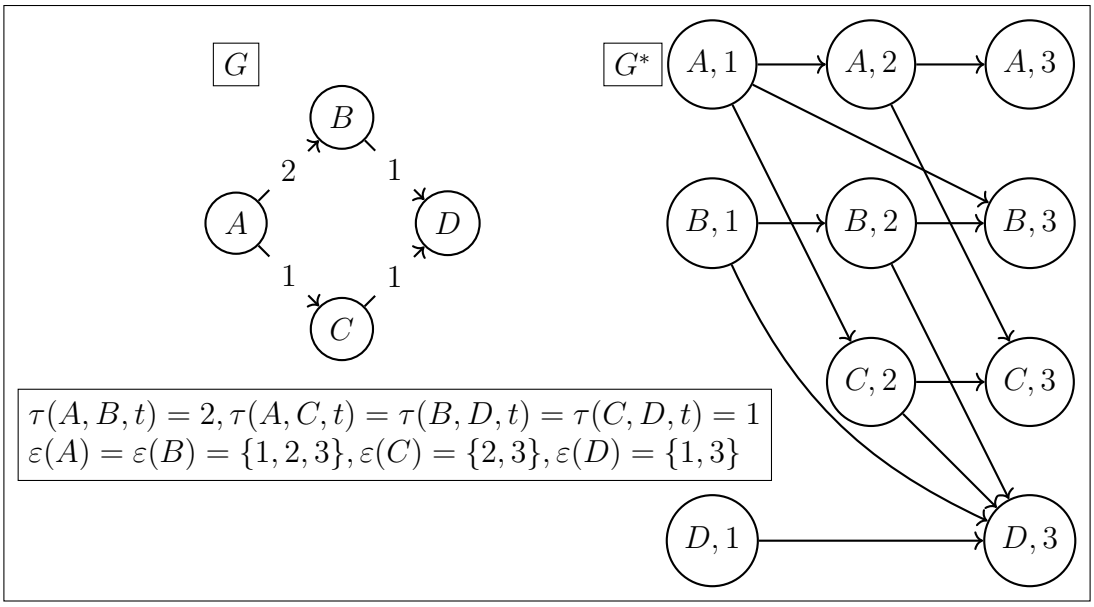


Figure 1: Illustration of a time layered graph.

Units for task u that are used to process samples before they are processed by a v-unit at time t, become available to be used again after some time $\omega(u, v, t)$. We call $\omega(u, v, t)$ the return time of u to v at t.

Let $H(i, u, t)$ denote the head of the arc in G^* corresponding to samples of order i at task u being sent to the subsequent task in its path at time t. To make this notation more clear, let us consider the following example. Consider $G = (\{A, B, C, D\}, \{(A, B), (A, C), (B, D), (C, D)\})$, with travel function $\tau(A, B, t) = 2, \tau(A, C, t) = \tau(B, D, t) = \tau(C, D, t) = 1\ \forall t$, and timepoint sets $\varepsilon(A) = \{1, 2, 3\}, \varepsilon(B) = \{1, 2, 3\}, \varepsilon(C) = \{2, 3\}, \varepsilon(D) = \{1, 3\}$. Following the construction of a time layered graph described above, the graph G along with the resulting time layered graph G^* are shown in Figure 1. Suppose the path of order i is $\Pi(i) = (A, B, D)$ and the path of order j is $\Pi(j) = (A, C, D)$. Then we have $H(i, A, 1) = (B, 3)$, $H(j, A, 1) = (C, 2)$, $H(j, A, 2) = (C, 3)$, and $H(i, B, 1) = H(i, B, 2) = H(j, C, 2) = (D, 3)$. Note that even though the travel time of task B is 1, since there is no node for task D at time 2, then the arc from $(B, 1)$ to D goes to $(D, 3)$, instead of $(D, 2)$. Moreover, depending on the choice of parameters, G^* may have unreachable nodes. Consider the node $(B, 1)$, since B is obtained

from A then any arcs coming from $(B,1)$ or $(B,2)$ cannot be used. Nevertheless, we may still construct G^* given G, τ, and ε using the process described in section 2.1.

We have three types of variables in the present problem. Let $y(u,v,t)$ denote the number of u-units dispatched to task v at time t. This allocation allows up to $\kappa(u)y(u,v,t)$ samples to be sent from task u to task v at time t. Let $x(i,u,t)$ denote the number of samples of order i sent from task u at time t to $H(i,u,t)$. Note that the sum of the x variables over all orders i, for a single task u at time t corresponds to the batch size for task u at time t. Let $z(i,u,t)$ denote the number of samples of order i that wait at task u at time t, i.e. instead of proceeding to $H(i,u,t)$, the samples proceed to $(u, n(u,t+1))$. Note that the x and z variables are flow variables on the arcs of G^*, and the resulting problem (P) is a flow problem. We now present the complete model (P), based on the model presented in [38].

$$\max_{x,y} \quad f_{G^*} = \sum_{i\in I}\sum_{u\in\Pi(i)}\sum_{t\in\varepsilon(u)} f(i,u,t)x(i,u,t) - \sum_{u\in V}\sum_{v\in N_G^+(u)}\sum_{t\in\varepsilon(u)} c(u,v,t)y(u,v,t) \quad (P)$$

$$\text{s.t.} \quad x(i,u,t) + z(i,u,t)$$

$$- \sum_{(w,t'):H(i,w,t')=(u,t)} x(i,w,t') = \alpha(i,u,t), \quad \forall\, i\in I, u\in\Pi(i), t\in\varepsilon(u) \quad (1)$$

$$\sum_{v\in N_G^+(u),t':t'\le t<t'+\omega(u,v,t')} y(u,v,t') \le \rho(u), \qquad \forall\,(u,t)\in V^* \quad (2)$$

$$\sum_{i\in I:\exists k,\Pi(i,k)=u,\Pi(i,k+1)=v} x(i,u,t)$$

$$-\kappa(u)y(u,v,t) \le 0, \qquad \forall\,(u,t)\in V^*, v\in N_G^+(u) \quad (3)$$

$$\sum_{i\in I:\exists k,\Pi(i,k)=u,\Pi(i,k+1)=v} x(i,u,t)$$

$$-\kappa(u)(y(u,v,t)-1) \ge 1, \qquad \forall\,(u,t)\in V^*, v\in N_G^+(u) \quad (4)$$

$$x(i,u,t) \ge 0, \qquad \forall\, i\in I, u\in\Pi(i), t\in\varepsilon(u) \quad (5)$$

$$z(i,u,t) \ge 0, \qquad \forall\, i\in I, u\in\Pi(i), t\in\varepsilon(u) \quad (6)$$

$$y(u,v,t) \ge 0, \qquad \forall\,(u,t)\in V^*, v\in N_G^+(u) \quad (7)$$

$$x, z, y \text{ integral}, \quad (8)$$

Constraints (1) are flow constraints which ensure that the in-flow that comes into a node (u,t) is equal to the out-flow that leaves (u,t) for each order i. Constraints (2) ensure that units are not over-allocated at any point in time. Constraints (3) ensure that enough u-units are allocated at time t to support the amount of samples scheduled to be processed. Constraints (4) enforce that we do not dispatch resources without need, i.e. we do not waste unit usage in our solution. Constraints (5) - (8) are non-negativity and integrality constraints for the variables. Note that both the z and y variables are completely determined by the values of the x variables; hence for convenience, we will refer to a solution to (P) as x but assume that the z and y variables are stored and accessible as well.

The objective function is a general weighted sum on the x and y variables which resembles maximization of throughput. $f(i,u,t)$ is the per sample objective weight of sending samples of order i from task u at time t to $H(i,u,t)$, and $c(u,v,t)$ is the per unit cost of dispatching u-units to task v at time t. There is only one requirement on f and c. To state such requirement, we need to first make a definition. Let us call a solution x', a "backward time-shifted" version of x if the following conditions hold: x and x' are solutions to (P) such that x' can be obtained from x by shifting some number of samples (δ) from a order i, scheduled to leave task u at time t, to an earlier time t'. More precisely, $\exists i \in I, u \in \Pi(i), t, t' \in \varepsilon(u), \delta \in \mathbb{Z}, \delta > 0$ such that $t' \leq t$, $x'(i,u,t') = x(i,u,t') + \delta$, $x'(i,u,t) = x(i,u,t) - \delta$, and $x'(i,u,t) = x(i,u,t)$ otherwise. Now, the only assumption that is considered on the objective function is that if x' is a backward time-shifted version of x, then $f_{G^*}(x') \geq f_{G^*}(x)$, i.e. the backward time-shifted solution is no worse than the original solution. Note that this is a fairly sensible assumption as it is natural to assume that only adding delays to a schedule without otherwise changing it will not improve its objective value.

The model (P) has been constructed to easily accommodate some of the features of the particular application that we study. However, it is similar to other general process network scheduling models proposed such as in[18] and [23], and was used to model a semiconductor

processing plant in [39]. In particular, the method for generating time grids for our model presented in section 3 may be tailored slightly to fit the general batch scheduling model presented in [23]. In section 4.6, we obtain results comparing our method applied to Velez and Maravelias' model to their proposed method.

3 Timepoint Modification Framework

In this section we discuss the framework for solving our scheduling problem (P) from section 2.2, using an iterative approach. We present Figure 2 to give an overview of how the framework proceeds, before describing the framework in more detail below.

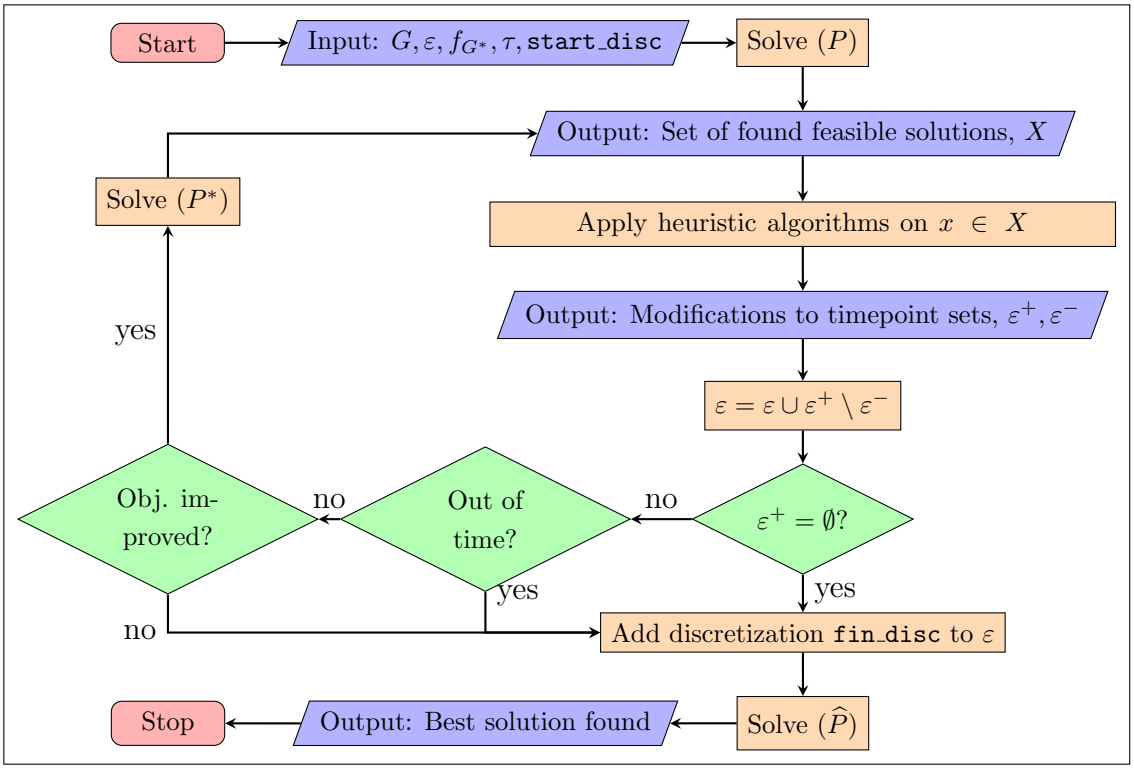


Figure 2: A flowchart outlining the dynamic timepoint framework.

