BSc Thesis Applied Mathematics

On Multi-Stage Job Processing with Asymmetric Transition Costs between Operations

Puru Vaish

Supervisor: Alexander Skopalik, Ernst Moritz Hahn

July, 2022

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

The topic was of interest to me because of it's setting in the overlapping domain of applied mathematics and computer science. I believe, due to the existence of the studied phenomenon in the real world, this article is important since it would fill the mathematical and algorithmic gap in the understanding of such problems.

I would like to thank my supervisors for their consistent feedback and their availability for meetings. My understanding of the topic I cover in this article, the scientific writing approach the ability to create proofs for important results and most importantly the ability to realise what is important to a reader has been expanded, and due to our brainstorm sessions I understand better how researchers approach involved problems.

I would also like to thank every creator whose content I have consumed in the form of art, anime, manga and light novels in the three years of my study.

On Multi-Stage Job Processing with Asymmetric Transition Costs between Operations

Puru Vaish[*]

July, 2022

Abstract

In this article, we present a mathematically rigorous approach to a facility cost problem. Here, a facility is a production plant where there are multiple jobs, each consisting of multiple operations. Each of them might require one or multiple other operations to be completed before it. Also, when successive operations are performed on a single machine, there is a transition cost to performing another operation after another. Unlike other models, we do not assume transition costs to be symmetric, i.e., by switching the orders of operations a different cost might be incurred.

In this article we provide the result that our facility cost decision problem, which asks if there exists a schedule with cost less than a budget b, being NP-complete. We also show how, if given a polynomial time solver for the facility cost decision problem, we can also produce a polynomial time solver for the facility cost optimisation problem in polynomial time as well.

For a single machine facility cost problem, we provide the result that we can use any solver (approximate or exact) that exists for the Asymmetric Travelling Salesperson Problem ($ATSP$). We also investigate the effect of having when the single machine facility has λ-quasi-metric transition costs and single operation jobs, for which we provide an approximation algorithm which gives a schedule which has a cost at most $(1 + \lambda)$ away from the cost of the optimal schedule.

For the multiple machine facility cost problem, we provide an exact algorithm produced using dynamic programming approach and also show it could be interpreted as a mixed-integer linear program. This approach is extended to produce approximation algorithms using greedy rounding and a naive greedy algorithm, of which one can prove the approximation ratio to scale linearly with the total number of operations. Finally, we show how results derived for single machine facility cost problems could also be used for multiple machines facility cost problem by choosing specific parameters for the cost function.

Keywords: transition costs, cost optimisation, approximation algorithm, multiple operations, multi-stage jobs, dependency, quasi-metric cost optimisation, $ATSP$, production facility

[*]p.vaish@student.utwente.nl

Contents

1 Introduction

In our research, we focus on modelling and analysis of a production plant that we call a *facility*. Each facility incurs a running *cost* when they produce products, like electric bills, maintenance and also sustainability taxes like carbon emission taxes, pollution tax and so on. This results in the motivation to reduce these costs for a facility manager, and we call this problem the *facility cost problem*.

This problem was inspired by real-life manufacturing plants, such as ice cream and metal manufacturing. In such plants, there are often multiple types of products that are produced, for example, in the case of an ice cream production plant, vanilla, walnuts, and chocolate ice cream and tens of different variants, vegan, peanuts free, gluten-free, etc.

When making ice cream, for instance, there are many steps involved: boiling milk, using a blender to mix dry ingredients, pasteurising the ice cream mix with the milk in a pasteuriser followed by freezing and hardening the ice cream in a cooler. These steps are called *operations*. We can not pasteurise the milk without having both the milk boiled, and the dry ingredients blended, and we certainly can not freeze and harden the ice cream before completing the pasteurisation. These are called *dependencies* of the operation, as they require a set of operations to be complete before starting this operation. The milk boiler, blender, pasteuriser and the cooler are called *machines*. Making the ice cream itself is called a *job*. When making different types of these ice creams then, when we reuse the same machines, we incur some *transition costs*, in this example, this transition costs may be considered to be the cleaning costs. When we use the blender to mix the dry ingredients of an ice cream that has nuts (or other relevant allergen), we need to make sure there are no traces of this when we make a nuts free ice cream. Similarly, when we pasteurise two ice cream mixes which are both gluten-free and nuts free, we do not need to cleaning for removing the nuts or gluten since they were not present already, i.e., we have a smaller transition cost.

In production plants, the operations of each job (of which there can be many) have much more complicated dependencies and other transition costs between operations. Due to the complexity of these products, the managers of these production facilities have to rely on intuition and experience on how to reduce these costs by deciding on the order of the operations in which they will be performed on the machines. Furthermore, a facility might have too many jobs for a manager to effectively and quickly be able to use their knowledge to minimise the cost.

This research focuses on the problem of cost minimisation in such a production facility from a mathematical and algorithmic point of view. *Algorithms* are defined as steps or instructions to be followed in calculations or other problem-solving operations.

However, this facility cost optimisation problem can be modelled as a combinatorial optimisation problem. A combinatorial optimisation problem consists of finding an optimal solution from a finite set of possible solutions, where the set of feasible solutions is discrete or can be reduced to a discrete set. For the facility cost problem, this finite discrete set is all possible order of operations on each machine scheduled in a way that the dependencies are met. When done naively, optimising a combinatorial optimisation problem would involve having to try all possible elements of this finite discrete set which usually has an exponential number of possible solutions, making it infeasible to solve such a problem in a reasonable running time. Here, we can apply another branch of mathematics and computer science, namely, approximation algorithms. These algorithms produce a solution which has a cost (also called measure) which is a provable bound around the actual optimum cost of the optimal solution in a favourable running time. This is extremely useful and a powerful technique, since we do not only get a result which can be used, but also the maximal distance from the optimal our result is.

We proceed with the research assignment as follows. In Section 2 the theoretical background of algorithmic complexity is given, followed by the introduction to approximation algorithms. In Section 3 we describe a mathematically formal model that we will tackle along with the assumptions and the solution that we require to form the model (i.e. the solution that we want our algorithm to produce). In Section 4 we analyse the model and desired solution to obtain the computability complexity results. In Section 5 we first obtain results on the special case where a facility only has one machine. In this case, we provide the strong result on how any solver (exact or approximate) for the asymmetric travelling salesperson (*ATSP*) problem could be applied to solve the single machine facility cost problem. We also discuss the special case where we add the assumption that the transition costs are quasi-metric to achieve another approximation algorithm with a ratio which is a function of the facility instance. In Section 6 we then extend our consideration to multiple machines providing an exact dynamic algorithm, an equivalent Mixed Integer Linear Program (*MILP*) and a naive greedy algorithm. We also relax our *MILP* and consider a greedy rounding scheme and obtain results on the approximation ratio. Finally, we also consider more special cases for the multiple machine facility cost problem, depending on specific parameters for the cost function to minimise. We make the observation that we can apply our algorithms from the single machine facility cost problem, and how depending on the parameters of our cost function has an effect on how we are able to approach the problem.

We conclude our article with a discussion on our results and possible future research in Section 7 and Section 8 respectively.

Related Works

As noted earlier, there has not been much work done in a mathematical analysis of the production facilities with dependencies and asymmetric transition costs.

There have been a few studies with heuristic algorithms [1] but no proof of difficulty or bounds on how well the algorithm would perform is missing. They also only consider operations which only has one dependency.

Other models [2] which consider many dependencies, but it again falls short on having asymmetric transition cost as being part of the model.

Scheduling algorithms for just single machine [3], but when dependencies are added on multiple machines, these algorithms do not work.

There are also multiple algorithms [4] that exist, but they are often written for only two machines, where all operations can be completed in any of the machines. This again does not help our problem since we have multiple machines and each operation can be done on only one specific machine and the operations have dependencies with asymmetric transition costs.

2 Theoretical Background

To effectively present some results, we need to introduce some theoretical background knowledge on relevant combinatorial and decision problems and concepts of approximation algorithms.

2.1 Combinatorial and Decision Problems

A combinatorial optimisation problem can be defined in the following way [5].

Definition 1 (Combinatorial Optimisation Problem). *A* combinatorial optimisation problem, *$\mathcal{P}$ can be defined as a quadruple $\langle \boldsymbol{I}_{\mathcal{P}}, SOL_{\mathcal{P}}, m_{\mathcal{P}}, g_{\mathcal{P}} \rangle$ where*

- *$\boldsymbol{I}_{\mathcal{P}}$ is a set of instances,*
- *given an instance $x \in \boldsymbol{I}_{\mathcal{P}}$ $SOL_{\mathcal{P}}(x)$ is the finite set of feasible solutions of x,*
- *$m_{\mathcal{P}}$ is the measure function, defined for pairs (x, y) such that $x \in \boldsymbol{I}_{\mathcal{P}}$ and $y \in SOL_{\mathcal{P}}(x)$. For every pair (x, y), $m_{\mathcal{P}}(x, y)$ provides a value for the feasible solution y, and*
- *$g_{\mathcal{P}} \in \{\min, \max\}$ specifies whether $\mathcal{P}$ is a maximisation or minimisation problem.*

A decision problem can be defined in the following way.

Definition 2 (Decision Problem). *A decision problem is a problem that given an instance asks a question on this instance which has only two possible answers: YES or NO.*

If the instance answers the question with a *YES*, then the instance is called a *YES* instance of the decision problem and vice-versa for *NO*.

Decision problems can be constructed for our combinatorial optimisation problem in the following way. Given a combinatorial optimisation problem $\mathcal{P}$, we can define a *decision problem*, $\mathcal{P}_{\mathcal{D}}$, which asks, "Given an instance $x \in \boldsymbol{I}_{\mathcal{P}}$ and a budget b if there exists a solution $y \in \mathrm{SOL}_{\mathcal{P}}(x)$ such that $m_{\mathcal{P}}(x, y) < b$ (or $m_{\mathcal{P}}(x, y) > b$)?".

In most cases, we want to know both the optimal measure and the solution for which it is achieved. However, finding the optimal measure and the solution can be difficult. In the following, we will formalise the difficulty of a decision problem $\mathcal{P}_{\mathcal{D}}$.

P and *NP* Problems

Decision problems are well understood, and there exists a classification with which each of the problems can be understood. The two most relevant class of problems for our assignment are *P* and *NP* problems.

To define *P* and *NP* problems, we will first define polynomial-time computability and the notion of a polynomial time solver.

Definition 3 (Polynomial Time [6]). *An algorithm is said to be of* polynomial time *if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm. Then we say that the running time function $T : \mathbb{N} \to \mathbb{R}$, $T(n) \in O(n^m)$ where n is the size of the input, for some positive constant m.*

Definition 4 (Polynomial Time Solver). *Let a decision problem be given. If there is an algorithm $\mathcal{S}$, which can compute a solution for any instance of the decision problem in* *polynomial time in the size of the input instance, then $\mathcal{S}$ is called a* polynomial time solver *of the decision problem.*

Now, we define what it means for a decision problem to be in P [7].

Definition 5 (Class of Problems P)**.** *A decision problem $\mathcal{P}_{\mathcal{D}}$ is said to be in P if there exists a polynomial-time solver that solves any instance of the decision problem.*

For defining the class of problems *NP*, we need the notion of a certificate and polynomial time verifier.

Definition 6 (Certificate)**.** *A* certificate *for an instance of a decision problem, is a possible construction of a solution which can be used to answer the question posed by the decision problem.*

Definition 7 (Polynomial Time Verifier)**.** *If there exists a polynomial time algorithm $\mathcal{A}$ that takes the pair of the instance of the decision problem and a certificate for this instance, and outputs YES if the instance is a YES instance of the decision problem and NO if the instance is a NO instance, then $\mathcal{A}$ is called a* polynomial time verifier *of the decision problem.*

We then define the class of problems that is in *NP*.

Definition 8 (Class of problems *NP*)**.** *A decision problem is said to be in NP if there exists a polynomial time verifier for the decision problem.*

We know that all P problems are also a *NP* since the polynomial time solver of the problem can be used as the polynomial time verifier as well, by ignoring the certificate. However, it is not yet known whether, $P = NP$ and the common assumption is that this is not the case. Thus, it is more likely easier to verify a solution of a problem instance than it to generate such a solution, unless $P = NP$.

We also introduce the definition of *NP*-hard and *NP*-complete. For defining *NP*-hard problems, we need the notion of polynomial time reductions.

Definition 9 (Polynomial Time Reduction)**.** *If there is a function f, which can be computed in polynomial time, such that a decision problem $\mathcal{D}$ is translated to another decision problem, $\mathcal{D}^*$ such that an instance x is a YES instance of $\mathcal{D}$ if and only if $f(x)$ is a YES instance of $\mathcal{D}^*$ then $\mathcal{D}$ is a* polynomial time reduction *of $\mathcal{D}^*$. We denote this by $\mathcal{D} \preceq_P \mathcal{D}^*$.*

Now we can define the class of problems that is *NP*-hard.