We begin by instantiating our timepoint sets, ε, to some sufficiently coarse uniform discretization which we will call `start_disc`. We then use a MIP solver, e.g. Gurobi or CPLEX, to solve problem (P) and we record a list, X, of any feasible solutions found by the optimization solver. Note that the parameters we use for the solver or the choice of

solver are not discussed here, we assume that we can obtain some list of feasible solutions to our problem using an abstract "solver". We would like to emphasize that the need to find a feasible solution is key to our framework; however the solution (or solutions) may be obtained by various means. For example, one could employ a greedy approach to obtain a feasible solution, or in fact scheduling no operations would also be a (albeit bad) feasible solution. Depending on the specific application being solved for and any additional information known in advance, one may be able to derive other simple algorithms for obtaining feasible solutions. Note that the quality of the initial solutions can impact the effectiveness of the rest of the framework. Beginning with better solutions will allow more of the search space to be cutoff, and therefore may make the iterative solves faster.Using each obtained solution $x \in X$ as input, we proposed a series of algorithms, i.e. Get Instant Start Timepoints, Get Overloaded Timepoints, and Get Dominated Timepoints (which will be discussed in more detail later) to obtain a set of timepoints ε^+ to add to our current timepoints and a set of timepoints ε^- to remove from our current timepoints. That is, algorithms Get Instant Start Timepoints and Get Overloaded Timepoints produce a set of timepoints to add for each solution $x \in X$, and Get Dominated Timepoints produces a set of timepoints to remove for each $x \in X$. To obtain ε^+, we take the union of all of the sets of timepoints to add, and to obtain ε^-, we take the intersection of all of the sets of timepoints to remove. By choosing to use the union for added timepoints, we add all timepoints that are identified as being potentially beneficial, and by choosing to use the intersection for removed timepoints, we remove only those timepoints which were considered not needed for every solution $x \in X$.

We proceed to construct a new instance (P^*) of problem (P), by adding ε^+ and/or removing ε^- from the current timepoints ε. The best solution found previously, x, is used to generate a new solution x^* which is feasible for our newly formed problem, (P^*). x^* is given as an initial solution to our new problem (P^*) and we repeat this process of solving the current problem, using the solution(s) found to generate new timepoints to add and remove, and then modifying the current set of timepoints. We provide the solution x^* as an

initial solution for (P^*) so that the solver may use x^* as a feasible solution, with the goal of optimizing (P^*) more quickly than if no initial solution was provided. This continues until we reach a stopping criterion such as reaching a computational time limit, ending in an iteration such that algorithms Get Instant Start Timepoints and Get Overloaded Timepoints do not produce any new timepoints to add, or completing an iteration in which there is insufficient improvement between the new solution and the previous solution. We call the allowed time between solutions `sols_tl`, and the time limit for the aforementioned iterative procedure `its_tl`. That is, suppose we are solving a single instance of (P) and have already found an incumbent solution. If no better solution is found within `sols_tl` seconds of finding the latest incumbent solution, then we quit the solution procedure for the iteration and return with any solutions that were found. The time limit for the entire iterative procedure, `its_tl`, is checked between iterations and if exceeded, we stop iterating and continue. The way this was implemented in our experiments was to call the MIP solver with the remaining time left of the iterative procedure (`its_tl` - (current time - start time)) as the time limit and to provide two types of callbacks to the MIP solver. The first type of callbacks are triggered any time a new solution is found and stores the solution in an array. These solutions are used by the timepoint modification algorithms later to determine which timepoints to add and remove. The second type of callbacks are triggered intermittently and check how long it has been since the last solution was found. If too much time has elapsed then we quit the MIP solver and proceed to the next iteration.

After this process of adding and removing timepoints iteratively has reached a stopping criteria and terminated, we store the best solution found previously, $\hat{x}$, add all of the timepoints associated with some chosen discretization (which we call `fin_disc`) to our timepoint sets, create a problem $(\widehat{P})$, and solve this problem with input $\hat{x}$ given as an initial solution. We call the time limit passed to the solution method for solving this final problem `fin_tl`. By adding all of the timepoints from a discretization which is assumed to provide acceptable solutions, we aim to find any solutions better than $\hat{x}$ we may have missed earlier during our

iterative process. However, despite adding possibly many new timepoints during this final step, we are still able to take advantage of the incumbent solution ($\hat{x}$) to be able to optimize ($\widehat{P}$) faster than if we optimized ($\widehat{P}$) from scratch as shown in Section 4. The framework terminates by returning the best solution found for problem ($\widehat{P}$).

Table 1 provides a summary of the parameters used in the framework with their corresponding descriptions for reference.

Table 1: Description of parameters used for the framework.

Parameter Name	Parameter Description
`start_disc`	The timepoint discretization to use for building the first iteration of problem (P)
`fin_disc`	The timepoint discretization to use for building the final solve of problem (P), after the iterative procedure
`its_tl`	The total amount of time to allow for the iterative procedure (before adding `fin_disc` to (P) and solving ($\widehat{P}$))
`obj_thresh`	The required objective improvement between iterations, measured as new objective value as a factor of old objective value
`sols_tl`	The allowed time between solutions during a single iteration. If more than `sols_tl` time has passed since finding the last solution, then quit the current iteration
`fin_tl`	The allowed time for solving the final problem ($\widehat{P}$), after adding `fin_disc` to (P)

3.1 Adding/removing timepoints

We now discuss the heuristics for adding and removing timepoints. The driving idea of the heuristics is to add timepoints such that actions from a previous schedule may be shifted to happen earlier in time, and to remedy cases where a task's units cannot be efficiently utilized because of a lack of timepoint availability. Similarly, we remove timepoints which seem to be unnecessary, from a time-shifting perspective, to reduce the size of the resulting optimization problems. Our goal is that by applying these heuristics iteratively, we may shape the starting coarse timepoint sets into sets which include sufficient timepoints for obtaining high quality solutions, but without the unneeded timepoints that a fine uniform discretization

may contain. Note that this is where we use the assumption that the objective function of (P) is such that backward time-shifting solutions does not result in worse objective values. If this assumption holds, then we expect that shifting scheduled events to happen earlier in time (possibly allowing for more events to be scheduled) will improve the best objective value over each iteration.

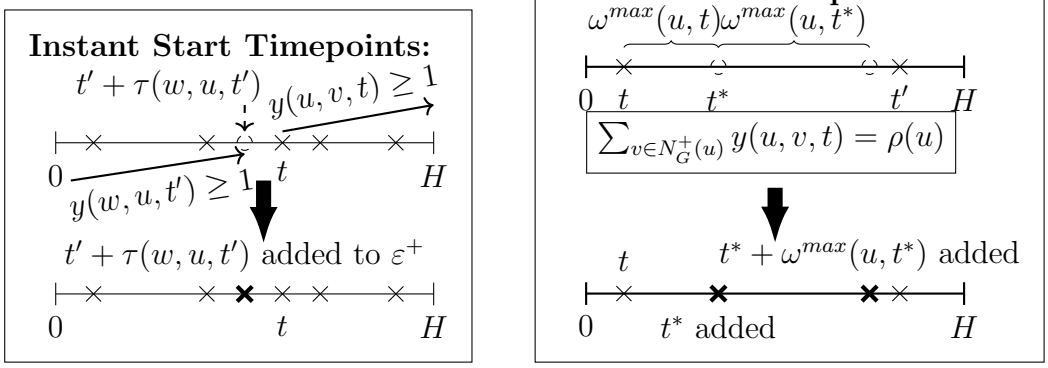
We introduce the following definition: Let $\omega^{max}(u,t) := \max_{v \in N_G^+(u)} \omega(u,v,t)$ be the longest return time for a u-unit which is utilized at time t. Similarly, let $\omega^{min}(u,t) := \min_{v \in N_G^+(u)} \omega(u,v,t)$ denote the minimum return time for a u-unit which is utilized at time t.

Algorithm Get Instant Start Timepoints describes how timepoints are added such that samples that leave from a task w to a task u may leave u upon arrival. Figure 3a provides a graphical representation of how instant start timepoints are identified and added to the timepoint sets. Namely, suppose a task u has units dispatched at timepoint t ($\sum_{v \in N_G^+(u)} y(u,v,t) > 0$). We then consider all vertices (w,t') of G^* that have an arc from (w,t') to (u,t) that is being used ($y(w,u,t') > 0$) and whose units will arrive before time t ($t' + \tau(w,u,t') < t$). For each of these neighbors, we add a new timepoint at the actual time that samples departing from this vertex arrive at task u (i.e. $t' + \tau(w,ut')$ is added to $\varepsilon(u)$). This procedure is carried out over all vertices (u,t) of G^* such that u-units are dispatched at time t.

We call the timepoints added by this heuristic, "instant start timepoints" as they allow samples which leave a task w and arrive at a task u to begin departing u immediately upon arrival ($t' + \tau(w,u,t')$) instead of waiting until time t to leave. The timepoints that this heuristic adds allows samples which are being scheduled by the model to have fewer instances where they must wait at a task before proceeding to the next one in their path. We anticipate that by including these timepoints, future schedules may obtain greater objective value by shifting operations to happen earlier, possibly allowing more operations to be scheduled later in the horizon.

Algorithm 1 Get Instant Start Timepoints

function GET INSTANT START TIMEPOINTS(G, G^*, ε, x)
  $\varepsilon^+ \leftarrow \{\varepsilon^+(u) = \emptyset : u \in V\}$ ▷ Instantiate set of timepoints to add
  for all $u \in V$ **do**
    for all $t \in \varepsilon(u)$ **do**
      if $\sum_{v \in N_G^+(u)} y(u, v, t) > 0$ **then** ▷ Samples are leaving (u, t) in the solution
        for all $(w, t') \in N_{G^*}^-((u, t)) : t' + \tau(w, u, t') < t$ **do** ▷ Consider neighbors such that samples departing from that neighbor arrives at u before t
          if $y(w, u, t') > 0$ **then** ▷ Arc from (w, t') to (u, t) that is being used in solution
            $\varepsilon^+(u) \leftarrow \varepsilon^+(u) \cup \{t' + \tau(w, u, t')\}$ ▷ Set $t' + \tau(w, u, t')$ to be added to $\varepsilon(u)$
    $\varepsilon^+ \leftarrow \varepsilon^+ \cup \{\varepsilon^+(u)\}$
  return ε^+ ▷ Return set of instant start timepoints to add



(a) New instant start timepoints.

(b) New overloaded timepoints.

Figure 3: Addition of new timepoints. Time flows along axis, with "X" representing the presence of a timepoint in $\varepsilon(u)$.