Definition 10 (*NP*-hard)**.** *A decision problem, $\mathcal{D}$, is NP-hard if the for all problems that are in NP there exists a polynomial time reduction to $\mathcal{D}$.*

A problem being *NP*-hard essentially means that it at least as hard as any other problem which is in *NP*. Note that *NP*-hard problems may not have a polynomial-time verifier; therefore, a problem which is *NP*-hard may not be in *NP* itself.

The class of problems that is *NP*-complete can now be defined.

Definition 11 (*NP*-complete)**.** *A problem $\mathcal{P}$ is NP-complete if the problem is NP-hard and is also in NP.*

The utility of showing that a problem is *NP*-complete or only *NP*-hard difficult is that it provides information on the difficulty of the problem relative to other well-known problems. It also means that solving any problem that is *NP*-complete in polynomial time would mean that we can solve any problem that is also NP-complete in polynomial time using the fact that there exists a polynomial time reduction between the two problems.

2.2 Approximation Algorithms

As mentioned earlier, it is important to produce a solution for a problem instance and not only to verify that a solution is correct. At this moment, since it is not known whether $P = NP$ that is, there exist problems for which we have found polynomial-time verifiers but no polynomial-time solvers have been found.

So given a combinatorial optimisation problem $\mathcal{P}$, and its corresponding decision problem, $\mathcal{P}_{\mathcal{D}}$ we might be both unable to tell in a polynomial time if there exists a solution which is better and also be unable to construct that optimal solution in polynomial time.

Because of this, it is often useful to apply approximation algorithms. An approximation algorithm can be defined as follows [8]:

Definition 12 (α-approximation)**.** *An α-approximation algorithm for an optimisation problem is an algorithm such that for an instance of the problem, it produces a solution whose value is within a factor α of the optimal value.*

For many approximation algorithms, we represent α to be a function of the instance itself, because a constant approximation ratio is not possible unless $P = NP$.

3 Model

To understand our problem at hand, we first need to define a model through which we can study the difficulties that arise and also compare and contrast between existing models and results.

A natural candidate for creating a model for problems reminiscent of scheduling, algorithmics are models that employ the use of graphs [9]. Directed graphs are a very natural way to visualise and understand dependencies because they provide the intuitive restriction of having to visit a vertex before we can visit other vertices, since those vertices are only accessible through another vertex.

Because of the abundant use of graph-like modelling methods for scheduling, cost optimisation and algorithmic problems, it also allows us to adapt and use existing algorithms for completely new problems in a mathematically rigorous approach.

First, we give a glossary of commonly used words in scheduling and cost-optimisation problems

1. Job: An activity that may constitute many operations to be completed for the job to be considered complete. For instance, making tea is a job.
2. Operation: The smallest task that cannot be broken down into smaller tasks. For instance, boiling water for the tea is one particular operation in the job of making tea.
3. Machine: An instrument through which a particular operation of a job can be completed. For example, an electric kettle is a machine.
4. Dependency: This relates two operations of a job, which implies that to start an operation we must complete/wait for the completion of another operation. For

example, filling the water kettle is dependent on boiling the water in the water kettle.

3.1 Model Definition

Let $F = \langle J, M, \boldsymbol{D}, \boldsymbol{c} \rangle$ be the instance of the facility cost problem. The instance is defined as follows:

- $J = \{1, \dots, n\}$. The jobs the facility has to process.
- $M = \{1, \dots, r\}$. The machines the facility has. In this model, a particular operation of a job will already be prescribed on a machine it will need to be processed on.

  Without loss of generality, the final operations of each of the jobs which have no processing time can be completed on a dummy machine, as those operations are only a marker of the job being completed.

  Each machine can only process one operation at a time.
- $\boldsymbol{D} = (D_j)_{j \in J}$. Each D_j represents the dependency graph of each job j for all its operations. These are represented as directed acyclic graphs (Appendix A) of the operations for each job $j \in J$. Each $D_j = (V_j, E_j)$ where:
  - $v \in V_j$ is the operation that is part of the job $j \in J$.
  - $(u, v) \in E_j \subseteq V_j \times V_j$ such that D_j is acyclic.
  - $M(v) \in M$ is the machine v needs to be performed on.
  - $T(v) \in \mathbb{N}$ is the processing time required to process an operation v independent of other operations done.
  - $\text{pred}_s\,(v) = \{u \mid (u, v) \in E_j\}$ defines the predecessors of an operation v. This set can be $\emptyset$.
  - Denote by $f^{(j)} \in V_j$ the operation such that $f^{(j)}$ is that this is the last operation for the completion of the job with zero processing time.

    Note: If no such operation exists, the job can be modified to add a terminating operation, $f^{(j)} = \hat{f}$ with $T(\hat{f}) = 0$ and $M(\hat{f})$ can be assigned to a dummy machine and adding the edges connecting the terminating operations with an out degree of 0 to this dummy operation $\hat{f}$ without changing anything about the job itself.
- For convenience, we define the following sets:

  $$V = \bigcup_{j \in J} V_j \qquad \text{and} \tag{1}$$

  $$\text{For all } m \in M \text{ we define } V|_m = \{v \in V \mid M(v) = m\} \tag{2}$$

  where V gives the set of all operations and $V|_m$ gives the set of all operations that need to be performed on the machine m.
- $\boldsymbol{c} = (c_m)_{m \in M}$ where each $c_m \colon V|_m \times V|_m \to \mathbb{N}$. For each operation $v \in V_j$ for a job $j \in J$, the extra time to perform an operation v due to the preceding operation u on the same machine is defined as $c_{M(v)}(v, u)$. This encapsulates extra time incurred

due to the last job and the actual cost for running the job v on the processor of the machine m.

For all operation v, we define $c_m(v, \emptyset) := 0$. When calculating the cost, we require this for an operation which has no preceding operations.

3.2 Desired Solution

The desired solution is the schedule, $s\colon V \to \mathbb{N}$. Essentially, the schedule is the map for each operation that maps each operation that needs to be performed to the time unit $t \in \mathbb{N}$ at which it needs to be performed.

For example, if we have the operations a on machine 1, b on machine 2 and c on machine 1 are to take place at time 0, 0 and 2 respectively, then $s(a) = 0$, $s(b) = 0$ and $s(c) = 2$.

3.3 Constraints and Assumptions

Constraints Now we can define the constraints on the facility cost problem.

Let a schedule s be given for a facility instance.

We define the following functions and conventions for convenience to talk about different aspects and events of the model:

- $\text{start}_s(v) : V \to \mathbb{N}$ be the time at which the operation v starts.
- $\text{end}_s(v) : V \to \mathbb{N}$ be the time at which the operation v ends.
- $\text{bef}_s(v) : V \to V$ be the operation that is scheduled before operation v on the machine $M(v)$. If there was no previous operation scheduled, then it returns $\emptyset$.
- $\text{ans}(v) = \bigcup_{u \in \text{pred}_s(v)} (\{u\} \cup \text{ans}(u))$, this is the set of all ancestors of v. Since each D_j is a directed and acyclic, we do not have the case where a node is its own ancestor.
- Define $\text{start}_s(\emptyset) = \text{end}_s(\emptyset) = 0$.

Then for the schedule to be valid, we have the following logical constraints due to dependencies on other operations:

1. All dependencies of the operations v must be completed first before v starts processing. This can be written as

$$\forall\, j \in J\ \forall\, v \in V_j\ \forall u \in \text{pred}_s(v)\ \ \text{start}_s(v) \geq \text{end}_s(u) \tag{3}$$

2. Before starting the job v on a machine m, the job scheduled before v needs to be completed. In our model, each machine can only do one operation at a time, therefore we only begin processing the next operation after the end of the previous operation. This can be written as

$$\forall m \in M\ \forall\, v \in V|_m\ \text{start}_s(v) \geq \text{end}_s(\text{bef}_s(v)) \tag{4}$$

Assumptions We analyse the model with these base assumptions.

1. A machine can only process one operation at any given time.
2. All processing times are deterministic.

3. Each job has a single starting operation and a single final operation. Formally, this means that the job has one starting operation and one ending operation, which can be the same operation as well. If this is not true, the original job can be modified by adding extra operations with transition from and to, respectively, with zero processing time on a dummy machine.

Note: We do not require $c_m(u, v) = c_m(v, u)$. That is, the cost may not necessarily be a symmetric function.

3.4 Objective Function

Finally, we define the calculation of the cost function that is to be minimised.

Maximum Makespan Formulation

- Finishing time of a job, $j \in J$ defined as $s(f^{(j)})$. This follows from the fact that each $f^{(j)}$ is the final operation of the job j with zero processing time.
- The cost function that we want minimise over can therefore be defined as

$$C(F, s) = \max_{j \in J} s(f^{(j)}) \tag{5}$$

  where F is the instance of a facility problem and s is the schedule that satisfies the constraints of the problem instance F. Since $f^{(j)}$ is the last operation of each job, which by definition has zero processing time on the dummy machine, each $s(f^{(j)})$ is the makespan of the job j and the cost is called the maximum makespan of all the jobs.

General Cost Formulation $C(F, s)$ in scheduling is called the maximum makespan objective function, that is the time at which all operations for all jobs have been completed. However, in some cases, we might want to consider adding different weights to each transition cost and processing time. For instance, when cleaning, a particular method of cleaning always done between two operations might be fast as opposed to when the order in which the operations are done is reversed. But, it may be expensive to undergo the former cleaning process, in which case the facility might want to avoid having that particular transition cost at all where possible, despite it being faster. The above function does not capture this. However, we can modify it to allow for such a case, and we can also show that the maximum makespan minimisation can be achieved from this formulation if required.

- Let $i, k \in V$.
  - Then, let $\boldsymbol{\omega}_{i,k} \in \mathbb{N}$ be the weights assigned to the extra time spent performing the operation k before i.
  - Let, $\alpha \in \mathbb{N}_0$ be given.

Then, we define the general cost function as

$$C(F, s; \alpha, \boldsymbol{\omega}) = \alpha \cdot \max_{j \in J} s\left(f^{(j)}\right) + \sum_{j \in J} \sum_{i \in V_j} \boldsymbol{\omega}_{i,\mathrm{bef}_s(i)} c_m\left(i, \mathrm{bef}_s\left(i\right)\right). \tag{6}$$

For the rest of this article we will use this cost function as the objective function.

The parameters α and $\boldsymbol{\omega}$ are not part of the model themselves, but are parameters solely for the calculation of the overall cost. This is a modelling decision since the objective function itself does not influence the model; hence, we choose to separate them

Notice that setting $\boldsymbol{\omega}_{i,k}$ to 1 for all $i, k \in V$ and setting α to 1 for all $j \in J$, we get exactly the maximum makespan cost function.

In practise, these weights $\boldsymbol{\omega}_{i,k}$ can be the cost per unit time of the specific transition going from operation k to i, set up that needs to be done and α can be the average cost to keep the facility open with no jobs running per unit time.

We could also take $\alpha = 0$ in this general cost function. This then disregards completely the makespan of each job, and is only concerned with minimising cost by considering the order of operations done on each machine, since effectively we no longer have the information of the "idle" times between successive operations. This may seem counterintuitive to consider as a cost function however, this is still useful in cases where the order of operations is more important than the makespan of the job itself or in case where the makespan cost hardly influences the total cost. We consider this special case in the article as well, in Section 6.5.

3.5 Lazy Schedules

We end the model discussion with a definition of a recurring notion.

Definition 13 (Lazy Schedule)**.** *A schedule s is called a* lazy schedule*, if there exists an operation in the schedule where the start time of the operation could be shifted to an earlier time without changing the order of the operations.*

Intuitively, a lazy schedule is the notion where an operation or a whole set of operations could simply be started earlier (shifted in time) such that the order of operations are the same while all constraints are met. When we consider the case where $\boldsymbol{\alpha} = \mathbf{0}$ we can end up with lazy schedules since the cost function will only optimise the order of operations and so if the operation is scheduled arbitrarily later than opposed to immediately after the last operation has the same cost, but from a production standpoint a lazy schedule is not desirable hence the final schedule an algorithm produces should never be a lazy schedule.

3.6 Model Validation

To validate our model, we looked at real-life manufacturing plants.

Transition Costs In many manufacturing plants, we see that there are often transition costs between two different operations that are performed on the same machine. For instance ice cream manufacturing plants where special flavours like vegan ice cream requires a different cleaning cost based on the ice cream produced before it on the same production machine; less if it was another vegan ice cream, more if the ice cream was not vegan. In our model we separate the time it takes to do the cleaning and the monetary cost of the cleaning itself by only modelling the notion of transition cost which denotes the time it takes to make the transition multiplied with the cost per unit time of the specific cleaning operation (as the parameter $\boldsymbol{\omega}$). This allows us to synchronise our scheduling as well, since any dependent process can only start after the end of all the predecessor operations and not after a particular cost was incurred.

Dependency Graphs Directed Acyclic Graphs were chosen to represent the hierarchy in the order the operations could be performed in time, since it was a natural way to model multiple dependencies of arbitrary-sized jobs.

The Objective Function The objective function is very involved, especially due to the parameter α. As explained before, α can be seen as the cost per unit of time to have the facility open at all without any operations. We also justify having the weights for each of $\boldsymbol{\omega}_{i,\mathrm{bef}_s(i)}$ for $c_m(i, \mathrm{bef}_s(i))$ term for every operation $i \in V_j$ for every job $j \in J$ as the cost per unit time of making the transition between two different operations. This can be the cost of the cleaning staff, cleaning materials, the wear it causes to the components undergoing the transition and so on.

The objective function could however be made more complicated by adding a weight for each job instead of only the makespan of the facility for all the jobs. This adds a sense of priority to the jobs. However, this further complicates the model and results we would like to achieve. It is to be noted that this decision does indeed affect the results we present, and so this is a non-trivial modelling decision.

4 Complexity of the Facility Cost Problem

Now that we have defined the model for each instance of the facility cost problem, we can define it as a combinatorial optimisation problem.

Definition 14 (Facility Cost Problem). *The facility cost problem is defined as the quadruple $\mathcal{F} := \langle I_{\mathcal{F}}, SOL_{\mathcal{F}}, m_{\mathcal{F}}, g_{\mathcal{F}} \rangle$ where*

- *$I_{\mathcal{F}}$ is the set of facilities as modelled in Section 3,*
- *$SOL_{\mathcal{F}}$ is the finite set of schedules which are not lazy schedules,*
- *$m_{\mathcal{F}}$ is the general cost function $C(F, s; \alpha, \boldsymbol{\omega})$ and*
- *$g_{\mathcal{F}} := \min$, i.e. the goal is to minimise the cost incurred by the facility.*

We also give here the corresponding decision problem, that we call the Facility Cost Decision Problem.

Definition 15 (Facility Cost Decision Problem). *Given the facility cost optimisation problem $\mathcal{F}$, the* Facility Cost Decision Problem *is the problem which asks, "Given an instance $F \in \boldsymbol{I}_{\mathcal{F}}$ and a budget b is there exists a schedule $s \in SOL_{\mathcal{F}}(F)$ such that $C(F, s; \alpha, \boldsymbol{\omega}) \leq b$?".*

Other than the decision problem, we can define two other variants that are of interest.

Definition 16 (Facility Cost Search Problem). *$\mathcal{F_S}$ the* Facility Cost Search Problem *of the facility cost problem $\mathcal{F}$ with the goal that given an instance of a facility $F \in \boldsymbol{I}_{\mathcal{F}}$ a budget b and parameters α and $\boldsymbol{\omega}$, give a solution $\hat{y} \in SOL_{\mathcal{F}}(F)$ such that $C(F, \hat{y}; \alpha, \boldsymbol{\omega}) < b$.*

Definition 17 (Facility Cost Optimisation Problem). *$\mathcal{F_{OPT}}$ the* Facility Cost Optimisation Problem *with the goal that, given an instance of a facility $F \in \boldsymbol{I}_{\mathcal{F}}$ and parameters α and $\boldsymbol{\omega}$, give the optimal solution y^* where $y^* := \arg\min_{\hat{y} \in SOL_{\mathcal{F}}(F)} C(F, \hat{y}; \alpha, \boldsymbol{\omega})$.*

Notice the Facility Cost Optimisation Problem explicitly requires the construction and giving the output of the optimal schedule rather than finding the minimum cost only.

4.1 Facility Cost Decision Problem is *NP*-complete

In this section, we show that $\mathcal{F}_{\mathcal{D}}$ is a *NP*-complete problem.

Theorem 1. *$\mathcal{F}_{\mathcal{D}}$ is NP-complete*

For showing that a problem is *NP*-complete, we must show that

1. the problem $\mathcal{F}_{\mathcal{D}}$ is in *NP* and
2. the problem $\mathcal{F}_{\mathcal{D}}$ is *NP*-hard.

The second step is usually shown by showing that another *NP*-complete problem can be reduced to the problem of interest $\mathcal{P}$, and then due to transitivity $\mathcal{P}$ is also *NP*-complete.

Claim 1. *$\mathcal{F}_{\mathcal{D}}$ is NP*

Proof. We have to show that $\mathcal{F}_{\mathcal{D}}$ has a polynomial time verifier, $\mathcal{A}$. Construct $\mathcal{A}$ in the following way:

Algorithm 1: Polynomial Time Verifier for $\mathcal{F}_{\mathcal{D}}$

```
function $\mathcal{A}(x, \hat{y})$ is
    $x \in \boldsymbol{I}_{\mathcal{F}}$;
    $\hat{y}$ is the certificate;
    Using Equation 6 calculate the Objective Function Value, which is the cost
      incurred by the facility
    Verify the constraints
       1. $\forall\ j \in J\ \forall\ v \in V_j\ \forall u \in \text{pred}_s\,(v)\ \text{start}_s\,(v) \geq \text{end}_s\,(u)$.
       2. $\forall m \in M\ \forall\ v \in V|_m\ \text{start}_s\,(v) \geq \text{end}_s\,(\text{bef}_s\,(v))$.
    if cost calculated is less than b and constraints are satisfied then
        result ⟵ YES;
    else
        result ⟵ NO;
    end
    return result
end
```

Each of the steps required only a polynomial number of computational operations with respected to the input, and hence this is a polynomial time verifier. This is seen below.

The number of additions to be done is in the order of the number of operations in the facility x in the instance, and the number of boolean statements to verify if constraints are also in the order of the number of operations in the facility x.

Hence, we have that $\mathcal{F}_{\mathcal{D}}$ has a polynomial time verifier $\mathcal{A}$ and therefore $\mathcal{F}_{\mathcal{D}} \in NP$. □

Claim 2. *$\mathcal{F}_{\mathcal{D}}$ is NP-hard.*

For this proof, we use the reduction of Directed Hamilton Path Problem [10] (D-HAM-PATH) to the Facility Cost Decision Problem $\mathcal{F}_{\mathcal{D}}$. D-HAM-PATH decision problem can be defined as follows:

Definition 18 (D-HAM-PATH)**.** *Given a complete weighted graph $G = (V, E)$ with edge cost given by $w_{(u,v)}$ for all $(u, v) \in E$ and budget b,* D-HAM-PATH *is the problem which asks "Is there a Directed Hamilton Path from vertices s to t such that the sum of edge weights is less than b?".*

A Directed Hamilton Path is one such that all vertices in a graph are visited exactly once.

D-HAM-PATH is a well known *NP*-complete problem, hence we only need to show D-HAM-PATH $\preceq_P \mathcal{F_D}$. See Appendix B for proof that D-HAM-PATH is *NP*-complete.

Proof. Given an instance $G \in$ D-HAM-PATH, construct an instance $F \in \mathcal{F_D}$ in the following way. Define a function ϕ where ϕ : D-HAM-PATH $\to \mathcal{F_D}$. Given an instance

Algorithm 2: Polynomial Time Reduction of D-HAM-PATH to $\mathcal{F_D}$

$G = (V, E)$ of D-HAM-PATH with n vertices, the edge weights for $(u, v) \in E$ as $w_{(u,v)}$ the starting and ending vertex s, t respectively, and the budget b.

1. Let $J = \{1\}$,
2. Let $M = \{1\}$, so there is only one machine.
3. For the job, 1 define the directed acyclic graph $D_1 = (V_1, E_1)$ with:
   (a) $V_j = \{s, 2, \dots, n-1, t\}$ with $\forall u \in V_j$ $u_T = 0$ and $u_M = 1$. So the jobs have no processing time and need to be performed on the machine 1. Operations s, t denote the starting and terminal nodes in the D-HAM-PATH problem.
   (b) $E_j = \{(s, 2), \dots (s, n-1), (2, t), \dots, (n-1, t)\}$. These edges denote that the dependencies that exist in the D-HAM-PATH that s is the starting node and so in the facility problem that translates to dependency that s has to be done before all other operations $2, \dots, n-1$ and similarly that the operation t (the terminal node in D-HAM-PATH instance) can only be done after all the other operations $2, \dots, n-1$ are complete.
4. We also need to give the cost function vector $\boldsymbol{c} = (c_1)$. Here we give $c_1(u, v) = w_{(u,v)}$, therefore, the cost function for each pair of operations is just the edge weight in instance G of D-HAM-PATH. For the cost function, we use the weights $\boldsymbol{\omega}_{i,k} = 1$ and $\alpha = 0$.

To show *YES* instance of D-HAM-PATH, G if and only if $\phi(G)$ is a *YES* instance of $\mathcal{F_D}$.

Firstly, *YES* instance of D-HAM-PATH, G implies $\phi(G)$ is a *YES* instance of $\mathcal{F_D}$.

Since G is a *YES* instance of D-HAM-PATH, there exists a path from s to t such that the sum of the weight of the edges on this path is less than b that visits each vertex exactly once. The $f(G) \in \mathcal{F_D}$ is also a *YES* instance since by performing the operations in the same order as the vertices are visited in the path in instance G will also have a cost less than b since the cost function will be exactly the edge weights in this path while also satisfying the constraints of the schedule where s is the starting operation and t is the terminal operation.

Secondly, *YES* instance of D-HAM-PATH, G is implied by $\phi(G)$ is a *YES* instance of $\mathcal{F_D}$

Since in $f(G)$ we have that s is a dependency of each other vertex $2, \dots, n-1$ and since t has the dependency of all the nodes $2, \dots, n-1$ it can only be performed at the end of all other operations hence the facility will always operate t last, i.e. visit t last. Following from the fact that the transition cost function $c_1 : V|_1 \times V|_1 \to \mathbb{N}$ is defined to be exactly the edge weights of the instance graph G. Since there is only one processor in the machine 1, all operations have to be done sequentially. The cost function $C(F, s)$ is exactly the sum of cost of doing the operations sequentially where all operations are executed exactly once, therefore we have that if $f(G)$ is a *YES* instance of $\mathcal{F}_{\mathcal{D}}$, G is also a yes instance of D-HAM-PATH.

Hence, we have shown that D-HAM-PATH $\preceq_P \mathcal{F}_{\mathcal{D}}$ and since D-HAM-PATH is *NP*-complete, we have shown that $\mathcal{F}_{\mathcal{D}}$ is *NP*-hard. □

Hence, we can finally give the proof of the theorem that Facility Cost Decision Problem is *NP*-complete.

Proof. Since Facility Cost Decision Problem, $\mathcal{F}_{\mathcal{D}}$, is in *NP* and is *NP*-hard, by definition $\mathcal{F}_{\mathcal{D}}$ is *NP*-complete. □

4.2 Oracles

In this section we provide results about the complexity of the Facility Cost Search Problem, $\mathcal{F}_{\mathcal{S}}$ and the Facility Cost Optimisation Problem, $\mathcal{F}_{\mathcal{OPT}}$ through the use of oracles.

We begin our discussion starting with the definition of an oracle.

Definition 19 (Oracle)**.** *An* oracle *is a black box algorithm, which may not be computable, which can solve a decision problem in a single call.*

The purpose of the oracle is not to provide results of the decision problem, but to be able to conclude something about the construction problems using the decision problem. For this we usually assume we have an oracle of the decision problem and then from it, we make our conclusions on a related construction problem. Since by definition oracles need not be something that is computable, we can enforce that we have an oracle that allows us to solve a decision problem which is *NP*-complete in polynomial time without needing to proof if $P = NP$. Hence, in this section we assume we have an oracle that solves the Facility Cost Decision Problem, $\mathcal{F}_{\mathcal{D}}$.

To be able to use the oracle later, we need the following result.

Lemma 1. *Given an instance F of the facility cost problem $\mathcal{F}$, there is an upper bound $T \in \mathbb{N}$ for the cost of any non-lazy schedule s such that $C(F, s; \alpha, \boldsymbol{\omega}) \leq T$.*

Proof. Let

$$\begin{aligned}
\alpha_{\max} &:= \alpha, \\
\omega_{\max} &:= \max_{j \in J \; v \in V_j \; u \in V|_{M(v)}} \boldsymbol{\omega}_{v,u}, \\
c_{\max} &:= \max_{j \in J \; v \in V_j \; u \in V|_{M(v)}} c_{M(v)}(v, u), \\
T_{\max} &:= \max_{i \in V} T(i) \qquad\qquad \text{and} \\
K &:= \alpha_{\max} \left(T_{\max} + c_{\max}\right) + \omega_{\max} c_{\max}.
\end{aligned}$$

Then,

$$
\begin{aligned}
C(F, s; \alpha, \boldsymbol{\omega}) &= \alpha \cdot \max_{j \in J} s\left(f^{(j)}\right) + \sum_{j \in J} \sum_{i \in V_j} \boldsymbol{\omega}_{i, \mathrm{bef}_s(i)} c_m\left(i, \mathrm{bef}_s\left(i\right)\right) \\
&\leq \sum_{j \in J} \left[\alpha_{\max} s(f^{(j)}) + \sum_{i \in V_j} \omega_{\max} c_{\max} \right] \\
&\leq \sum_{j \in J} \sum_{i \in V_j} \alpha_{\max}\left(T(i) + c_{\max}\right) + \omega_{\max} c_{\max} \\
&\leq \sum_{j \in J} \sum_{i \in V_j} \alpha_{\max}\left(T_{\max} + c_{\max}\right) + \omega_{\max} c_{\max} \qquad (7) \\
&= \sum_{i \in V} K \\
&=: T
\end{aligned}
$$

where the second inequality follows from the fact that since we are now performing only one operation at a time on any machine, and the schedule is not a lazy schedule, so the term $s(f^{(j)})$ is bounded by the sum of the running time of all operations including the transition times to the next operation. □

Corollary 1. *Let an instance facility F with parameters α and $\boldsymbol{\omega}$ be given. Then the upper bound T on the cost of any non-lazy schedule is polynomial in the size in bits of the given instance F, α and $\boldsymbol{\omega}$.*

Proof. Denote with B the number of bits used to represent the instance facility F, α and $\boldsymbol{\omega}$. Each of $\log(\alpha)$, $\log(\boldsymbol{\omega}_{u,v})$, $\log(c_m(u, v))$ and $\log(T(u))$ are a polynomial number of bits of B itself, as each of those values are part of the instance itself.