Algorithm Get Overloaded Timepoints describes how timepoints are added for tasks which are heavily utilized but whose timepoint sets is lacking potentially useful timepoints. Figure 3b demonstrates how overloaded timepoints are added to the timepoint sets. Suppose a task u has all of its units dispatched at time t ($\sum_{v \in N_G^+(u)} y(u, v, t) = \rho(u)$). We use this criteria to identify times of high unit utilization for task u. We then add a timepoint for task u at the earliest time that we can guarantee that the units will be available again, that is after the maximum return time of u at time t, $t + \omega^{max}(u, t)$ (labeled as t^* in Figure 3b).

Algorithm 2 Get Overloaded Timepoints

function GET OVERLOADED TIMEPOINTS(G, G^*, ε, x)
  $\varepsilon^+ \leftarrow \{\varepsilon^+(u) = \emptyset : u \in V\}$ ▷ Instantiate set of timepoints to add
  for all $u \in V$ **do**
    for all $t \in \varepsilon(u)$ **do**
      if $\sum_{v \in N_G^+(u)} y(u, v, t) = \rho(u)$ **then** ▷ Solution is dispatching all units of task u at time t
        $t' \leftarrow n(u, t+1)$
        while $t + \omega^{max}(u, t) < t'$ **do** ▷ Current time is between t and proceeding timepoint
          $t \leftarrow t + \omega^{max}(u, t)$
          $\varepsilon^+(u) \leftarrow \varepsilon^+(u) \cup \{t\}$ ▷ Set t to be added to $\varepsilon(u)$
  return ε^+ ▷ Return set of overloaded timepoints to add

We continue to repeat adding timepoints spaced by $\omega^{max}(u, t)$ until we reach the timepoint proceeding t, $n(u, t+1)$ (labeled as t' in Figure 3b). Note that this heuristic will only affect timepoints where there is extra time between when a u-unit will return to be used again, and the next timepoint when dispatching may occur at $(t + \omega^{max}(u, t) < n(u, t+1))$. We carry out this procedure over all vertices (u, t) of G^* such that all u-units are dispatched at time t.

We call the timepoints added by this heuristic "overloaded timepoints" because they are added in cases where we identify a fully utilized resource. This heuristic aims to reduce cases where there is a backlog of samples waiting at a task, but the task's units are underutilized because u has insufficient timepoints around time t. In these cases, we add timepoints for task u so that$u$-units may be used once they return from a neighboring task and become available again, thereby allowing better unit utilization for this task.

Algorithm Get Dominated Timepoints describes how timepoints are removed in cases when we have two adjacent timepoints that are sufficiently close. Figure 4 shows how we identify dominated timepoints to remove from the timepoint sets. Consider a task u that has two adjacent timepoints, t and $n(u, t+1)$, such that no u-units are dispatched at time $n(u, t+1)$ $(\sum_{v \in N_G^+(u)} y(u, v, n(u, t+1)) = 0)$. Suppose that the timepoints are also close enough that a unit dispatched at time t may not be dispatched at time $n(u, t+1)$, that

Algorithm 3 Get Dominated Timepoints

function GET DOMINATED TIMEPOINTS(G, G^*, ε, x)
  $\varepsilon^- \leftarrow \{\varepsilon^-(u) = \emptyset : u \in V\}$ ▷ Instantiate set of timepoints to remove
  for all $u \in V$ **do**
    for all $t \in \varepsilon(u)$ **do**
      if $\sum_{v \in N_G^+(u)} y(u, v, n(u, t+1)) = 0$ **then** ▷ No samples leaving task u at time $n(u, t+1)$
        if $n(u, t+1) - t < \omega^{min}(u, t)$ **then** ▷ We cannot allocate the same unit to both timepoints
          if $t + \tau(u, v, t) \leq n(u, t+1) + \tau(u, v, n(u, t+1))\ \forall v \in N_G^+(u)$ **then** ▷ Samples leaving at time t arrives before samples leaving at time $n(u, t+1)$
            if $\sum_{(w,t') \in N_{G^*}^-((u,n(u,t+1)))} y(w, u, t') = 0$ **then** ▷ No samples arrive after t and at or before $n(u, t+1)$
              $\varepsilon^-(u) \leftarrow \varepsilon^-(u) \cup \{n(u, t+1)\}$ ▷ Set $n(u, t+1)$ to be removed from $\varepsilon(u)$
  return ε^- ▷ Return set of dominated timepoints to remove

is the time difference between the points $(n(u, t+1) - t)$ is less than the minimum return time of u at time t, $\omega^{min}(u, t)$. Finally, if there are also no flows on arcs whose head is $(u, n(u, t+1))$ $(\sum_{(w,t') \in N_{G^*}^-((u,n(u,t+1)))} y(w, u, t') = 0)$, and samples that leave task u at time t arrive at their destination before samples which leave u at time $n(u, t+1)$ $(t + \tau(u, v, t) \leq n(u, t+1) + \tau(u, v, n(u, t+1))\ \forall v \in N_G^+(u))$, then we remove $n(u, t+1)$ from task u's timepoints. Note that these criteria checks are done using the previous value of ε, before any of the added timepoints are actually added. Therefore, we do not directly undo any of the added timepoints before they have undergone at least one iteration of the algorithm. However, it is possible that a timepoint added in iteration k of the algorithm may be removed during iteration $k+1$.

We call the timepoints which are removed according to the previous criteria "dominated timepoints". Based on the assumption that backward time shifting a solution cannot worsen its objective value, then any solution which uses u-units at time $n(u, t+1)$ could be changed to an equal or better solution which uses the same units at time t. Identifying and removing these timepoints helps to reduce the number of timepoints in the model, and hence the model size with the aim of improving the CPU cost of solving the model.

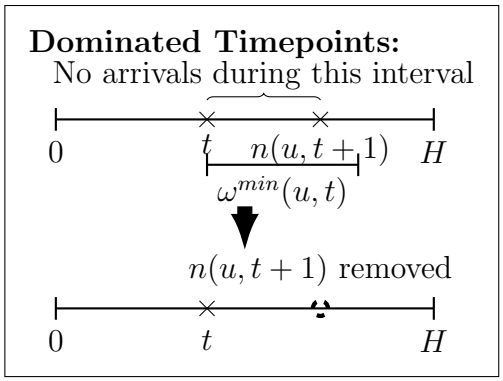


Figure 4: A figure demonstrating under what conditions we mark timepoints as dominated, for removal. Time flows along axis, with "X" representing the presence of a timepoint in $\varepsilon(u)$.

The proposed framework described above has several limitations. Given that this is a heuristic approach, global optimality is not guaranteed; thus, we accept practical solutions that may be sub-optimal but can be resolved in short CPU times. The algorithms Get Instant Start Timepoints, Get Overloaded Timepoints, and Get Dominated Timepoints used for modifying the timepoints between iterations have no guarantees, even though the ideas behind their development seems suitable and intuitive. With respect to the convergence of the framework, we did not observe any cycling (timepoints being added and subsequently removed many times) in our testing, but we do not guarantee that cycling may not occur. However, if such cycling is possible, the various stopping criteria discussed above will still ensure that the framework terminates.

Additionally, there are a number of parameters whose values must be chosen a priori such as the starting discretization to use for the initial problem (`start_disc`), the values for the stopping criteria (`its_tl`, `obj_thresh`, and `sols_tl`), and the parameters relating to solving ($\widehat{P}$) (`fin_disc` and `fin_tl`). The choices for these parameters will have an influence on the performance of the framework, and setting these parameters will depend on the specific problem being solved and one's solving preferences (CPU time limitations, hardware limitations, required solution quality, etc.). We present the actual choices for the parameters used in our experiments in section 4.1.

4 Computational Experiments

Up to this point, we have described the problem as a scheduling problem over a time layered graph. We used this more general setting to emphasize that this method is not solely applicable to our case study and may be used for other applications such as fermentation or other chemical production scheduling[23], railroad[40,41] or truck routing[22], or job shop problems[6]. In this section, we will introduce two case studies. The first involves a large-scale facility in the scientific services sector, with the aim of presenting the key features of our framework including analysing the performance and behaviour of our approach. The second case study presented in section 4.6 is used to demonstrate the application of our framework to other applications in chemical engineering. In this second case study, we also compare the performance of our framework against a previously reported time-discretization in the literature.

The scientific services sector is focused on carrying out analyses on samples that are ordered by clients for various purposes, e.g. performing a nutritional analysis on a food item to create the nutritional facts panel before bringing the product to market, or performing air quality analyses to check for hazardous materials such as asbestos. Companies in the scientific services sector may receive on the order of thousands of samples on a daily basis to be processed at their facility and as such require a suitable and efficient method of scheduling operations.

In particular, we have the following differences compared to the general description of the framework above. We have a single return time for u-units used, independent of what time the units are utilized and to which neighboring task they are dispatched to, $\omega(u, v, t) = \omega(u)\ \forall v \in N^+(u), t \in \varepsilon(u)$. We have a single travel time from task u to any other neighboring task v, again independent of what neighbor the samples travel to and at what time, $\tau(u, v, t) = \tau(u)\ \forall v \in N^+(u), t \in \varepsilon(u)$. Furthermore, the u-units become available immediately after the samples have been moved from task u to task v, i.e. $\tau(u) = \omega(u)\ \forall u \in V$.

The plant used for our experiments is based on a multipurpose industrial-scale scientific services facility. Due to confidentiality agreements, we cannot disclose detailed data. The facility is rather large with nearly 200 distinct tasks, each of which may have multiple identical machines. During a thirty day timespan, the facility received orders comprising of over 150 unique paths, using approximately 100 unique tasks. Over this timespan, they received several hundred orders comprising of more than 20,000 samples. The capacities and processing times of the individual tasks vary greatly. The largest capacity among all tasks is over 1,300 times the size of the smallest capacity, similarly the processing times of the tasks vary from a few minutes to several days. In the supplementary material, we present normalized values for capacity, processing time, and number of units for each task considered in the facility.

4.1 Policies Tested

Let us begin by defining the non uniform discrete M (NUDM) and uniform discrete (UDM) discretizations. The UDM discretization is defined as having a timepoint at times $S, S + M, S + 2M, \ldots, S + \lfloor H/M \rfloor M$ for each task, where S is the start time of the horizon and H is the length of the horizon. The NUDM discretization uses the *minimum* of M and the travel time of a task as the timestep for each task's time grid, where we assume that the travel time of a task does not depend on the destination. The NUD60 discretization (M = 60 minutes) was shown in [9] to have a better tradeoff between schedule performance and CPU solving time than the other uniform discretizations that were tested. Therefore we will refer back to this discretization several times throughout this section.

The iterative policies that were tested in this work are presented in table 2. The policies are named according to the following format: "max seconds between solutions" - "min objective improvement between iterations" - "starting discretization used". Therefore, for the "60 - 1.05 - UD240" policy, we begin with a uniform discrete time grid with 240 units space between each timepoint. During each iteration of the framework described in section

Table 2: Descriptions of iterative policies tested.