Then it follows, $\log(\alpha_{\max})$, $\log(\omega_{\max})$, $\log(c_{\max})$ and $\log(T_{\max})$ are also polynomial in the number of bits B. Let K be as defined in Lemma 1 for the instance. $\log(K)$ is also polynomial in the number of bits B since this is a sum of a polynomial number of $\alpha_{\max}$, $\omega_{\max}$, $c_{\max}$ and $T_{\max}$. Due to Equation 7 we know the size of T, $\log(T)$ is polynomial in the size of K which is $\log(K)$ since we sum over the number of the total number of operations in the facility instance, $|V|$, which again has a size which is polynomial in the size of the instance F, B bits.

Hence, we have shown that $\log(T)$ is polynomial in the bits of $|V|K$ which is polynomial in the size of bits of the instance F, hence $\log(T)$ is polynomial in the size of the instance F. □

Having now established that there is an upper bound on the cost of non-lazy schedules we can use this Lemma 1 to show that given an oracle that can solve the Facility Cost Search Problem, $\mathcal{F}_{\mathcal{S}}$ in polynomial time, we can solve the Facility Cost Optimisation Problem, $\mathcal{F}_{\mathcal{OPT}}$ using a polynomial number of calls with respect to the size of the input, to this oracle, hence constructing a polynomial time algorithm.

We can also show that given an oracle that solves for the Facility Cost Decision Problem, $\mathcal{F}_{\mathcal{D}}$, in polynomial time, we can solve the Facility Cost Optimisation Problem, $\mathcal{F}_{\mathcal{OPT}}$, with polynomial number of calls with respect to the size of the input, to the oracle, hence we can construct a polynomial time algorithm.

We also use this lemma later to prove bounds of approximation algorithms in Section 6.

Theorem 2. *$\mathcal{F}_{\mathcal{S}}$ is in P implies $\mathcal{F}_{\mathcal{OPT}}$ is in P*

Proof. Let the weights $\boldsymbol{\omega}$ and α be given for the cost function. Let the oracle that solves $\mathcal{F}_{\mathcal{S}}$ in polynomial time be denoted by $\mathcal{S}$. For providing a polynomial time solver of, $\mathcal{F}_{\mathcal{OPT}}$ we will perform a polynomial number of calls with respect to the size of the input instance to $\mathcal{S}$, the solver of $\mathcal{F}_{\mathcal{S}}$. Let T be the upper bound for any non-lazy schedule for the instance $x \in I_{\mathcal{F}_{\mathcal{OPT}}}$. This bound is guaranteed to exist and to be finite due to Lemma 1. We know now that the cost of any non-lazy schedule is between 0 and T. We can use these two bounds to run a binary search routine. We first query $\mathcal{S}$ with $b = T/2$. If it answers *NO*, then we know that the minimum cost of any schedule must be between T and $T/2$. If it produces a schedule, then we know that the minimum cost of any schedule must be between 0 and $T/2$. We can continue the binary search in this recursive fashion until we have reached the resolution of the minimum cost and the corresponding schedule, and we would have only made $\log(T)$ calls to the oracle $\mathcal{S}$.

Finally, we know $\log(T)$ is polynomial in the size of the facility instance due to Corollary 1. Hence, we have constructed a polynomial time solver for $\mathcal{F}_{\mathcal{OPT}}$. □

Theorem 3. *$\mathcal{F}_{\mathcal{D}}$ is P implies $\mathcal{F}_{\mathcal{OPT}}$ is P.*

Proof. To show: If there is a polynomial time solver for $\mathcal{F}_{\mathcal{D}}$ then using a polynomial number of calls to the solver, we can solve $\mathcal{F}_{\mathcal{OPT}}$.

Let the oracle that solves $\mathcal{F}_{\mathcal{D}}$ in polynomial time be denoted by $\mathcal{S}_{\mathcal{D}}$. Then we follow the algorithm as given in Algorithm 3.

In the algorithm, the binary search step takes $\log(T)$ order of calls, where T is again as in Lemma 1 which is again polynomial in the size of the input instance due to Corollary 1, and the searching for the schedule takes $O(|V|^2)$ order of calls to the $\mathcal{S}_{\mathcal{D}}$ since in the worst case we need to iterate over all operations and each possible preceding operation.

Because $\mathcal{S}_{\mathcal{D}}$ is a polynomial time solver for the $\mathcal{F}_{\mathcal{D}}$ problem, we have found a polynomial time solver for $\mathcal{F}_{\mathcal{OPT}}$.

Algorithm 3: Polynomial Time Solver for $\mathcal{F}_{\mathcal{OPT}}$ given Polynomial Time solver for $\mathcal{F}_{\mathcal{D}}$

```
function polyTimeOptSolver(F, $\mathcal{S}_{\mathcal{D}}$) is
    /* $\mathcal{S}_{\mathcal{D}}$ is the polynomial time solver for $\mathcal{F}_{\mathcal{D}}$.    */
    By using binary search using $\mathcal{S}_{\mathcal{D}}$ we can find the minimum cost of any schedule.
      This process is the same as in the proof for Theorem 2;
    Denote this by $b^*$. ;
    Let $h^*$ denote the order of operations on every machine.;
    for $v \in V$ do
        for $u \in V|_{M(v)} \setminus ans(v)$ do
            $c_m(u, v) \longleftarrow \infty$;
            Query $\mathcal{S}_{\mathcal{D}}$ with budget $b^*$ and denote result with $y$;
            if $y$ is True then
                /* that mean a schedule with a cost less or equal to $b^*$ is
                   possible if we don't perform $v$ after $u$              */
                reset the transition cost definition for the pair $u, v$;
                continue to the next $u$;
            else
                Fix $u$ before $v$ in the schedule;
                Continue to the next operation $v$;
            end
        end
    end
    Produce $s^*$ the non lazy schedule from $h^*$ by assigning each operation the earliest
      start time such that constraint of the model is met. This is the minimum cost
      schedule as well.;
    return $s^*$
end
```

□

5 Single Machine Optimisation

In this section, we consider a special results that can be derived when we have an additional assumption. We add further the assumption that the facility has exactly one machine.

The motivation to consider this special case is that our facility model generalises more than just production facilities with multiple machines. For instance if we are to consider a computer, which also receives multiple jobs of computation task which can again be of multiple stages and can involve complicated input and output device operations, reading and writing to memory, loading and clearing cache files, these can be seen as transition costs as well. In this computer facility, we still require minimising the cost and in effect the order of operation in which the tasks will be done on the single machine.

The fact we only need to optimise the order in which operations will be done on the machine is formalised in the following lemma.

Lemma 2. *For every single machine facility cost problem with given α and $\boldsymbol{\omega}$. Let, $\hat{\boldsymbol{\omega}}_{i,k} := \boldsymbol{\omega}_{i,k} + \alpha$. Then, there exists a $A \in \mathbb{N}$, such that $C(F, s, \alpha, \boldsymbol{\omega}) = A + C(F, s, 0, \hat{\boldsymbol{\omega}})$.*

Intuitively this follows from the fact that now that we only have one machine the makespan part of the cost is now only dependent on the sequence of the operations done on this one machine rather than also operations done on other machines. Hence, the total makespan is only a sum of the sequence of operations.

Proof.

$$\begin{aligned}
C(F,s;\alpha,\boldsymbol{\omega}) &= \alpha \cdot \max_{j\in J} s\left(f^{(j)}\right) + \sum_{j\in J}\sum_{i\in V_j} \boldsymbol{\omega}_{i,\mathrm{bef}_s(i)} c_m\left(i, \mathrm{bef}_s\left(i\right)\right) \\
&= \sum_{j\in J}\sum_{i\in V_j} \alpha\left(T(i) + c_m(i,\mathrm{bef}_s\left(i\right))\right) + \sum_{j\in J}\sum_{i\in V_j} \boldsymbol{\omega}_{i,\mathrm{bef}_s(i)} c_m(i,\mathrm{bef}_s\left(i\right)) \\
&= A + \sum_{j\in J}\sum_{i\in V_j} \left(\boldsymbol{\omega}_{i,\mathrm{bef}_s(i)} + \alpha\right) c_m(i,\mathrm{bef}_s\left(i\right)) \\
&= A + \sum_{j\in J}\sum_{i\in V_j} \hat{\boldsymbol{\omega}}_{i,\mathrm{bef}_s(i)} c_m(i,\mathrm{bef}_s\left(i\right)) \\
&= A + C(F,s;0,\hat{\boldsymbol{\omega}})
\end{aligned}$$

□

Therefore, essentially the cost to be minimised is only dependent on the sequence of operations rather than the exact schedule times of each operation.

We further use the following lemma to show that we only need to optimise for the parameter $\boldsymbol{\omega}$ and not for the derived weights $\hat{\boldsymbol{\omega}}$.

Lemma 3. *Let the parameter $\boldsymbol{\omega}$ be given and define $\hat{\boldsymbol{\omega}}_{i,k} := \boldsymbol{\omega}_{i,k} + \alpha$. Then, the schedule, s, that minimises $C(F,s;0,\boldsymbol{\omega})$ also minimises $C(F,s;0,\hat{\boldsymbol{\omega}})$.*

Proof. Let a facility F be given. Let $d := \sum_{j\in J}\sum_{i\in V_j} \alpha c_m\left(i,\mathrm{bef}_s\left(i\right)\right)$.

Then let a schedule s be given such it is the minimum cost schedule for the facility F given parameters $\alpha = 0$ and $\boldsymbol{\omega}$. For the sake of contradiction, assume there exists a different schedule $\hat{s}$ such that it is the minimum cost schedule for the facility F with parameters $\alpha = 0$ and $\hat{\boldsymbol{\omega}}$.

But then, $\hat{s}$ is also the minimum cost schedule for the facility F with parameters $\alpha = 0$ and $\boldsymbol{\omega}$ due to the following:

$$\begin{aligned}
C(F,\hat{s};0,\hat{\boldsymbol{\omega}}) &= \sum_{j\in J}\sum_{i\in V_j} \hat{\boldsymbol{\omega}}_{i,\mathrm{bef}_s(i)} c_m\left(i,\mathrm{bef}_s\left(i\right)\right) \\
&= \sum_{j\in J}\sum_{i\in V_j} \alpha c_m\left(i,\mathrm{bef}_{\hat{s}}(i)\right) + \sum_{j\in J}\sum_{i\in V_j} \boldsymbol{\omega}_{i,\mathrm{bef}_{\hat{s}}(i)} c_m\left(i,\mathrm{bef}_{\hat{s}}(i)\right) \\
&= \hat{d} + \sum_{j\in J}\sum_{i\in V_j} \boldsymbol{\omega}_{i,\mathrm{bef}_{\hat{s}}(i)} c_m\left(i,\mathrm{bef}_{\hat{s}}(i)\right) \\
&= \hat{d} + C(F,s;0,\boldsymbol{\omega})
\end{aligned}$$

Since, $\hat{d} := \sum_{j\in J}\sum_{i\in V_j} \alpha c_m\left(i,\mathrm{bef}_{\hat{s}}(i)\right)$ is only a constant given a schedule $\hat{s}$ which is the minimum cost schedule for facility F with parameters $\alpha = 0$ and $\hat{\boldsymbol{\omega}}$, the same schedule $\hat{s}$ is also the minimum cost schedule for the facility F with parameters $\alpha = 0$ and $\boldsymbol{\omega}$. Hence, since schedule s is different from schedule $\hat{s}$, s is not a minimum cost schedule for the facility F with parameters $\alpha = 0$ and $\boldsymbol{\omega}$, which is a contradiction. □

Having gained this insight, it opens up a big toolbox of algorithmic results and mathematical results that we can apply to our single machine facility cost problem that we cover in the next two subsections.

5.1 Quasi-Metric Transition Costs

We present here an approximation for the single machine facility cost problem. To produce the following results we need some additional assumptions, each operation has no dependencies and the transition costs are quasi-metric. This is stated here.