Policy Name	Starting Discretization	Time Limit for Iterative Phase (s) (`its_tl`)	Min. Objective Change (`obj_thresh`)	Max. Time Between Solutions (s) (`sols_tl`)	Time Limit for Final Solve (s) (`fin_tl`)
5 - 0 - UD60	UD60	600	1	5	600
5 - 0 - UD120	UD120	600	1	5	600
5 - 0 - UD240	UD240	600	1	5	600
60 - 1.05 - UD240	UD240	600	1.05	60	600

3, we will only allow up to 60 seconds between solutions before proceeding to the next iteration. If an iteration does not produce more than a 5% improvement in objective value compared to the output of the last iteration, then we exit. These parameter values were chosen experimentally, as the resulting policies performed best in our testing. We test each of UD60, UD120, and UD240 as starting discretizations (`start_disc`) with no minimum percent improvement between iterations (`obj_thresh` = 1) and a five second time limit between solutions during a single iteration (`sols_tl` = 5 seconds). By testing several different UD starting discretizations, we may observe how this choice affects the performance of the algorithm. These choices represent a policy which favors doing many iterations without spending too much time on each one. Note that the tuning of some of these parameters may need to be tuned by hand for different applications. For instance, if one wanted to increase the amount of time spent searching for better solutions before modifying the time grid and proceeding to the next iteration, then `sols_tl` should be raised. Alternatively, if one wanted to decrease the number of iterations that are performed, one could increase the `obj_thresh` parameter to force the process to stop as soon as the per iteration improvement starts decreasing too much. We also test a policy which starts with the UD240 discretization but allows sixty seconds between solutions (`sols_tl` = 60 seconds), and requires a minimum objective value improvement of 5% between iterations (`obj_thresh` = 1.05). This set of parameters represents a policy which is willing to spend more time searching for solutions during each iteration, but also may not perform as many iterations if the amount of improve-

ment slows. All iterative policies were given a 10 minute time limit for both the iterative phase (`its_tl` = 10 minutes) and the final solve after adding the known good discretization (`fin_tl` = 10 minutes). The NUD60 discretization was chosen as the set of timepoints to add after the iterative process ends (`fin_disc` = NUD60).

These iterative policies were compared against using the following discretizations without any sort of modifications: UD60, UD120, UD240, NUD60. We will refer to the use of these discretizations without modification as *static discretizations* throughout this section. These discretizations were chosen because they provide a wide range of granularity in the timepoint sets and so they should allow us to observe the tradeoff between computational expense and solution quality between the different discretizations. Additionally, the NUD60 discretization was shown to perform best among these other static discretizations in [9], and so we use it as a benchmark for comparing the iterative policies' performance.

4.2 Testing Procedure

Regarding the objective function of (P), f_{G^*}, we use $f(i,u,t) = 1 \; \forall i \in I, u \in \Pi(i), t \in \varepsilon(u)$, and $c(u,v,t) = 0 \; \forall u \in V, v \in N^+(u), t \in \varepsilon(u)$. These choices for f and c were used so that we may maximize the total amount of samples that is being sent through the network assuming that the costs of utilizing units are negligible. To evaluate the efficacy of the framework described previously in section 3 we use the following procedure. For each policy that is being tested, we record the maximum objective value that the policy has obtained after 1 minute, 5 minutes, 15 minutes, 30 minutes, and 60 minutes have elapsed. We will refer to these elapsed times as "checkpoints". To normalize the results between instances, for each policy and checkpoint, we compare the objective value at that checkpoint as a percentage of the maximum objective value any policy obtains over the entire procedure. We chose to report performance indicators at various intervals of time so that conclusions could be made from a practical standpoint, based on the amount of time that could be allotted to creating a schedule. That is, we assume that there is some external constraint which requires we have

a schedule after a fixed amount of time. If we are given as much time as needed, we may as well use a continuous time formulation with arbitrarily many event points to ensure that we obtain an optimal solution. The times indicated above were chosen for comparison purposes based on a wide range of solving requirements and based on the observed solving times of our problems. Additionally, we also recorded the amount of time taken for each policy to complete the procedure and report this value as a proportion of the maximum time taken over all policies. Our intention is to present the results so that we can discuss which method performs best under different solving time constraints.

We sort our results into three categories based on size. We measure instance size in terms of number of variables and constraints present in the model created when using the NUD60 static discretization. Instances were categorized into small, medium, and large size problems. Instances with less than 1,000,000 variables were considered to be small, instances with at least 2,000,000 variables were considered to be large, and other instances were considered medium sized. Note that all of the variables in our model are integer variables, and hence these numbers (1,000,000 and 2,000,000) refer to the total number of (integer) variables in the instances. More information about the sizes of the instances in each category is shown in table 3. Columns 2 and 3 present the minimum number of variables and constraints among all instances in each category. Similarly, columns 4 and 5 present the maximum number of variables and constraints among all instances. Column 6 indicates how many instances were sorted into each size category. Note that the number of instances sorted into each category varies from 10 to 34. The reason for this is because the instances were first formulated based on realistic horizon lengths and order arrivals using data from the industrial partner. The instances were not sorted based on size until the results were analyzed, at which point this discrepancy was observed.

Each instance of the problem was generated according to two parameters: the length of the horizon that we are solving over, and the number of samples that start in the facility. The horizon length we solve over varies from one day to seven days. The number of samples

considered in each instance varies between 5,000 and 30,000. The paths of the orders were sampled according to the observed frequencies of paths received at the plant over a thirty day span. Each instance was solved assuming that the facility was able to operate constantly over the scheduling horizon to simplify testing, that is we assume that there is no downtime. Detailed information about each instance including model size, relative objective values at each checkpoint, and which size category each instance was sorted into is provided in the supplementary material. It is worth clarifying that we classify the instances' size based on the number of variables and constraints because tweaking the horizon length and number of samples in the facility change the size of the problem drastically. For example, we have one instance using a horizon length of one day and considering 30,000 samples that has approximately the same number of variables and constraints as an instance using a horizon length of five days that considers 5,000 samples.

Table 3: Sizes of instances in each category.

Size Category	Min. Variables	Min. Constraints	Max. Variables	Max. Constraints	Instances
Small	304,444	209,449	698,866	388,218	10
Medium	1,146,524	614,159	1,997,392	1,108,319	34
Large	2,016,628	1,117,913	3,399,906	1,880,512	16

Tests were run using 2 threads running at 3.2 GHz on a machine with access to 8 GB of RAM. The implementation was done using the Julia programming language (version 0.6.2), along with the Gurobi.jl (version 0.4.1) and JuMP.jl[42] (version 0.18.2) packages. Gurobi[43] (version 8.0.1) was used for solving the problems. The optimality tolerance given to Gurobi was set to stop solving when the best found solution was within 0.5% of the optimal solution. The rest of Gurobi's settings were left at their defaults.

4.3 Performance on Small Size Problems

The experiments described in this section consist of instances which were categorized as being small. Figure 5 depicts the results over the small instances. We immediately observe that

the values of the various policies are approximately equal at each checkpoint, so we focus on the 1 minute checkpoint. As expected among the fixed time discretizations, the finer discretizations obtain better objective values than the coarser discretizations. Interestingly, the dynamic discretizations are able to match the performance of the NUD60 policy, However, the finish time results indicate that these dynamic policies take much more time to complete than the NUD60 for these problem instances.

By examining the results for the dynamic timepoint policies, we observe that the performance obtained using any of the dynamic variants is approximately equal to that of the NUD60 after 1 minute has elapsed. This demonstrates that we may find better solutions by applying the framework to a starting discretization compared to doing no timepoint modifications. Furthermore, the performance of the dynamic timepoint policies increases, slightly surpassing the NUD60 policy, after 5 minutes have elapsed. However, the final cluster of bars depicting the amount of time required for each policy to finish demonstrates that on these instances, the dynamic policies take much longer to terminate compared to the static policies. This extra time may be attributed to the iterative process of adding timepoints and solving several models. From a tradeoff standpoint, the static NUD60 discretization offers nearly the best performance, at a fraction of the computational cost that the dynamic policies have.

This efficient tradeoff between objective value and solving time was presented in[9] and demonstrates that when instances are small, the NUD60 discretization is a good performer on the present scheduling problem. In these cases where the problem size is relatively small and the NUD60 policy may finish quickly, the dynamic framework appears to be unnecessary as it may improve the solution quality only marginally but may also incur a comparatively large computational cost.

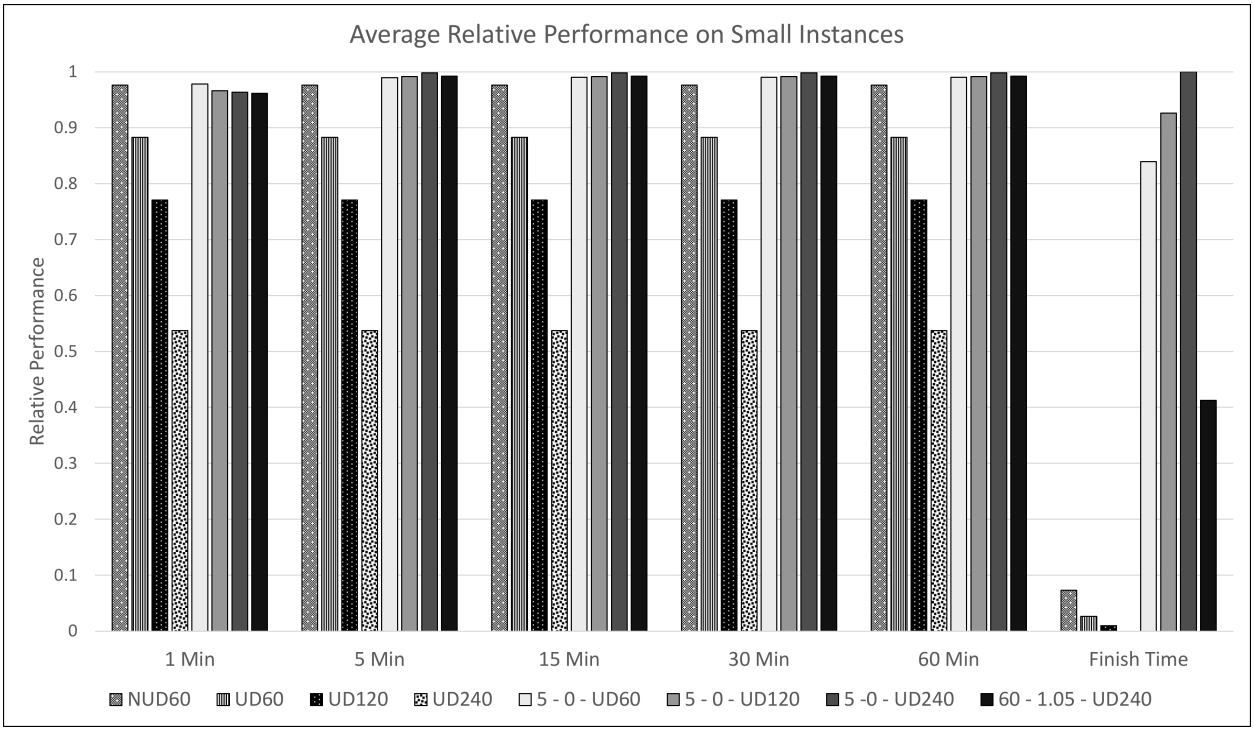


Figure 5: Results for small instances.

4.4 Performance on Medium Size Problems

The results on the set of medium sized instances presented by Figure 6 indicate a different story, however. Observing the performance after 1 minute, we no longer see the NUD60 discretization being the best performer. In fact, the UD240 and UD120 static discretizations both have performance better than the UD60 and NUD60 static policies after 1 minute. This can be attributed to the instances being larger, and hence the solver not being able to make as much progress as quickly, on the problems with more dense timepoint sets, compared to those with coarser timepoint sets. This demonstrates the weakness of using a fine discretization and the tradeoff between computational cost, solution quality, and discretization used. Furthermore, by comparing the coarser dynamic policies against their static counterparts, e.g. the 60 - 0 - UD240 dynamic policy and the UD240 static policy, after 1 minute has elapsed, we observe that even in cases where we desire solutions very quickly, and hence are forced to use coarse discretizations, there may still be a benefit to using an iterative approach. In this case, the 60 - 0 - UD240 dynamic policy is able to obtain approximately

77% of the greatest solution after 1 minute, while the UD240 obtains approximately 61% of the greatest solution on average. It is worth reiterating that we could stop our framework as needed to meet solving time constraints, as long as a solution has been obtained. Therefore, even given only one minute, there are cases where applying the framework may improve objective value.