Definition 20 (λ-Quasi-Metric)**.** *The facility only has one machine, which we denote by m. Then the transition costs with their weights respect the following:*

- *The triangle inequality. That is, given for all $u, v, w \in V|_m$, it holds that $\boldsymbol{\omega}_{w,u} c_m(w,u) \leq \boldsymbol{\omega}_{v,u} c_m(v,u) + \boldsymbol{\omega}_{w,v} c_m(w,v)$ and*
- *for each u we have that $v \in V|_m \setminus ans(u)$ we have that $\boldsymbol{\omega}_{v,u} c(v,u) \leq \lambda \boldsymbol{\omega}_{u,v} c(u,v)$.*

Note: We do not relax the assumption that transition costs with their weights are not necessarily symmetric, hence quasi-metric.

Motivation Before we present the results and the algorithm, it is also worth considering whether this situation is realistic and worth considering. We argue that this assumption in practice is a plausible occurrence, since the assumption essentially states that the total cost of transitioning between three operations (from u to v and v to w) is greater or equal to the cost of transitioning straight from u to w. This is realistic as well, because adding an extra operation is intuitively expected to have a higher total cost than none. Since in many cases we expect a similar situation to exist, we consider this case specially.

However, demanding that there is only no precedence operation is very restrictive, but since we already have restricted ourselves to a single machine it is not unlikely that the operations to be done are also sequential, and each job is just one operation performed independently of other operations in other jobs. Additionally, mathematically, it also allows us to use more techniques which we are able to compare and contrast to existing techniques.

Theorem 4. *Given a single machine facility instance F where operations have no dependencies (single operation jobs) and parameters α and $\boldsymbol{\omega}$ for the cost function such that the facility is λ-Quasi-Metric. Let the optimal schedule be s^*, then there exists a polynomial time approximation algorithm which gives a schedule, s which has the cost $C(F, \hat{s}; \alpha, \boldsymbol{\omega}) \leq (1+\lambda)\, C(F, s^*\ \alpha, \boldsymbol{\omega})$.*

This special case is interesting due to the ease with which a result can be approximated in polynomial time and depending on the instance of the problem the approximation bounds can be very small but also arbitrarily bad since the max ratio λ can be very large.

For the approximation algorithm, we first present a transformed version of the problem which is equivalent to minimising the cost function but allows us to use an approximation algorithm. This is given in Algorithm 4. The approximation algorithm is given Algorithm 5.

Note: we require each operation to only have one dependency, otherwise it is possible that during the step of calculating the minimum directed spanning tree and then later in finding the pseudo tour, we might actually visit an operation and then one of its ancestors,

Algorithm 4: Convert a Quasi-Metric Cost Single Machine Facility to an Acyclic Digraph

```
function convertFacilityToDiGraph (F) is
    First we define the following constant:
        $\chi \in \mathbb{N} \text{ such that } \chi > \max_{u \in V \; v \in V|_m \setminus \text{ans}(u)} \boldsymbol{\omega}_{u,v} c_m(u,v) \quad (8)$
    Then, we define a new digraph $\hat{D} = \left(\hat{V}, \hat{E}\right)$ in the following way:
    $\hat{V} = V|_m$, all the operations that need to be performed on the machine $m$;
    Add the following directed edges $\forall u, v \in V|_m$ where $u \neq v$;
    /* Since each job is just one operations we do not have to worry
       about ancestors.                                              */
    add edge $(u, v)$ to $\hat{E}$ with edge cost $\boldsymbol{\omega_{v,u}} c_m(v, u)$
    Add a universal vertex, $\hat{u}$ and add edge from $\hat{u}$ to all other vertices and an edge
      from all other vertices to $\hat{u}$, all with edge costs $\chi$.
    Notice, these edges also satisfy the triangle inequality.
        $\boldsymbol{\omega}_{u,v} c_m(u,v) \leq \chi \leq \chi + \chi = \boldsymbol{\omega}_{u,\hat{u}} c_m(u,\hat{u}) + \boldsymbol{\omega}_{\hat{u},v} c_m(\hat{u},v)$
        $\boldsymbol{\omega}_{u,\hat{u}} c_m(u,\hat{u}) \leq \chi \leq \boldsymbol{\omega}_{u,v} c_m(u,v) + \boldsymbol{\omega}_{v,\hat{u}} c_m(v,\hat{u})$
        $\boldsymbol{\omega}_{\hat{u},v} c_m(\hat{u},v) \leq \chi \leq \boldsymbol{\omega}_{\hat{u},u} c_m(\hat{u},u) + \boldsymbol{\omega}_{u,v} c_m(u,v)$
    return $\hat{D}$
end
```

Algorithm 5: Approximation Algorithm for Quasi-Metric Costs

```
function approximateQuasiMetricFacility (F) is
    $\hat{D} \longleftarrow convertFacilityToDiagraph(F)$;
    Take the minimum directed spanning tree of the di-graph, $\hat{D}$ rooted at $\hat{u}$ and
      denote it with $MDST$.
    /* This can be computed in $O(|\hat{E}| + |\hat{V}| \log(|\hat{V}|))$ using Edmond's
       Algorithm [11]                                                 */
    Perform a Depth First Search ($DFS$) on $MDST$ creating a list in the order the
      vertices are visited;
    When performing the depth first search on this tree, we traverse first the children
      of the current node and only then the siblings recursively. In this step each visit
      back to the parent vertex before visiting the sibling is also recorded every time;
    We call this the Pseudo Tour ($PT$);
    Create the order in which the jobs have to be processed by deleting all repeated
      visits (keep only the first visits to each vertex) and removing all visits to the
      universal node and call this $P^*$;
    Use $P^*$ as the order of operations that need to be performed on the machine $m$;
    Create the non lazy schedule $\hat{s}$ based on the order of operations such that
      constraints are met;
    return $\hat{s}$
end
```

which should not be allowed. This is not possible when there is only one dependency for each operation.

Now, having given the algorithm, we also need to provide the proof of the bounds and its running time. We begin by providing the proof for the bounds on the cost of the schedule produced by the algorithm.

Lemma 4. *The cost of the schedule produced by the Algorithm 5 is at most an* $(1+\lambda)$ *approximation ratio algorithm.*

Proof. For convenience, we define the following function. Given a single machine facility instance F with parameters α and $\boldsymbol{\omega}$. Then the weight of a graph $H = (V^*, E^*)$ where $V^* \subseteq V_m$ and $E \subseteq V|_m \times V|_m$ as:

$$w(H) = \sum_{(u,v)\in E^*} \boldsymbol{\omega}_{u,v} c_m(u,v)$$

where c_m is the transition cost function of the machine m that is given.

Claim 3. *Denote the minimum directed spanning tree (also called arborescence) of* $\hat{D}$ *by* $MDST$ *as defined in Algorithm 5. Then* $w(MDST) \leq OPT$ *where* OPT *is the weight of the optimal tour of operations on the machine,* m *denoted by a tour* D.

Note: this graph is a tour, so it only has one edge that makes it cyclical. Since at the end of the algorithm we remove the visits to the universal edge, we get a path. Essentially, the weight for any tour is increased by a constant amount of 2χ. For any tour, when we delete the start and end visit to the universal node, we get a path, and this path translates to the order of operations in which we need to perform the operations on the machine.

Proof. For the sake of contradiction, assume $OPT < w(MDST)$. Remove the edge from D that makes it cyclical and call this directed tree SDT. Then we have a SDT such that $w(SDT) \leq OPT < w(MDST)$. Hence, $MDST$ is not minimal, which is a contradiction. □

Claim 4. *Let the pseudo tour as defined in Algorithm 5 we denoted with PT. Then,* $w(PT) \leq (1+\lambda)\,OPT$.

Proof. For convenience, define the digraph with all edges in minimum spanning directed tree reversed as $IMDST$.

$$\begin{aligned}
w(PT) &= w(MDST) + w(IMDST)\\
&= w(MDST) + \sum_{(u,v)\in MDST} \boldsymbol{\omega}_{v,u} c_m(v,u)\\
&= w(MDST) + \sum_{(u,v)\in MDST} \boldsymbol{\omega}_{u,v} c_m(u,v) \frac{\boldsymbol{\omega}_{v,u} c_m(v,u)}{\boldsymbol{\omega}_{u,v} c_m(u,v)}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \boldsymbol{\omega}_{u,v} c_m(u,v) > 0\\
&\leq w(MDST) + \max_{(u,v)\in MDST}\left(\frac{\boldsymbol{\omega}_{v,u} c_m(v,u)}{\boldsymbol{\omega}_{u,v} c_m(u,v)}\right) \sum_{(u,v)\in MDST} \boldsymbol{\omega}_{u,v} c_m(u,v)\\
&\leq (1+\lambda)\, w(MDST)\\
&\leq (1+\lambda)\, OPT
\end{aligned}$$

□

Claim 5. *Let P^* be as defined in Algorithm 5. Then, $w(P^*) \leq w(PT)$, and hence, $w(P^*) \leq (1 + \lambda)\, OPT$.*

Proof. This follows immediately from the triangle inequality. Notice, in the tour when edges to a repeated vertex are made, they are replaced with the edge going from destination of the out edge from the repeated vertex, and therefore because of the triangle inequality which is assumed to hold for the weighted costs, the result follows.

The edges to and from the universal vertex also hold the triangle inequality, so when visits to the universal edges are removed all together □

Claim 6. *The cost of the non-lazy schedule $\hat{s}$ that is produced from Algorithm 5 respects $C(F, \hat{s}; \alpha, \boldsymbol{\omega}) \leq (1 + \lambda)\, C(F, s^*; \alpha, \boldsymbol{\omega})$*

Proof. We only need to optimise the order of operations due to Lemma 2 and Lemma 3, hence in the worst case we have that $C(F, \hat{s}; \alpha, \boldsymbol{\omega}) \leq (1 + \lambda)\, C(F, s^*; \alpha, \boldsymbol{\omega})$. □

Note that the worst case is achieved only when the term $\sum_{j \in J} \sum_{i \in V_j} \alpha T(i)$ is zero. Therefore, this approximation algorithm will always perform with a smaller approximation ratio than $(1 + \lambda)$ when the term is non-zero.

Hence, we have now proven the approximation ratio bound on our approximation algorithm, as stated in Lemma 4. □

We now provide the proof that the approximation algorithm given by Algorithm 5 is a polynomial time with respect to the instance.

Lemma 5. *Algorithm 5 is a polynomial time algorithm.*

Proof. We denote the total number of operations with $k := |\hat{V}|$.

- Finding the maximum edge cost from the weighted cost function is in the order of $O(k^2)$, since we have the order of k^2 weighted cost function expression to compute in the graph we construct.
- Finding $MDST$ is in the order of $O(k^2)$ using the Edmond's Algorithm [11] which has time complexity of $O(|\hat{E}| + k \log(k)|)$ but since the number of edges are in the quadratic order of the number of vertices we have the complexity of this step in $O(k^2)$.
- Performing the DFS is linear in the number of edges of the minimum directed spanning tree, hence $O(k)$
- Deleting the repeated visits and the visits to the universal edge is again linear in the number of operations, therefore $O(k)$.

Therefore, the time complexity of this algorithm is $O(k^2)$. □

Now we can provide the proof of Theorem 4.

Proof. It follows directly from Lemma 4 and Lemma 5. □

5.2 Using Solvers of *ATSP*

In this section we make use of well known problem in discrete optimisation Travelling Salesperson Problem (*TSP*) and the Asymmetric Travelling Salesperson Problem (*ATSP*). We show that we can essentially use any solver (exact or approximate) for *ATSP* for this special case of the facility optimisation problem. The formulation of *TSP* and *ATSP* can be found in Appendix C and Appendix D respectively. The main difference between the two formulations is the fact in *ATSP* costs are allowed to be asymmetric, i.e. the travelling cost between vertices can be different when traversed in the opposite direction, while in *TSP* they are not.

For the single machine, we can also apply the Subtour Elimination Heuristic [12] by constructing an equivalent *ATSP* problem instance which can be used for *ATSP* without assuming the costs are quasi-metric.

There are two ways to apply the Subtour Elimination algorithm, using the Mixed Integer Linear Program Formulation from Miller-Tucker-Zemlin formulation for TSP [13] which can be adapted for *ATSP* or by using the Hungarian Algorithm [14] for solving *ATSP*.

Note that both methods are adapted from *ATSP* and this is sufficient since following from Lemma 2 where we showed that the cost can be minimised by minimising the cost of the order of operations on the machines, which is exactly the objective function of *ATSP* as well, visiting all cities exactly once such that cost to visit them is minimal. For solving both formulations, we only need to adapt our given problem by adding a universal vertex that as in and out edge to and from every other operation. Here, instead of proving the effectiveness of any one of these techniques, we instead prove a more general result.

Theorem 5. *Given a single machine instance F with $k := |V|$ operations, then given a solver, $\mathcal{S}$ for ATSP with running time $O(T_{\mathcal{S}}(k))$, Algorithm 6 solves the optimisation problem with the running time of $O(T_{\mathcal{S}}(k) + k^2)$.*

Proof. We only need to optimise the order of operations due to Lemma 2 and using only the parameter $\boldsymbol{\omega}$ provided instead of the derived weights $\hat{\boldsymbol{\omega}}$ due to Lemma 3.