We observe that all of the dynamic policies perform better than the static policies after 5 minutes have elapsed and that the NUD60 policy does not attain the same performance until 30 minutes have elapsed. Furthermore, at each checkpoint in time, a dynamic policy is performing best relative to the other static policies. This difference between the best performing static policy and the best performing dynamic policy ranges from 0% (at 30 or 60 minutes) - 20% (at 5 minutes). This demonstrates that for obtaining strictly the best performance, these policies were suitable on these medium sized problems. However, the same observation discussed above still applies. As the problems are larger, the policies which use a more fine initial discretization must solve larger problems and hence we see that the 5 - 0 - UD60 policy does not attain similar performance to the other dynamic policies until the 15 minute checkpoint. This observation highlights the need to pick the parameters used for the dynamic policies suitably depending on the problem specifications, e.g. if solutions must be attained within 1 minute, then using a more coarse starting discretization would be preferable to starting with the UD60 discretization. However, after 15 minutes, the 5 - 0 - UD60 policy performs best, so if it is acceptable to spend longer amounts of time to solve a particular problem, this policy may be preferable.

Most of the dynamic policies are still taking the longest to complete, and are taking longer than the NUD60 policy, but the amount of extra time spent is less significant than when the instances were small. Overall, over these instances, it appears that the 60 - 1.05 - UD240 policy is able to remain one of the best performers at each check point and also terminates sooner than the other dynamic policies and NUD60.

Depending on the user's preferences, on medium size instances it appears to be beneficial

to use a dynamic timepoint policy over a static discretization. However, this decision will be influenced by how large the problems to solve are, the amount of performance the user must obtain, and the time restrictions the problems must be solved within.

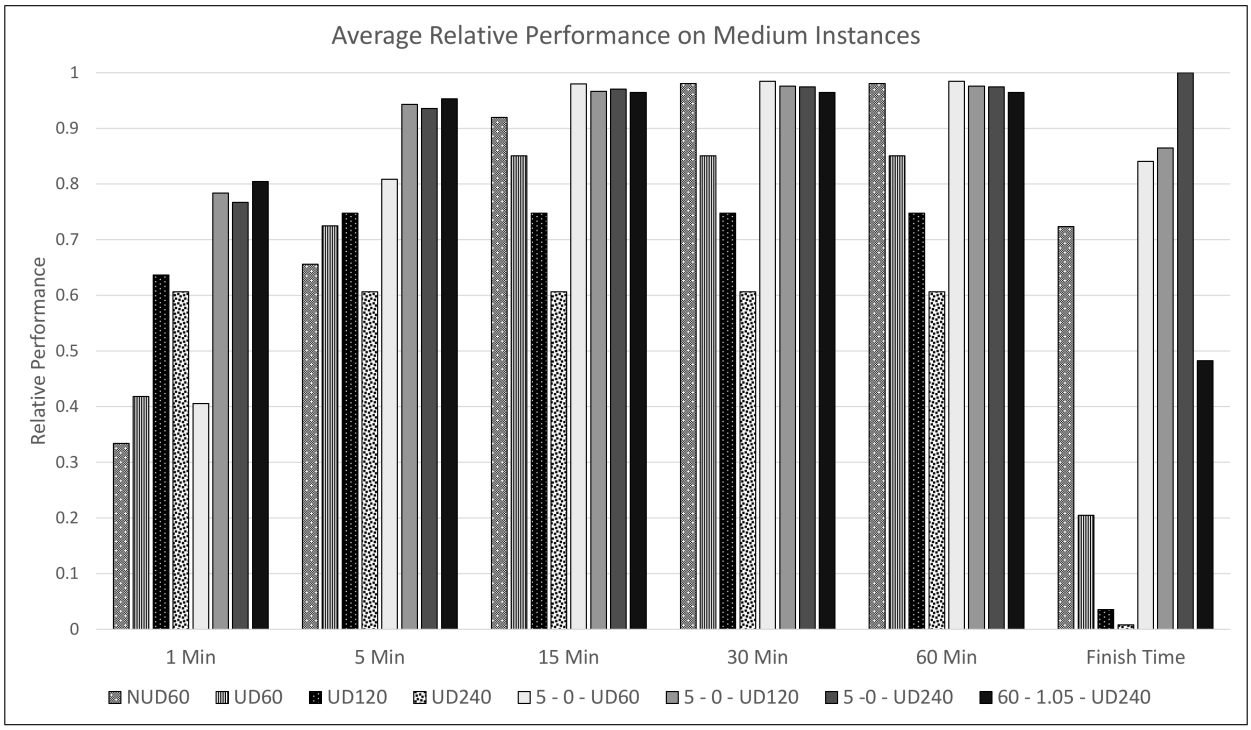


Figure 6: Results for medium instances.

4.5 Performance on Large Size Problems

Figure 7 shows the performance results on the large instances. It is at this point that we observe the breaking point of the NUD60 policy. On these instances, the NUD60 policy does not become competitive with the other policies until the 30 minute mark, and furthermore does not obtain a feasible solution within 1 minute on any of the large instances. The dynamic policies again offer good performance relative to the other policies at all checkpoints in time. The extra time cost associated with the dynamic policies compared to their fixed discretization counterparts is further reduced compared to the small and medium size instances. Over these instances, our iterative method offers a better alternative to the NUD60 discretization for users who require higher performance than a uniform discretization may

offer by either being equal to or exceeding the performance of the NUD60 policy at each checkpoint and by terminating in less than 30% of the time.

Furthermore, we note a benefit to increasing the amount of time to allow between solutions in these instances between the 60 - 1.05 - UD240 and the 5 - 0 - UD240 policies. In these instances, the extra time spent per iteration translates to an overall improvement in solution quality at each of the checkpoints, whereas this performance difference between these two policies was not the same over the smaller instances. In fact, over the smaller instances, we observed that by spending less time per iteration, but performing more iterations, the 5 - 0 - UD240 policy outperformed the 60 - 1.05 - UD240 policy. This observation highlights again the importance of the parameter selection discussed in section 4.1.

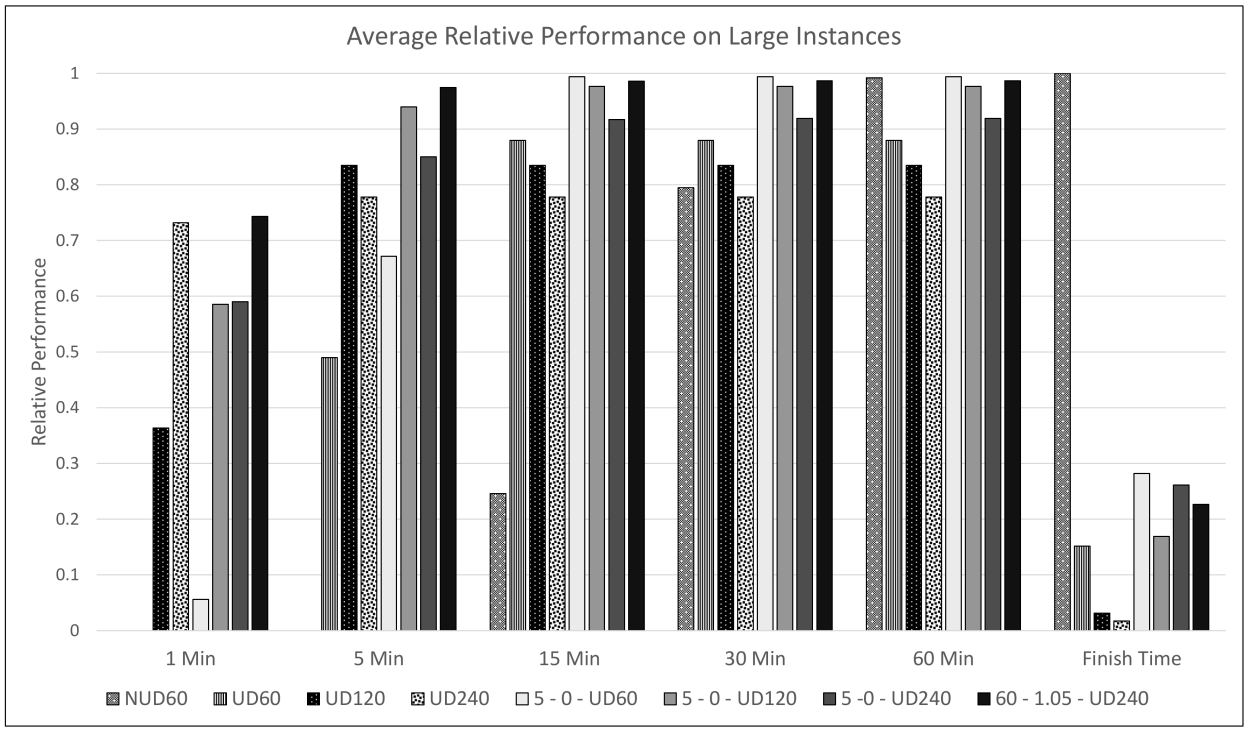


Figure 7: Results for large instances.

We also performed some experiments observing the performance of our framework, analyzed by iteration. This was not the focus of this work, so we include detailed discussion in the supporting information. The conclusions of this analysis were as follows. Using coarser discretizations lead to more iterations being done, however these extra iterations could be

limited by setting the `obj_thresh` to a value greater than one to ensure that we do not perform too many iterations with minimal improvement. Finer grids may actually decrease the total number of timepoints in early iterations because many of them are unnecessary, which supports the idea of beginning with a coarser discretization and adding only timepoints which seem promising. Moreover, the magnitude of the timepoint modifications between iterations drops off quickly after a few iterations, suggesting that setting other stopping criteria based on the number of timepoints added or removed may be beneficial to avoid continuing to iterate without much benefit.

4.6 Comparison to Previous Case Studies

In this section, we present some results comparing our method against that presented by Velez and Maravelias[23]. These experiments were done to compare the performance of our method of iteratively refining time grids against an existing alternate method of selecting timepoints statically. These experiments also demonstrate that our framework is not dependent on the model (P) presented in section 2.2, and may be applied to another general model. The approach of Velez and Maravelias[23] was proposed for use on process networks with varying time scales, such as for fermentation processes, or multi-site facilities. The application that Velez and Maravelias investigate[23] is a general batch scheduling problem. We test two process networks from their experiments. Process network 1 models a fermentation process where the time taken for different stages, (i.e. fermentation, purification, and processing) varies from a few days to a few minutes. Process network 2 models a process with different facilities whereby each facility is using its own time scale, and facilities 1 and 2 supply materials to facility 3. Note that Velez and Maravelias selected those networks because "they represent the types of process for which the multi-grid model can reduce the number of time points significantly while still finding the optimal solution", and so we also use them to compare against our model. Diagrams of Velez and Maravelias' process networks that were used for testing are included in the appendix as figure 9.

To compare these methods, two different experiments were constructed. Firstly, we compare the performance of our methods detailed in section 3 applied to the instances presented[23], using their model. The second set of experiments also used Velez and Maravelias' model, however the instances are from section 4.3 which have been transformed into instances that are compatible with Velez and Maravelias' model. Note that this can be done because Velez and Maravelias presented a generic batch scheduling problem and corresponding model. The model that was implemented and used for testing purposes is given in [23], pages 77-79. Any notation which is used in the context of Velez and Maravelias' model throughout this section is in reference to the notation used in [23]. To implement our framework in this model, we adapted the heuristics presented in section 3, to fit the context of this other model. The heuristics that were used for the following experiments are presented as algorithms 4, 5, and 6 and discussed in appendix A.

The implementation was performed using Julia, and the JuMP.jl and Gurobi.jl packages with the same versions as described in section 4.2. The instances were solved using Gurobi (version 8.0.1) with default values, on a 48 core machine running at 2.3GHz, with access to 256GB of RAM. We tracked the best objective value found over time, starting time at the moment that Gurobi begins its solve. That is, we do not include the time spent generating the models when reporting time values. A summary of the parameters that were used for the experiments we conducted is shown by table 4.

Table 4: Summary of parameters used for comparison experiments.

Figure	Network Description	Common Parameters				Our Parameters			VM Parameters
		Discretized By	Horizon Length (hours)	Time Limit (seconds)	Obj. Function Used	`start_disc` (hours)	`obj_thresh`	`sols_tl` (seconds)	(μ, ν) Tested
8a	VM Fermentation Process	hours	120	1,800	VM Max Profit	NUD5	1.01	30	(1,1), (2,2), (4,4)
8b	VM Multi-Site Process	hours	120	1,800	VM Max Profit	NUD5	1.01	30	(1,1), (2,2), (4,4)
8c	Small Instance #1	minutes	6	21,600	Max f_{G^*}	NUD1	1.05	60	(1,1), (2,2), (4,4)
8d	Small Instance #2	minutes	6	21,600	Max f_{G^*}	NUD1	1.05	60	(1,1), (2,2), (4,4)

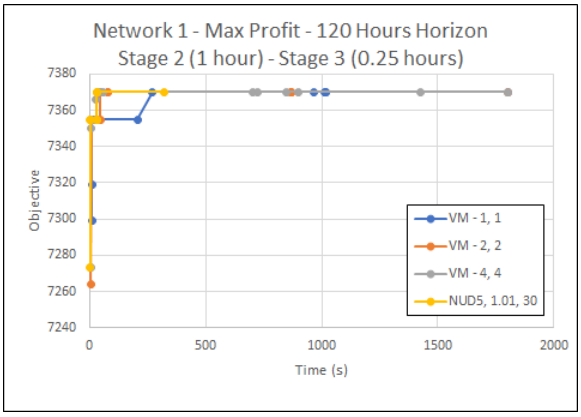
The results of these experiments are presented in Figure 8. Figures 8a and 8b present the results comparing both methods when solving the problem instances from [23]. Based on the results from [23], we used their instances with the smallest processing times and used their profit maximization function as our objective function. We used these instances as we expected them to be the most difficult instances to solve, since none of the instances using profit maximization as their objective were solved to optimality in [23], and we expect that decreasing processing times should increase the space of feasible schedules. We tested the performance of Velez and Maravelias' method using parameters: $(\mu = 1, \nu = 1)$, $(\mu = 2, \nu = 2)$, and $(\mu = 4, \nu = 4)$, represented by the data points "VM - 1, 1", "VM - 2, 2", and "VM - 4, 4" respectively. The μ, and ν parameters are used by Velez and Maravelias to introduce approximations into the time grids generated by their algorithms. μ determines "how much the step size can vary between tasks in the same unit", and ν determines "how much the step size can increase moving downstream". As these parameters are increased, the approximation allowed in the time grids increases. We choose these values for μ and ν as they were used by Velez and Maravelias[23] and provided various tradeoffs between time and objective quality. For our method we used parameters `start_disc` = NUD5 (hours), `obj_thresh` = 1.01, and `sols_tl` = 30 seconds, with no `fin_disc` and corresponding final solve. A 30 minute time limit was imposed on these problems. The graphs presented by figures 8a and 8b show that at any given time while solving these instances, the best objective value found using each method are approximately equal. This is particularly evident in figure 8b where the graphs of all four methods are almost completely overlapping. Whereas in figure 8a there is an exception as VM - 1, 1 takes approximately 200 more seconds to improve its objective value from 7,355 to 7,370, compared to the other methods. These results demonstrate that our method may perform similarly to that of Velez and Maravelias over these instances.

Figures 8c and 8d present the results when solving some of the small instances we previously discussed in section 4.3. We used the same objective function as discussed in section 4.2, and a 6 hour horizon. We tested these instances using the same parameters for Velez and

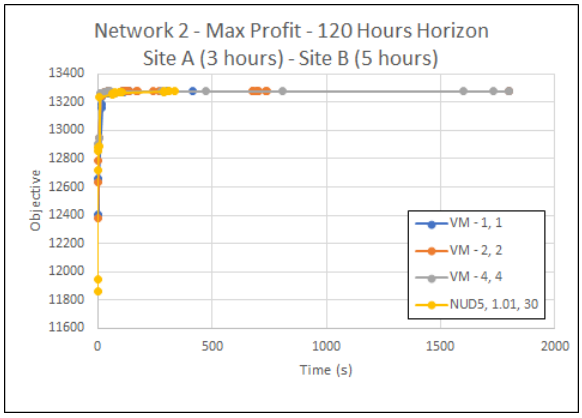
Maravelias' method as before, however we set the time limit passed to Gurobi to be 21,600 seconds (6 hours). For our method we used parameters `start_disc` = NUD60 minutes, `obj_thresh` = 1.05, and `sols_tl` = 60 seconds. Similarly to the previous experiment, we do not add a final discretization `fin_disc`. After we reach an initial stopping criteria during the iterative improvement phase of our method, we cease the procedure and return with the best objective found. The results presented by figures 8c and 8d show that when the size of the input data grows large, the models generated by Velez and Maravelias' method can grow to significantly large sizes, and therefore the time to solve these models can also increase greatly. This increase in solving time can be attributed to the large number of timepoints that are added by Velez and Maravelias' method to ensure that the optimal solution is not cutoff. Note that in figure 8c, our method achieves the greatest objective (11,387) after 647 seconds, which is only matched by the VM - 4, 4 approximation after 21,400 seconds. The VM - 1, 1 method achieves its greatest objective of 10,977 after 20,907 seconds and the VM - 2, 2 method achieves 11,134 after 20266 seconds. The results are slightly different in figure 8d. In this experiment, our method finishes with an objective of 11,545 after 509 seconds, which is surpassed by VM - 4, 4 after 3,458 seconds (objective 11,598), and finishes with an objective of 11,722 found after 20,935 seconds. Similarly to before, VM - 1, 1 achieves 11,248 after 17,484 seconds and VM - 2, 2 achieves 11,399 after 17,881 seconds. These two methods are unable to achieve the same objective values during the duration of our experiment because of the increased problem size, and therefore increased computational requirements. Even though our method does not obtain the greatest objective value in this experiment, it is still able to find an objective which is within 2% of the greatest objective found and it finds this objective in less than 15% the time taken to find a better objective by the VM - 4, 4 method. Note that this disparity in computation times is compounded when the time spent to generate the problems is considered. Our implementation of Velez and Maravelias' algorithms required a considerable amount of time to be spent simply selecting which timepoints to include in the model, however it may be possible to improve the speed of these

algorithms through more sophisticated implementations, therefore as mentioned before, we do not consider these times in our experiments.

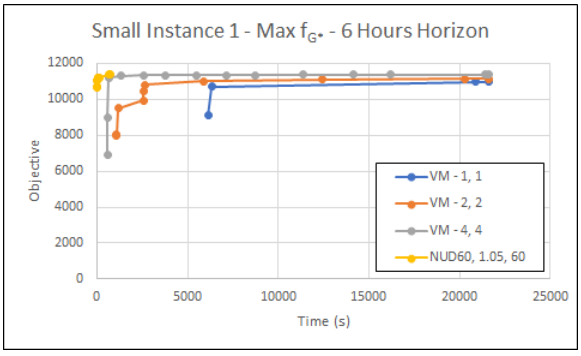
These results highlight the advantage of our approach of beginning with a coarse discretization and refining it compared to that of generating a fine initial discretization. When instances are small, we are able to perform roughly as well as when using Velez and Maravelias' method. However as instance size increases, our method is able to start reaping benefits. Comparing our method against that of Velez and Maravelias, we observe that their method generates problems which become computationally expensive in some cases, while ours continues to find good, but not necessarily optimal solutions in comparatively short timespans.



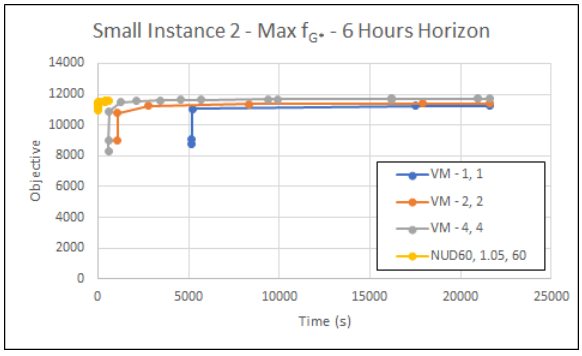
(a) Objective value over time on instance using Process Network 1[23].



(b) Objective value over time on instance using Process Network 2[23].



(c) Objective value over time on small instance 1, converted to be compatible with Velez and Maravelias' model.



(d) Objective value over time on small instance 2, converted to be compatible with Velez and Maravelias' model.

Figure 8: Performance comparison against Velez and Maravelias' method[23].

5 Conclusion

This work presents a practical framework for iteratively refining time grids for the scheduling of a multipurpose facility. The general framework was presented along with a discussion on the motivating ideas used to choose new timepoints. Computational experiments were conducted which showed that by refining the time grids of our scheduling problem with the proposed framework, we were able to improve performance over the conventionally used discretizations without substantially increasing the computational cost.

We observed the impact that the parameter selection for the algorithm can have on the results. One should use their requirements to guide the selection of the parameters for the algorithm, e.g. if short solving times are required, then beginning with a more coarse starting discretization may be advisable. However, if longer solve times are acceptable, then increasing the granularity of the starting discretization or increasing the duration of `sols_tl` may improve solution quality. In general, we would advise operators to begin with a coarser discretization so that an initial, sub-optimal solution can be obtained quickly. Then, the framework can be used to add only those timepoints which the heuristics suggest may improve objective value. The goal of this approach is to improve the objective value to be comparable with a fine discretization, while saving on the computational cost of obtaining the final solution by solving smaller problems.

The performance of the iterative policies stayed relatively stable across various problem sizes, but with improved tradeoff as problems became large. Moreover, analyzing the performance of the framework over each iteration showed that most of the performance improvements and timepoint modifications occur in the first few iterations. The results of these experiments show that using a refinement strategy can improve solution quality over the conventionally used uniform discretizations even when very short solving times are required. It is also worth noting that the proposed method is a general strategy which may be applied to other scheduling applications with similar requirements, bypassing the need to experimentally determine efficient time grids on a per problem basis. It is worth emphasizing

that the algorithms presented are heuristics and have no guarantee about performing well for all applications, even though they seemed to perform well in our case studies. Overall, we expect them to be more well suited to applications for which it is computationally difficult to solve the resulting problem when using a fine uniform discretization.

Regarding future work, there are a number of ways the framework presented may be augmented. Investigations into selecting a more specialized initial time grid could help the iterative method converge even more quickly. Additional heuristics for choosing which timepoints to add or remove during the method could be proposed and their efficacy tested. This could be especially useful when the performance differences between iterations become small to further improve performance. Finally, more research into how to fine tune the framework parameters, particularly for different problem applications, would be a worthwhile endeavor. In this work the values for these parameters were chosen experimentally, but an algorithmic method of selecting these values would be helpful when applying this method to other applications or facilities.