Adding the universal edge to the problem instance is in order $O(k)$, where k is the number of operations. Defining the function δ for each pair of operations takes order of $O(k^2)$, since we have k^2 different pairs. The *ATSP* solver runs in the given run time of $T_{\mathcal{S}}(k)$. Creating the schedule with the earliest possible start time from the order of operations, again takes order $O(k)$ order of operations. Therefore, the running time of the Algorithm 6 is $O(T_{\mathcal{S}}(k) + k^2)$. □

Corollary 2. *Given an approximation algorithm, $\mathcal{A}$, for ATSP with a running time of $O(T_{\mathcal{A}}(k))$. and a single machine facility instance F, then Algorithm 6 solves the optimisation problem with the same approximation ratio and running time of the approximation algorithm $O(T_{\mathcal{A}}(k) + k^2)$.*

Proof. The proof for the running time is the same as in Theorem 5. We only need to prove the quality of the approximation ratio. Since the instance we create only has one extra vertex, the universal node v_0, with ingoing and outgoing edge costs of 0 to all other nodes, the final schedule created without the node v_0 has a cost reduced by 0, hence has the same approximation ratio as the approximation algorithm for the *ATSP* given by $\mathcal{A}$. □

Algorithm 6: Single Machine Facility solver from *ATSP* solvers

```
function singleMachineSolverFromATSPSolver ($\mathcal{F}$, ,$\mathcal{S}_{ATSP}$) is
    Let $v_0$ be a dummy operation with zero processing time and no dependency that
      runs on the machine $m$.;
    /* $m$ is the single machine                                                */
    $V \longleftarrow V|_m \cup \{v_0\}$;
    $E \longleftarrow \bigcup \{(u, v) \in V|_m \times V|_m \mid u \neq v \wedge v \notin \text{ans}(u)\}$;
    /* Set the edge costs;
                                                                                */
    $\delta_{v_0 v} \longleftarrow 0$;
    $\delta_{u v_0} \longleftarrow 0$;
    $\delta_{uv} \longleftarrow \boldsymbol{\omega}_{u,v} c_m(u, v)$;
    $\delta_{uu} \longleftarrow \infty$;
    $\delta_{uv} \longleftarrow = \infty$ if $v \in \text{ans}(u)$.;
    $h^* \longleftarrow \mathcal{S}_{ATSP}(V, E, \delta)$;
    /* Produce the order in which the operations need to be performed
       using $\mathcal{S}_{ATSP}$;
                                                                                */
    Produce the schedule $s^*$ using the order of operations in $h^*$ starting from the
      dummy operation $v_0$ and ending at the same operation $v_0$ scheduling each
      operation at the earliest possible time.;
    Remove $v_0$ from the schedule;
    /* The final cost is unaffected either way since it only adds a
       w=cost of 0                                                              */
    return $s^*$.
end
```

This theorem is powerful in the sense that it allows us to solve any single machine facility problem using a given *ATSP* solver, which allows for asymmetric transition costs.

Also, we can not use *TSP* solvers in general unless they do not require costs to be symmetric, and also approximation algorithms like the Christofides Algorithm [15] do not work unless stronger assumptions are made, namely that the costs are metric.

Using Corollary 2 we can also give the following theorem.

Theorem 6. *For single machine optimisation, there exists a* $22 + \epsilon$ *for any* $\epsilon > 0$*, approximation ratio algorithm.*

Proof. Using the results of Theorem 46 in article [16], and Corollary 2 the result follows. □

6 Multiple Machine Optimisation

When dealing with multiple machines, because of the more complex setting in which operations may dependencies on an operation running on a different machine, i.e. Lemma 2 is not true for multiple machine facility cost problem. It is not enough to minimise the cost based only on the sequence of operations. This is because that could lead to idle times on other machines which could be optimised which in the case of a single machine would give a higher cost but in terms of the objective function the cost is actually lower.

In this section, we will approach the multiple machine cost problem through many ways.

6.1 Exact Dynamic Programming Approach

The first approach that we look at is an exact algorithm that we use the dynamic programming technique.

Despite usually having exponential running time, exact algorithms are still valuable to write since they can be used as a subroutine for approximation algorithms when the sample instance is broken into a smaller instance, and it also helps us understand the details of the problem where approximation can be made.

Theorem 7. *Let an instance of Facility Cost Optimisation problem F be given. Let $k := |V|$. Then, there exists a $\Theta(k^2 2^k)$ running time exact algorithm to solve the instance.*

The algorithm with its initial call is given in Algorithm 7 with recursive call in Algorithm 8.

Algorithm 7: Initial Call for Dynamic Algorithm

```
function exactDynamicFacility (F) is
    $F := \langle J, M, \boldsymbol{D}, \boldsymbol{c} \rangle$;
    for $j \in J$ do
        $s(u) \longleftarrow \infty \ \forall u \in V_j$;
    end
    for $m \in M$ do
        $\boldsymbol{S} \longleftarrow (S_m)_{m \in M}$ where;
        $S_m \longleftarrow V|_m$;
    end
    $I = \left\{ u \in \bigcup_{m \in M} S_m \mid \mathrm{pred}_s(u) = \emptyset \right\}$;
    /* set of first operation in each job */
    $minCost \longleftarrow \infty$;
    $s^* \longleftarrow s$;
    for $u \in I$ do
        Copy schedule $\hat{s} \longleftarrow s$;
        $\hat{s}_u \longleftarrow 0$;
        $\hat{\boldsymbol{S}} \longleftarrow (S_m \setminus \{u\})_{m \in M}$;
        $candidateMinCost = \min_{u \in \hat{\boldsymbol{S}}} \left\{ cost(F, u, \hat{\boldsymbol{S}}, \hat{s}) \right\}$;
        if $candidateMinCost < minCost$ then
            $minCost \longleftarrow candidateMinCost$;
            $s^* \longleftarrow \hat{s}$
        end
    end
    return $minCost$, $s^*$
end
```

Algorithm 8: Recursive Call for the Dynamic Program

```
function cost (F, u, S, s) is
    if $\sum_{m \in M} |S_m| \neq 0$ then
        /* if not all operations are scheduled */
        $minCost \longleftarrow \infty$;
        $s^* \longleftarrow s$;
        for $v \in S$ do
            Copy schedule $\hat{s} \longleftarrow s$;
            if $pred_s\,(v) = \emptyset$ then
                schedule $v$ to begin on the first free time on machine $M(v)$;
            end
            else
                schedule $v$ to be begin on the maximum time over the end of it's
                  predecessors and first free time on machine $M(v)$;
                if $s(v) = \infty$ then
                    skip to the next $v$;
                end
            end
            $\hat{\boldsymbol{S}} \longleftarrow (S_m \setminus \{v\})_{m \in M}$;
            $candidateMinCost = \min_{v \in \hat{\boldsymbol{S}}} \left\{ cost(F, v, \hat{\boldsymbol{S}}, \hat{s}) + \boldsymbol{\omega}_{v,u} c(v, u) \right\}$;
            if $candidateMinCost < minCost$ then
                $minCost \longleftarrow candidateMinCost$;
                $s^* \longleftarrow \hat{s}$
            end
        end
    else
        $minCost \longleftarrow \max_{j \in J} \alpha s(f^{(j)})$;
        $s^* \longleftarrow s$
    end
    return $minCost, s^*$;
end
```

In the dynamic program, the costs saved in each step are identified using $(u, \hat{\boldsymbol{S}})$. The facility F is a constant and the schedule is part of the recursive call only for the final schedule construction.

We provide the proof for the time complexity of the algorithm here.

Proof. Let $k := |V|$. The algorithm considers all 2^k subsets of V. For each subset, the algorithm computes the cost function for all its elements, of which there are at most k. For each of those k elements, the algorithm again computes no more than k values based on the results obtained from the previous step. In effect, the time complexity is in the order of $\Theta(k^2 2^k)$. □

The results obtained for the multi-stage cost optimisation problem are consistent with the Bellman-Held-Karp algorithm [17] which was the dynamic program given to solve the *TSP*.

6.2 Mixed Integer Linear Programming Approach

Our next approach looks at the formulation of a Mixed Integer Linear Program (*MILP*) which is also inspired from a similar formulation for *TSP*. Here, we look at adapting the Miller-Tucker-Zemlin formulation [13] for our problem. This formulation is a mixed integer linear program which was first given to give an exact result of the *TSP*, however this is still only feasible for small instances since Mixed Integer Linear Program are in the class of *NP*-hard.

Define for all $i \in V$ $s(i) \in \mathbb{N}$ the starting time of the operation i on the respective machine $M(i)$. Then we can rewrite the objective function to minimise by introducing the decision variables and the constraints as follows:

$$\min_{s,z,x} \; z + \sum_{j \in J} \sum_{i \in V_j} \sum_{k \in V|_{M(i)}} \boldsymbol{\omega}_{i,k} c_m(i,k) x_{ki}$$

$s.t.$

$$z \geq \alpha s\left(f^{(j)}\right) \qquad \forall j \in J \tag{9}$$

$$\sum_{k \in V|_{M(u)}} x_{kv} = 1 \qquad \forall v \in V \tag{10}$$

$$\sum_{k \in V|_{M(u)}} x_{uk} = 1 \qquad \forall u \in V \tag{11}$$

$$s(v) \geq s(u) + T(u) + \sum_{k \in V|_{M(u)}} c_m(u,k) x_{ku} \qquad \forall v \in V \; \forall u \in \operatorname{pred}_s(v) \tag{12}$$

$$s(v) \geq \sum_{k \in V|_{M(v)}} (s(k) + T(k) + c_m(v,k)) x_{kv} \qquad \forall v \in V \tag{13}$$

$$s(u) \in \mathbb{R}_{\geq 0} \qquad \forall u \in V \tag{14}$$

$$x_{kv} \in \{0,1\} \qquad \forall v \in V \; \forall k \in V|_{M(i)} \tag{15}$$

First, notice that the objective function no longer has the makespan. This is because max is not a linear function and therefore can not be used in *MILP* formulation. However, we can re-write the inner max as part of the constraints, as done in Equation 9.

The constraint of Equation 10 requires that an operation is done following exactly one other operation on the machine. This follows from the constraint since for all operations u it is preceded by exactly only one other operation.

The constraint of Equation 11 requires that an operation is only followed by one other operation. This eliminates the possibilities of scheduling multiple operations on the machine after one operation.

The constraint of Equation 12 covers the fact the operation can only be completed after the end time of all of its dependencies. The end time includes the starting of the job, the processing time of the job and the extra time incurred because of the operation preceding the dependency.

The constraint of Equation 13 covers the fact that we are working with only single processor machines, and so the processing operation must be complete before we can start the next operation.

Constraints of Equation 14 tells us each start time can be a real positive number and Equation 15 is the decision that operation i is performed after operation k.

This mixed integer linear program produces an exact optimal schedule giving the start time of each operation that minimises the objective function. However, because of the decision variables, this *MILP* is only feasible for small instances. This is because since the decision variables are binary (0 or 1) the program will need to try many permutations rather than optimise continuously over the decision variables [18].

6.3 Relaxed *MILP*

Now we relax the integer value constraints to produce a linear program that we can then use to give a greedy algorithm.

Relax Equation 15 to the following:

$$x_{ki} \in [0, 1] \tag{16}$$

If after solving the linear program is all the decision variables are only 0 and 1 then we do not need to approximate, and we have an exact result.

In case the results are not binary, we can use advanced techniques like branch and bound algorithm or use greedy rounding such that primal problem constraints are still met. We discuss greedy rounding below.

In case of rounding, we also need to shift the operations forward in the schedule. This follows from since the decision variables were relaxed to fractions Equation 12 and Equation 13 inequalities correspond to the schedule where the constraints of all the dependency operation being completed or the last operation on the machine having being completed may not be true. So, due to the shifting, we have a $O(k)$ approximation ratio algorithm. We state this formally in the following theorem.

Theorem 8. *Given a facility instance F with k total number of operations. The relaxed MILP when used with greedy rounding is a $O(k)$ ratio approximation algorithm.*

Proof. When we apply greedy rounding, we round each decision variable to either 0 or 1 based on which was the maximum and such that Equation 10 and Equation 11 constraints are met.

When these are rounded, the constraints of Equation 12 and Equation 2 may not be met. This involves sequentially shifting the operations schedule iteratively. The increase in cost is bounded above by K, as defined in Lemma 1 for each operation. This is because K is defined to be the maximum possible cost that any operation can incur on the facility.

We do this at most k times, in case each operation need to be shifted in such a manner. Hence, the final schedule's cost is increased by at most increased kK. Therefore, the approximation ratio scales linearly with the number of operations k, hence the approximation ratio is $O(k)$.

Therefore, for the greedy rounding scheme for the relaxed *MILP* is a $O(k)$ ratio approximation algorithm. □

In terms of an approximation algorithm, rounding would be the least beneficial method due to the fact the approximation would become worse when there are more operations in terms of how far from optimal the result would be.