Acknowledgement

The authors thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Ontario Centers for Excellence (OCE), and the industrial partner in the analytical services sector for the financial support provided.

Nomenclature

Graph Symbols

V A set of nodes in a graph with no time component

A A set of arcs between nodes in a graph with no time component

G A graph $G = (V, A)$ with no time component

V^* A set of nodes in a time layered graph

A^* A set of arcs between nodes in a time layered graph

G^* A time layered graph $G^* = (V^*, A^*)$

u, v, w A particular node of V

$\varepsilon(u)$ A set of integer timepoints for a node $u \in V$

$n(u, t)$ The timepoint $t' \in \varepsilon(u)$, proceeding $t \in \varepsilon(u)$

$\tau(u, v, t)$ Time taken to go from node u to v at time t

$\delta_G^+(q)$ Set of arcs in G leaving node q

$\delta_G^-(q)$ Set of arcs in G entering node q

$N_G^+(q)$ Set of nodes in G that can reach q by using a single arc

$N_G^-(q)$ Set of nodes in G that can be reached by q by using a single arc

A_L^* Set of arcs in G^* that go between distinct nodes in V

A_H^* Set of arcs in G^* that start and end at the same node in V

Problem Symbols

I Set of orders to be scheduled

i A particular order in I

$\Pi(i)$ The path of an order i indicating the sequence of processes that must be performed on i

$\alpha(i, u, t)$ The number of samples of order i that arrive at task $u \in \Pi(i)$ at time $t \in \varepsilon(u)$

u-unit A unit that can carry out task u

$\rho(u)$ The number of units that carry out task u

$\kappa(u)$ The capacity of each u-unit

$\omega(u, v, t)$ The amount of time needed before a u-unit which is used at time t to bring samples to task v, can be used again

$H(i, u, t)$ The head of the arc in G^* corresponding to samples of order i at task u being sent to the subsequent task in its path at time t

Model Symbols

$y(u, v, t)$ The number of u-units dispatched to task v at time t

$x(i, u, t)$ The number of samples of order i that begin processing by a u-unit at time t

$z(i, u, t)$ The number of samples of order i that do not begin processing by a u-unit at time t

$f(i, u, t)$ The per sample weight of processing samples of order i on task u at time t

$c(u, v, t)$ The per unit cost of dispatching u-units to task v at time t

Timepoint Symbols

UDM, $M \in \mathbb{N}$ The uniform discretization such that timepoints are spaced by M units of time

NUDM, $M \in \mathbb{N}$ The non uniform discretization such that timepoints for task u are spaced by the minimum of M, and the processing time of u, $\tau(u)$

Supporting Information Available

As supporting information, we include further results and analysis on the performance of our framework with respect to each iteration. We also present normalized process parameter

values for the case study using the scientific services facility, and precise performance results for each of the instances tested in sections 4.3 - 4.5. This information is available free of charge via the Internet at http://pubs.acs.org/.

References

(1) Steinrücke, M. Integrated production, distribution and scheduling in the aluminium industry: a continuous-time MILP model and decomposition method. *International Journal of Production Research* **2015**, *53*, 5912–5930.

(2) Stablein, T.; Aoki, K. Planning and scheduling in the automotive industry: A comparison of industrial practice at German and Japanese makers. *Int. J. Production Economics* **2015**, *162*, 258–272.

(3) van Elzakker, M. A. H.; Zondervan, E.; Raikar, N. B.; Grossmann, I. E.; Bongers, P. M. M. Scheduling in the FMCG Industry: An Industrial Case Study. *Ind. Eng. Chem. Res.* **2012**, *51*, 7800–7815.

(4) Froger, A.; Gendreau, M.; Mendoza, J. E.; Pinson, E.; Rousseau, L.-M. Maintenance scheduling in the electricity industry: A literature review. *European Journal of Operational Research* **2016**, *251*, 695–706.

(5) Paraskevopoulos, D. C.; Laporte, G.; Repoussis, P. P.; Tarantilis, C. D. Resource constrained routing and scheduling: Review and research prospects. *European Journal of Operational Research* **2017**, *263*, 737–754.

(6) Chaudhry, I. A.; Khan, A. A. A research survey: review of flexible job shop scheduling techniques. *International Transactions in Operational Research* **2016**, *23*, 551–591.

(7) Pinedo, M. L. *Scheduling: Theory, Algorithms, and Systems*, 4th ed.; Springer: New York, 2012; pp 183–220.

(8) Lee, H.; Maravelias, C. T. Combining the advantages of discrete- and continuous-time scheduling models: Part 1. Framework and mathematical formulations. *Computers and Chemical Engineering* **2017**, *116*, 176–190.

(9) Lagzi, S.; Yeon Lee, D.; Fukasawa, R.; Ricardez-Sandoval, L. A. A Computational Study of Continuous and Discrete Time Formulations for a Class of Short-Term Scheduling Problems for Multipurpose Plants. *Ind. Eng. Chem. Res.* **2017**, *56*, 8940–8953.

(10) Velez, S.; Maravelias, C. T. Theoretical framework for formulating MIP scheduling models with multiple and non-uniform discrete-time grids. *Computers and Chemical Engineering* **2015**, *72*, 233–254.

(11) Floudas, C.; Lin, X. Continuous-time versus discrete-time approaches for scheduling of chemical processes: a review. *Computers and Chemical Engineering* **2004**, *28*, 2109–2129.

(12) Sundaramoorthy, A.; Maravelias, C. T. Computational Study of Network-Based Mixed-Integer Programming Approaches for Chemical Production Scheduling. *Ind. Eng. Chem. Res.* **2011**, *50*, 5023–5040.

(13) Mockus, L.; Reklaitis, G. V. Continuous Time Representation Approach to Batch and Continuous Process Scheduling. 1. MINLP Formulation. *Ind. Eng. Chem. Res.* **1999**, *38*, 197–203.

(14) Schilling, G.; Pantelides, C. C. A simple continuous-time process scheduling formulation and a novel solution algorithm. *Computers and Chemical Engineering* **1996**, *20*, S1221–S1226.

(15) Zhang, X.; Sargent, R. W. H. The optimal operation of mixed production facilities—a general formulation and some approaches for the solution. *Computers and Chemical Engineering* **1996**, *20*, 897–904.

(16) Lagzi, S.; Fukasawa, R.; Ricardez-Sandoval, L. A multitasking continuous time formulation for short-term scheduling of operations in multipurpose plants. *Computers and Chemical Engineering* **2017**, *97*, 135–146.

(17) Kondili, E.; Pantelides, C. C.; Sargent, R. W. H. A general algorithm for short-term scheduling of batch operations—I. MILP formulation. *Computers and Chemical Engineering* **1993**, *17*, 211–227.

(18) Shah, N.; Pantelides, C. C.; Sargent, R. W. H. A general algorithm for short-term scheduling of batch operations—II. Computational issues. *Computers and Chemical Engineering* **1993**, *17*, 229–244.

(19) Merchan, A. F.; Lee, H.; Maravelias, C. T. Discrete-time mixed-integer programming models and solution methods for production scheduling in multistage facilities. *Computers and Chemical Engineering* **2016**, *94*, 387–410.

(20) Rakovitis, N.; Zhang, N.; Li, J. A novel unit-specific event-based formulation for short-term scheduling of multitasking processes in scientific service facilities. *Computers and Chemical Engineering* **2020**, *133*.

(21) Dash, S.; Günlük, O.; Lodi, A.; Tramontani, A. A Time Bucket Formulation for the Traveling Salesman Problem with Time Windows. *INFORMS Journal on Computing* **2012**, *24*, 132–147.

(22) Boland, N.; Hewitt, M.; Marshall, L.; Savelsbergh, M. The Continuous-Time Service Network Design Problem. *Operations Research* **2017**, *65*, 1303–1321.

(23) Velez, S.; Maravelias, C. T. Multiple and nonuniform time grids in discrete-time MIP models for chemical production scheduling. *Computers and Chemical Engineering* **2013**, *53*, 70–85.

(24) Crainic, T. G.; Hewitt, M.; Toulouse, M.; Vu, D. M. Service Network Design with Resource Constraints. *Transportation Science* **2016**, *50*, 1380–1393.

(25) Erera, A.; Hewitt, M.; Savelsbergh, M.; Zhang, Y. Improved Load Plan Design Through Integer Programming Based Local Search. *Transportation Science* **2013**, *47*, 412–427.

(26) Andersen, J.; Christiansen, M.; G., C. T.; Grønhaug, R. Branch and Price for Service Network Design with Asset Management Constraints. *Transportation Science* **2011**, *45*, 33–49.

(27) Rajeswaran, A.; Narisimhan, S.; S., N. A graph partitioning algorithm for leak detection in water distribution networks. *Computers and Chemical Engineering* **2018**, *108*, 11–23.

(28) Pulsipher, J. L.; Rios, D.; Zavala, V. M. A computational framework for quantifying and analyzing system flexibility. *Computers and Chemical Engineering* **2019**, *126*, 342–355.

(29) Lin, Y.-C.; Fan, L. T.; Shafie, S.; Bertók, B.; Friedler, F. Generation of light hydrocarbons through Fischer-Tropsch synthesis: Identification of potentially dominant catalytic pathways via the graph-theoretic method and energetic analysis. *Computers and Chemical Engineering* **2009**, *33*, 1182–1186.

(30) Lin, Y.-C.; Fan, L. T.; Shafie, S.; Bertók, B.; Friedler, F. Graph-theoretic approach to the catalytic pathway identification of methanol decomposition. *Computers and Chemical Engineering* **2010**, *34*, 821–824.

(31) Heckl, I.; Friedler, F.; Fan, L. T. Solution of separation-network synthesis problems by the P-graph methodology. *Computers and Chemical Engineering* **2010**, *34*, 700–706.

(32) Heckl, I.; Halász, L.; Szlama, A.; Cabezas, H.; Friedler, F. Process synthesis involving multi-period operations by the P-graph framework. *Computers and Chemical Engineering* **2015**, *83*, 157–164.

(33) Tang, W.; Allman, A.; Pourkargar, D. B.; Daoutidis, P. Optimal decomposition for distributed optimization in nonlinear model predictive control through community detection. *Computers and Chemical Engineering* **2018**, *111*, 43–54.

(34) Moharir, M.; Kang, L.; Daoutidis, P.; Almansoori, A. Graph representation and decomposition of ODE/hyperbolic PDE systems. *Computers and Chemical Engineering* **2017**, *106*, 532–543.

(35) Daoutidis, P.; Tang, W.; Jogwar, S. S. Decomposing complex plants for distributed control: Perspectives from network theory. *Computers and Chemical Engineering* **2018**, *114*, 43–51.

(36) Gupta, U.; Heo, S.; Bhan, A.; Doutidis, P. Time scale decomposition in complex reaction systems: A graph theoretic analysis. *Computers and Chemical Engineering* **2016**, *95*, 170–181.

(37) Friedler, F.; Tarján, K.; Huang, Y. W.; Fan, L. T. Graph-theoretic approach to process synthesis: axioms and theorems. *Chemical Engineering Science* **1992**, *47*, 1973–1988.

(38) Patil, B.; Fukasawa, R.; Ricardez-Sandoval, L. A. Scheduling of Operations in a Large-Scale Scientific Services Facility via Multicommodity Flow and an Optimization-Based Algorithm. *Ind. Eng. Chem. Res.* **2015**, *54*, 1628–1639.

(39) Lee, D.; Fukasawa, R.; Ricardez-Sandoval, L. Bi-objective short-term scheduling in a rolling horizon framework: a priori approaches with alternative operational objectives. *Computers & Operations Research* **2019**, *111*.

(40) Wang, Y.; Liao, Z.; Tang, T.; Ning, B. Train scheduling and circulation planning in urban rail transit lines. *Control Engineering Practice* **2017**, *61*, 112–123.

(41) Lange, J.; Werner, F. Approaches to modeling train scheduling problems as job-shop problems with blocking constraints. *Journal of Scheduling* **2018**, *21*, 191–207.

Algorithm 4 Get Instant Start Timepoints (VM)

function GET INSTANT START TIMEPOINTS
  $Y^+ \leftarrow \{Y^+(i,j) = \emptyset : i \in I, j \in J_i\}$ ▷ Instantiate set of timepoints to add
  for all $i \in I$ **do** ▷ Task i
    for all $j \in J_i$ **do** ▷ Unit j can perform i
      for all $n \in N_{ij}^{IJ}$ **do** ▷ Timepoints of unit j, task i
        if $X_{ijn} > 0$ **then** ▷ Samples processed at ε_n by unit j for task i
          $z \leftarrow \varepsilon_n + \tau_{ij}$ ▷ z is the finishing time of this operation
          for all $k \in K_i^+$ **do** ▷ k is material output from i
            for all $i' \in I_k^-$ **do** ▷ i' is task which uses k as input
              for all $j' \in J_{i'}$ **do** ▷ Unit j' can perform i'
                $n' \leftarrow Get\_Timepoint(N, z)$ ▷ Get timepoint associated with actual time z
                $Y^+(i',j') \leftarrow Y^+(i',j') \cup \{n'\}$ ▷ Add n' to timepoints of unit j', task i'
  return Y^+ ▷ Return set of instant start timepoints to add

(42) Dunning, I.; Huchette, J.; Lubin, M. JuMP: A Modeling Language for Mathematical Optimization. *SIAM Review* **2017**, *59*, 295–320.

(43) Gurobi Optimization, L. Gurobi Optimizer Reference Manual. 2018; http://www.gurobi.com.

A Adapted Heuristics for Velez and Maravelias' Model

Algorithms 4, 5, and 6 present the algorithms that were used for the experiments run in section 4.6. These heuristics were created to be analogous to those presented in section 3 for the model (P), presented in section 2.2.

Algorithm 4 is adapted from algorithm 1. It states that if we process materials in unit j of task i at time ε_n, then we find out when this operation will finish, call this time z. Any materials k which are produced from task i will be available at time z. For each of these materials k, we lookup any tasks i' that consume material k and corresponding units j' of i' and allow processing of the newly created material to begin in unit j' of task i' at time z.

Algorithm 5 Get Overloaded Timepoints (VM)

function GET OVERLOADED START TIMEPOINTS
  $Y^+ \leftarrow \{Y^+(i,j) = \emptyset : i \in I, j \in J_i\}$ ▷ Instantiate set of timepoints to add
  for all $i \in I$ **do** ▷ Task i
    for all $j \in J_i$ **do** ▷ Unit j can perform i
      for all $n \in N_{ij}^{IJ}$ **do** ▷ Timepoints of unit j, task i
        if $X_{ijn} > 0$ **then** ▷ Samples processed at ε_n by unit j of task i
          $mats = \sum_{k \in K_i^-} S_{kn}$ ▷ $mats$ is the total amount of material that can be consumed by task i that is available at time ε_n
          $cap = \sum_{j \in J_i} \beta_j^{max}$ ▷ cap is the sum of max capacities of all units that can process task i
          if $mats > capacity$ **then** ▷ Current inventory of material k exceeds the max processing capacity of task i
            $start \leftarrow \varepsilon_n$ ▷ Time associated with timepoint n
            $step \leftarrow \tau_{ij}$ ▷ Processing time of unit j doing task i
            $end \leftarrow$ actual time of subsequent timepoint of N_{ij}^{IJ}
            $current \leftarrow start + step$ ▷ Step ahead
            while $current < end$ **do**
              $n' \leftarrow Get\_Timepoint(N, current)$ ▷ Get timepoint associated with actual time $current$
              $Y^+(i,j) \leftarrow Y^+(i,j) \cup \{n'\}$ ▷ Add n' to timepoints of unit j, task i
              $current \leftarrow current + step$ ▷ Step ahead
  return Y^+ ▷ Return set of overloaded timepoints to add

Algorithm 5 is adapted from algorithm 2. If material is processed in unit j of task i at time ε_n, then we calculate the total amount of material in inventory that could be processed by task i at that time. This is an upper bound on the amount of demand of task i at this time. We compare this value against the total maximum capacity of task i, assuming all units j that could process i were used at the same time. If the possible demand exceeds the capacity, then we add timepoints spaced apart by τ_{ij}, until we reach the next timepoint that was already present.

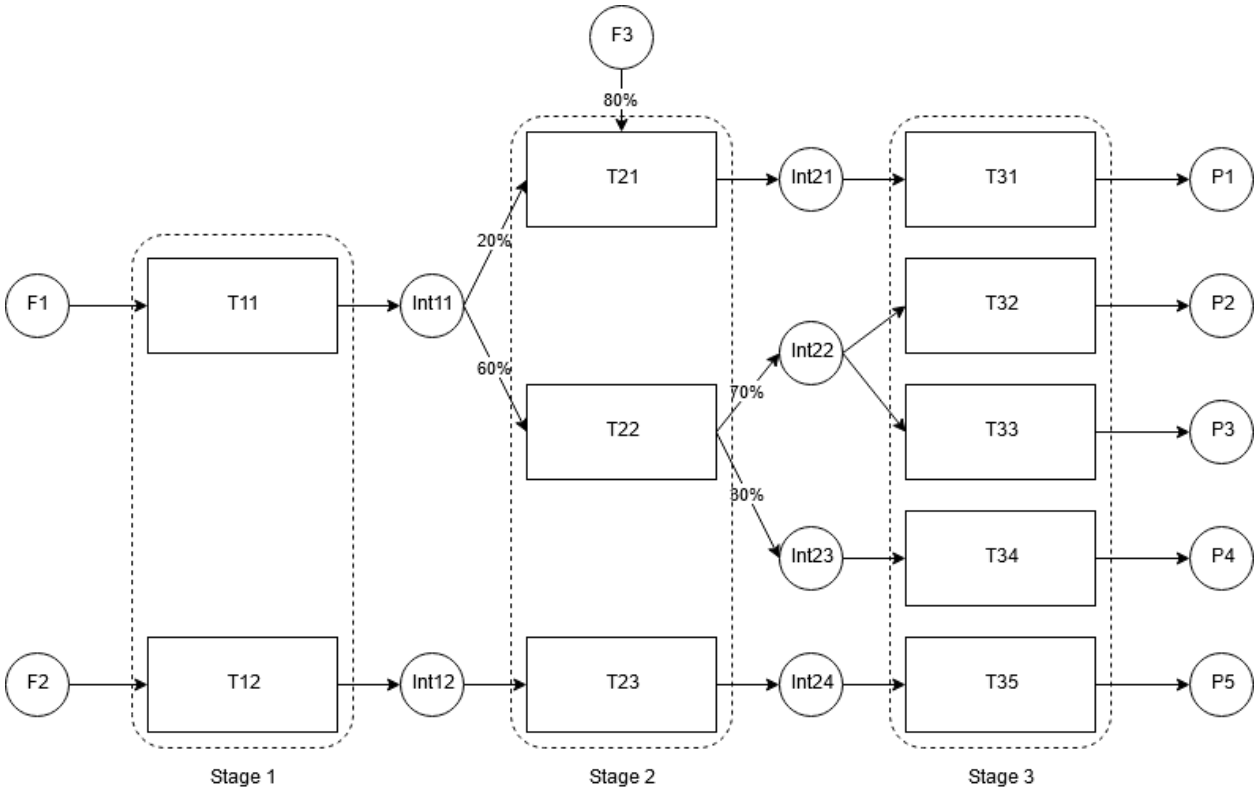
Algorithm 6 is adapted from algorithm 3. If unit j of task i is used at time ε_n, then we consider the subsequent timepoint for unit j processing task i. Call this subsequent timepoint n'. If the duration of time between n' and n is small enough such that only one of them can be used in a schedule, because of processing time constraints, then we remove n'

Algorithm 6 Get Dominated Timepoints (VM)

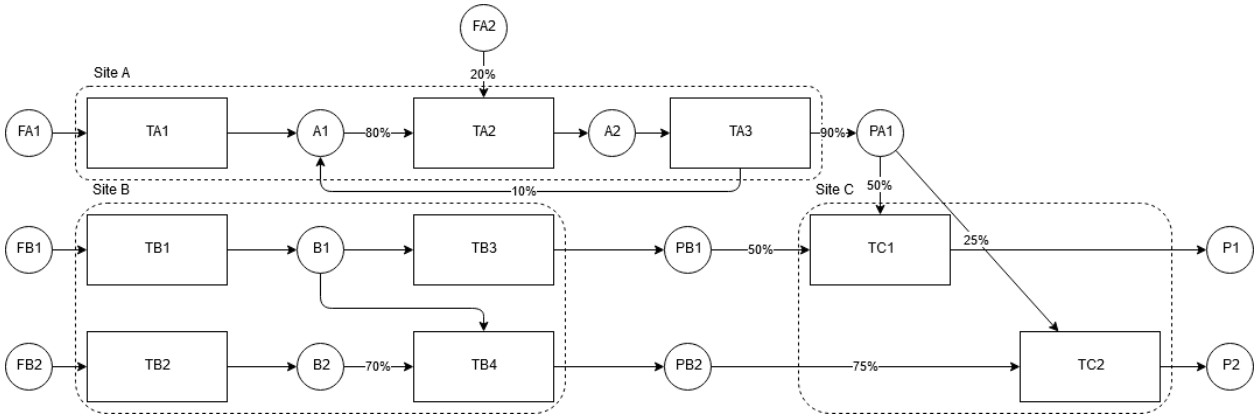
function GET DOMINATED TIMEPOINTS
  $Y^- \leftarrow \{Y^-(i,j) = \emptyset : i \in I, j \in J_i\}$ ▷ Instantiate set of timepoints to remove
  for all $i \in I$ **do** ▷ Task i
    for all $j \in J_i$ **do** ▷ Unit j can perform i
      for all $n \in N_{ij}^{IJ}$ **do** ▷ Timepoints of unit j, task i
        if $X_{ijn} > 0$ **then** ▷ Samples processed at ε_n by unit j of task i
          if (Subsequent timepoint $n' \in N_{ij}^{IJ}$ is such that $\varepsilon_{n'} - \varepsilon_n < \tau_{ij}$) **then** ▷ The next timepoint is close enough that not both n and n' could be used by unit j
            $Y^-(i,j) \leftarrow Y^-(i,j) \cup \{n'\}$ ▷ Remove n' from timepoints of unit j, task i
  return Y^- ▷ Return set of dominated timepoints to add

from the grid for task i, unit j. The reasoning is to remove timepoints that we do not need, if it is wanted again in the future, it may still be re-added by the instant start or overloaded timepoint heuristics.

B Process Networks



(a) Velez and Maravelias' Fermentation Process Network.



(b) Velez and Maravelias' Multi-Site Network.

Figure 9: Velez and Maravelias' Process Networks.

Graphical TOC Entry