6.4 Greedy Algorithm

A greedy algorithm is one of the approaches for an approximation algorithm that can be given which does not enforce any extra assumptions on the problem. However, a greedy algorithm is usually not the best approximator of the optimum solution since the idea of the greedy algorithm is to make decision based on the current state only rather than on future states examples can be produced where we can see that a greedy algorithm is arbitrarily bad. The reason to still consider greedy algorithms are

1. Quick results can be produced
2. Bounds can be achieved to restrict the search space for other more advanced algorithms
3. In some cases greedy algorithms are as good as other approximation algorithms, hence they are worth the consideration before more assumptions are enforced.

The algorithm is given in Algorithm 9.

Algorithm 9: Greedy Algorithm for Facility Problem

```
function greedySolver (F) is
    Let $\hat{s}$ and $s^*$ be schedules with no operations scheduled;
    $\hat{V} \longleftarrow \bigcup_{j\in J} V_j$;
    while $\hat{V} \neq \emptyset$ do
        $x \longleftarrow \arg\min_{v\in\hat{V}} C(\hat{F}, \hat{s}; \alpha, \boldsymbol{\omega})$;
        where $\hat{F}$ is the facility problem augmented only to include the operations
          $V \setminus \hat{V}$;
        and $\hat{s}$ is the schedule with $v$ scheduled in a non lazy way, and in case $v$ can not
          be scheduled due to dependency not being met, the cost is simply replaced
          with $\infty$;
        Schedule $x$ in the schedule $s^*$ in a non lazy way;
        Remove $x$ from $\hat{V}$;
        Set $\hat{s} \longleftarrow s^*$;
    end
    cost $\longleftarrow C(F, s^*; \alpha, \boldsymbol{\omega})$;
    return cost, $s^*$
end
```

This algorithm is greedy in terms of always choosing the operation which increases the cost of the facility by the least in each step of the scheduling process. This algorithm can be arbitrarily bad. This can be seen in the following example.

In Figure 1 we have a job with 4 operations $\{0, 1, 2, f\}$ with 0 and f being the start and final jobs respectively. Operations $1, 2$ both require an operation 0 to be completed and f requires both operations $1, 2$ to be completed. All operations need to occur on one machine, and on the edges of the figure we have written the weighted transition costs. Let K be a cost that is arbitrarily big. According to the greedy algorithm and the constraints of the Facility Cost Problem, we have a schedule in which the order of operations are $0 \to 1 \to 2 \to f$. This means our cost is increased with at most K cost (which is arbitrarily large) per operation. Hence, the greedy algorithm can be arbitrarily bad, bounded with max transition cost and the number of operations, hence the greedy

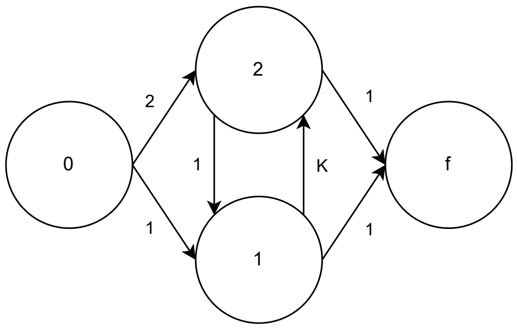

FIGURE 1: Example when Greedy Algorithm Can be Arbitrarily Bad

algorithm is also $O(k)$ approximation algorithm where k is the number of operations.

Theorem 9. *The greedy algorithm, as written in the Algorithm 9, is an approximation algorithm with ratio $O(k)$ where k is the number of operations with running time of order $O(k^2)$.*

6.5 Special Case with Quasi-Metric Costs and Multiple Machines

First, we define the notion of a restricted facility to a machine $m \in M$.

Definition 21 (Restricted Facility)**.** *An instance of facility, F is restricted to a machine $m \in M$ is defined by $F|_m := \langle J, m, \boldsymbol{D}|_m, c_m \rangle$ where each dependency graph $D_i \in \boldsymbol{D}|_m$ is defined by:*

- *$D_i = (V_i|_m, E_i|_m)$ where $V_i|_m$ are only the operations that occur on the machine m and $E_i|_m$ are the edges that preserve the ancestry in the original dependency graph.*

For instance, we have a job which has the operations a on machine 1, operation b on machine 2 to be done after operation a and operation c on machine 1 to be done after operation b. Then the facility restricted to machine 1 will augment the job such that we have operation a on machine 1 and operation c on machine 1 after operation a, hence preserving the ancestry and only operations of that specific machine exist.

Now, if again we assume the costs are quasi-metric, but now we use the parameters, $\alpha = 0$ and arbitrary $\boldsymbol{\omega}_{uv} \in \mathbb{N}$ we again have an $(1 + \lambda)\, OPT$ approximation algorithm, since the cost only is a sum of the weighted transition costs on each machine regardless of how far apart in time each operation might be scheduled, which can achieved by using Algorithm 5 but on the restricted facility on each machine as defined in Definition 21. This is written as a corollary below.

Corollary 3. *For a facility with parameters $\alpha = 0$ and $\boldsymbol{\omega}$ given such that it is λ-quasi-metric and where each operation has no precedence constraint (single operations jobs), there exists a $(1 + \lambda)$ ratio approximation algorithm, where*
$\lambda = \max_{\forall m \in M \ u,v \in V|_m} \frac{\boldsymbol{\omega}_{u,v} c_m(v,u)}{\boldsymbol{\omega}_{v,u} c_m(u,v)}$

Proof. Follows from only the order of operations on each machine is taken into account in the cost function since $\alpha = 0$ and Theorem 4 applied to the facility restricted to each machine iteratively. □

In case the $\alpha \neq 0$ we can not conclude the same as we did for in the case of single machine due to the fact the dependency on other machines add a non-zero value to the final cost which may lead to an order of operations which is more expensive to actually be beneficial for the overall cost.

Similarly, for the case $\alpha = 0$ and arbitrary $\boldsymbol{\omega}_{u,v} \in \mathbb{N}$ but without quasi-metric costs assumption, we can again use Theorem 5 but repeated for each machine to produce an exact solution that minimises the Facility Problem using any solver (exact or approximate) of *ATSP*. This is written formally below.

Corollary 4. *For Multiple machine instance with parameter $\alpha = 0$, arbitrary $\boldsymbol{\omega}_{u,v} \in \mathbb{N}$ and k number of operations then given any solver, $\mathcal{A}$ for ATSP (exact or approximate) with running time $O(T_{\mathcal{A}}(k))$ and approximation ratio $\beta(k)$ will solve this facility cost problem in the running time of $O(T_{\mathcal{A}}(k) + k^2)$ and the same approximation ratio.*

Proof. Follows directly from the fact that only the order of operations on each machine is taken into account in the cost function, and Theorem 5 and Corollary 2 applied to the facility restricted to each machine iteratively. □

Finally, we can conclude the existence of a constant ratio approximation algorithm when $\alpha = 0$ using the previous theorem.

Corollary 5. *For multiple machine facility cost problem with $\alpha = 0$, there exists a $22 + \epsilon$ for any $\epsilon > 0$, approximation ratio algorithm.*

Proof. Using the results of Theorem 46 in article [16], and Corollary 4 the result follows. □

Here we again we can not make the same conclusion with general parameter values of α, as the maximum makespan of the job might be minimised for an order of operations which has a more expensive sum of transition costs.

This owes itself to the discussion about how choice of parameter has a dramatic effect on the number of feasible algorithms one could use to solve the problem.

Motivation for $\alpha = 0$ At first this is a result that follows from the algorithms and considerations that we have made so far, and it is hard to find an intrinsic reason why a production manager would ever require this case. However, such a case is also useful. In situations where the total makespan is not an object of concern either because the weights α are all small or perhaps because the transition costs are too big that it dwarfs the sum added by the makespan times of each job, it is beneficial to disregard the total makespan time since optimising the transition cost and by effect the order of operations on each machine would also optimise the general cost function and doing so with an algorithm with a faster runtime.

This also reflects the benefit of modelling the transition costs as a product of transition time and cost per unit time of transition, since it also allows the production facility manager to choose the $\boldsymbol{\omega}$ to fit the priority of the facility. For instance perhaps a cleaning is cheap, but it ends up damaging the facility such that it incurs higher costs later in the future, the production facility manager can increase the specific weight for that transition without changing the other values and produce an alternate schedule.

7 Discussion

Importance of the Parameters of the Cost Function In the results we obtained, it is notable that the results were directly influenced by the parameters of the general cost function. For instance in the case of multiple machines, covered in Section 6.5, we see that by choosing a particular parameter, allowed us to use a different approach in the algorithms we design that do not work in general.

In this case we also have shown that by using the parameters $\alpha = 0$ and arbitrary $\boldsymbol{\omega}_{u,v}$ values, we could use any *ATSP* solver as shown in Theorem 5.

Therefore, in our study it shows that not only understanding more about the *ATSP* problem is directly beneficial for a problem that is only passingly similar, however it also exposes the differences between our problem and very well studied problems can not help us. This difference is the fact that when $\alpha \neq 0$, we have a cost function whose value changes per decision made, unlike the transition cost which are given before decisions are made. In such a case there is only a decision tree representation for our choices which explodes in size as the number of operations we need to schedule increases, leading to the case where any algorithm is at least exponential unless we choose to approximate our results or have stronger assumptions on our model like the transition costs being quasi-metric or only having a single machine.

(In)Significant Differences by Adding Assumptions The idea that by enforcing an extra assumption in our model, we are not necessarily able to produce stronger results can be found in our results.

We were able to produce a quadratic time algorithm with $(1 + \lambda)$ approximation bound in Section 2.2 and Section 6.5, as opposed to a quadratic time algorithm which was $O(k)$ approximate with k being the number of operations in Section 6.2 and Section 6.4.

On one hand the result of the approximation algorithm when we have the extra assumptions is a bound that does not scale with the number of operations in the facility but with the transition costs instead, however the assumption added in the case of the multiple machine optimisation on using specific parameters for the cost function is harder to justify.

Furthermore, if we were to consider using the exact techniques and algorithms that we also presented in our article, Section 6.2 and Section 6.1 in a production facility where even small increase in efficiencies in scheduling can translate to significant cost reductions, it can not be relied upon these algorithms to produce a result in a feasible time due to exponential time complexity they run in, however they can be used as a subroutine for the approximation result when the (augmented) instance is "small enough".

We do not however provide a mathematical proof on the existence or absence of an algorithm which can provably produce approximation results with lower bounds, or produce similar bounds without adding an extra assumption. This owes to the fact the model we work with is more involved in that there are fewer properties to exploit.

Importance of Asymmetry of the Transition Costs A final remark can be made on the cases where the asymmetry of the costs posed a problem, or in other words actually influenced the results.

Asymmetry of the transition costs directly influenced the results of the approximation algorithm in Section 2.2, namely the ratio λ. In case the costs were metric, this ratio would be exactly 2, giving us results which match the 2-approximation for metric *TSP*.

However, it is also important to note that asymmetry did not influence any of the other results we presented, for instance the Exact Dynamic Algorithm (6.1), the Mixed Integer Linear Programming Formulation (6.2) and the Greedy Algorithm (6.4). This is of note since it implies even with better approximation algorithms for *ATSP*, the problem where the distance between cities to and fro might differ, there is effectively only a difference in the special case where we add an extra assumption of quasi-metric costs and in all other cases we iterated over the different operations without ever having to modify our approach because the costs were not symmetric.

8 Future Research

Specific Values of α and $\boldsymbol{\omega}$ As discussed in the last section, the results and the possible algorithms depend on the parameters we use. This could be investigated further.

Relaxing the Constraint of Single Operation Jobs for Quasi-Metric Facilities In the proof of quasi metric facilities, we require there to be no precedence constraints, since otherwise the algorithm given may schedule an operation before visiting all the dependencies. We could not find a way to relax this constraint, and so warrants further investigation.

Use of Disjunctive Programming Disjunctive Programming [19], is a mathematical tool that is used to solve many non-convex optimisation problem including mixed integer program. This approach was not covered in this article and so it would be of interest how a different approach to solving Linear Programs would owe itself to producing different approximation algorithms and perhaps even better approximation algorithms.

Proving or Disproving existence of Approximation Schemes As stated in our article before, we did not provide the proof of existence/non-existence of (fully) polynomial time approximation schemes for the facility cost problem and any of the special variants we study. Producing such a result would extend the understanding of the topic further from the perspective of a rigorous mathematical and algorithmic spectacles.

Use of Reinforcement Learning In our research, we had the assumption that all the processing of the operations were deterministic. This simplifies the model greatly, since we do not have to consider the case of failures and delays that could add to the cost of the facility. In the case the facility wants to relax this assumption it is possible to approach the problem in multiple ways, one of which is by only taking the general mean of the processing costs, while the alternate option is to use reinforcement learning. Since the processing costs of any operation is only dependent on the previous operation and the operation itself, we have the state space of the facility that can be defined as the state of each machine and the previous states, which could then be used by a reinforcement learning model to converge to an optimal strategy.

This can be highly effective assuming we generalise further where instead of having jobs we have job profiles, i.e. jobs which always have the same set of operations and dependencies. In such a case, a model could to trained on different sets of job profiles and then instead of solving one particular case of a facility, the same model could then be used to solve multiple different facilities by performing a single call to this model with the new facility instance.

Relaxing the assumption of a one operation at a time on the machines. This assumption is a strong assumption since it essentially limits any facility to only having one of any machinery in the facility where it might be normal to have multiple. One way to circumvent this assumption is to assume a proportional model where if you have multiple machines then the time it takes to complete a task is exactly inversely proportional to the number of machines. But this modelling choice is only valid in limited scenarios, for instance it is valid to model a newspaper printing press where having twice the printing lines will produce twice the papers in the same time, however a scenario where ice cream needs to be churned for exactly 1 hour at a certain temperature, it is not possible to achieve that in 0.5 hours by having two churners. Therefore, the case where having multiple machines affects the starting time of a job, as opposed to the speed at which it can be completed, is fundamentally different and therefore requires a different consideration.

9 Conclusion

In this paper, we first produced a deterministic model of a facility and how it can be interpreted as a combinatorial optimisation problem. We equipped the model with the objective function that we would like to minimise and added our validation of the model itself from existing real life facilities.

We then used this mathematical model to provide proofs of the complexity of our problem using well established notions of Decision and Optimisation Problems, and we showed that our Decision Problem was *NP*-complete. We could already provide results on how, from an oracle that solves our decision problem, we can produce a polynomial time solver for our optimisation problem.

We continued our investigation of the problem by searching for approximation and exact algorithms for the problem, and its special case variants.

We find that our problem in the special case of a single machine optimisation or when specific cost function parameters are used we can solve the problem by the same algorithms that could be used to solve asymmetric travelling salesperson problems and under more assumptions even through solvers of symmetric travelling salesperson problem. We also define the notion of the λ-Quasi-Metric costs and how if our facility respects this and has only single operations jobs, we can produce an approximation algorithm which has approximation ratio of at most $(1 + \lambda)$.

In our approach to solve the facility cost problem with multiple machines, we presented an exact approximation algorithm and gave an equivalent mixed integer linear program. We also chose specific parameter values for the cost function, which allowed us to compare and contrast the formulation of the special case with the formulation of the asymmetric travelling salesperson problem.

We concluded by suggesting the improvements and further research that could be performed on the problem.

References

[1] N. Álvarez Gil, S. Á. García, R. Rosillo, and D. de la Fuente, “Problem instances dataset of a real-world sequencing problem with transition constraints and asymmetric costs,” *Data in Brief*, vol. 41, p. 107844, Apr. 2022.

[2] L. Shi, Z. Ma, Y. Fan, Y. Shi, X. Ding, and Z. Li, “Online Task Scheduling Algorithm with Complex Dependencies in Edge Computing,” *Procedia Comput. Sci.*, vol. 202, pp. 158–163, Jan. 2022.

[3] R. E. Gomory and T. C. Hu, “Multi-Terminal Network Flows on JSTOR,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 9, pp. 551–570, Dec. 1961.

[4] J. Błażewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, *Scheduling Computer and Manufacturing Processes.* Berlin, Germany: Springer.

[5] G. Ausiello, A. Marchetti-Spaccamela, P. Crescenzi, G. Gambosi, M. Protasi, and V. Kann, “The Complexity of Optimization Problems,” in *Complexity and Approximation*, pp. 1–37, Berlin, Germany: Springer, 1999.

[6] M. Sipser, *Introduction to the Theory of Computation.* Boston, MA, USA: Cengage Learning, June 2012.

[7] S. Arora and B. Barak, *Computational Complexity : A Modern Approach.* Cambridge, England, UK: Cambridge University Press, Apr. 2009.

[8] D. P. Williamson and D. B. Shmoys, *An Introduction to Approximation Algorithms*, p. 3–26. Cambridge University Press, 2011.

[9] Y. Robert, “Task Graph Scheduling,” in *Encyclopedia of Parallel Computing*, pp. 2013–2025, Boston, MA, USA: Springer, Boston, MA, 2011.

[10] E. W. Weisstein, “Hamiltonian Path,” *Wolfram Research, Inc.*, Mar. 2003.

[11] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, “Efficient algorithms for finding minimum spanning trees in undirected and directed graphs,” *Combinatorica*, vol. 6, pp. 109–122, June 1986.

[12] M. Desrochers and G. Laporte, “Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints,” *Oper. Res. Lett.*, vol. 10, pp. 27–36, Feb. 1991.

[13] C. E. Miller, A. W. Tucker, and R. A. Zemlin, “Integer Programming Formulation of Traveling Salesman Problems,” *J. ACM*, vol. 7, pp. 326–329, Oct. 1960.

[14] S. Akshitha, K. S. A. Kumar, M. NethrithaMeda, R. Sowmva, and R. S. Pawar, “Implementation of Hungarian Algorithm to obtain Optimal Solution for Travelling Salesman Problem,” in *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, pp. 2470–2474, IEEE, May 2018.

[15] N. Christofides, “Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem,” *Oper. Res. Forum*, vol. 3, pp. 1–4, Mar. 2022.

[16] V. Traub and J. Vygen, “An improved approximation algorithm for ATSP,” *arXiv*, Dec. 2019.

[17] M. Held and R. M. Karp, “A Dynamic Programming Approach to Sequencing Problems on JSTOR,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, pp. 196–210, Mar. 1962.

[18] “Mixed Integer Programming Algorithms,” in *Production Planning by Mixed Integer Programming*, pp. 77–113, New York, NY, USA: Springer, New York, NY, 2006.

[19] E. Balas, *Disjunctive Programming.* Cham, Switzerland: Springer International Publishing.

[20] G. Gutin and A. P. Punnen, *The Traveling Salesman Problem and Its Variations.* Springer US, May 2002.

A Commonly Used Terminology

Definition 22 (Undirected and Directed Graph)**.** *A graph $G = (V, E)$ where $\forall (u, v) \in E$ u is the starting node and v is the end node is called a directed edge from u to v. If no notion of direction is enforced then $(u, v) = (v, u)$ and we assume $(u, v) \in E \implies (v, u) \in E$, and the graph G is then called undirected.*

Definition 23 (Complete Graph)**.** *A complete graph $G = (V, E)$ is such that $E = V \times V$.*

Definition 24 (Hamilton Path)**.** *A Hamiltonian Path or traceable path is a path that visits each vertex of the graph exactly once. A graph that contains a Hamiltonian path is called a traceable graph. A graph is Hamiltonian-connected if for every pair of vertices there is a Hamiltonian path between the two vertices.*

Definition 25 (Hamilton Cycle)**.** *A Hamiltonian cycle, Hamiltonian circuit, vertex tour or graph cycle is a cycle that visits each vertex exactly once. A graph that contains a Hamiltonian cycle is called a Hamiltonian graph.*

B Proof D-HAM-PATH is NP-Complete

Theorem 10. *D-HAM-PATH is NP-complete.*

This proof has been adopted from the University of Toronto Course "CSC 463".

Proof. Let G be a directed graph. We can check if a potential s, t path is Hamiltonian in G in polynomial time. Now, we will give a polynomial-time reduction $3SAT \prec_P$D-HAM-PATH to complete the proof of completeness.

Suppose a conjunctive normal form formula $\phi = \bigwedge_{i=1}^{m} \phi_i$ has n variables and m clauses.

To simplify the proof, we may assume that no clause in ϕ contains both a variable x_i and its negation x_i, since any clause containing both literals can be removed with ϕ without affecting its satisfiability. The reduction uses the following steps.

1. First, for every variable x_i, we generate $2m+1$ vertices labelled $v_{i,j}$ and add directed edges $(v_{i,j}, v_{i,j+1})$ and $(v_{i,j+1}, v_{i,j})$ for $0 \leq j \leq 2m$.
2. Next, we connect vertices associated to different variables by adding four directed edges $(v_{i,0}, v_{i+1,0})$, $(v_{i,0}, v_{i+1,2m+1})$, $(v_{i,2m+1}, v_{i+1,0})$, $(v_{i,2m+1}, v_{i+1,2m+1})$.
3. Next, we create a vertex labelled c_j. If x_i appears in clause ϕ_j positively, add directed edges $(v_{i,2j_1}, c_j)$ and $(c_j, v_{i,2j})$. Otherwise, if x_i appears negatively in clause ϕ_j, add directed edges $(v_{i,2j}, c_j)$ and $(c_j, v_{i,2j-1})$.
4. Finally, we add two extra vertices s and t connect s by adding $(s, v_{1,1})$, $(s, v_{1,2m+1})$ and connect t by adding $(v_{n,1}, t)$ and $(v_{n,2m+1}, t)$. Call the graph generated G_ϕ. See Figure 2 for the resulting graph.

Now, we have to show that ϕ is satisfiable if and only if there is Hamiltonian path in G_ϕ from s to t. Suppose ϕ is satisfiable. Then we can visit each variable vertex starting from s by going from $v_{i,0}$ to $v_{i,2m+1}$ from left to right if x_i is assigned true, and going from $v_{i,2m+1}$ to $v_{i,0}$ from right to left if x_i is assigned false, and ending at t after $v_{n,0}$ or $v_{n,2m+1}$ is visited. Furthermore, each clause vertex can be visited since by assumption, each clause ϕ_i has some literal $l_i = x_k$ or $l_i = x_k$ that is assigned true. Each vertex c_j by can visited by using the edges $(x_{k,2j-1}, c_j)$ and $(c_j, x_{k,2j})$ in the first case, and $(x_{k,2j}, c_j)$ and $(c_j, x_{k,2j-1})$

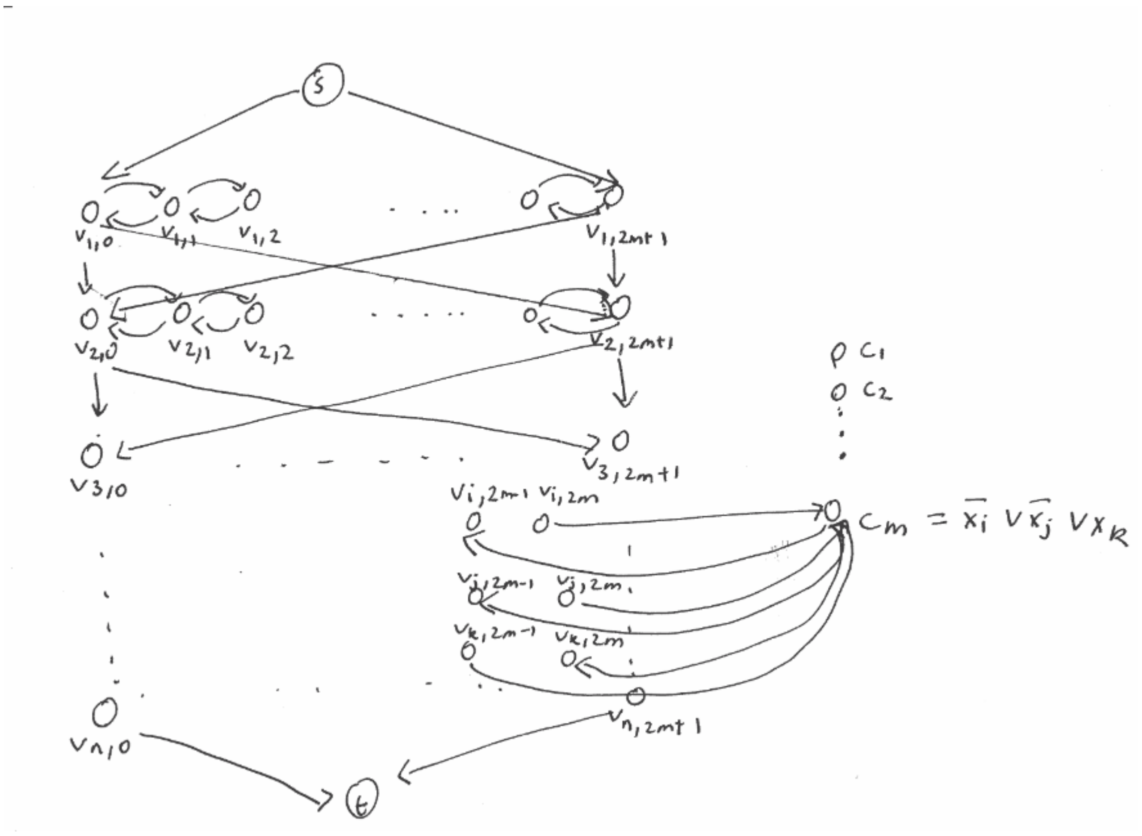

FIGURE 2: Reduction 3-SAT to D-HAM-PATH

in the second case, since the path goes right in the graph for variables assigned true and the path goes left for variables assigned false. Thus, G_ϕ has a Hamiltonian path if ϕ is satisfiable. Conversely, suppose G_ϕ has an (s,t) Hamiltonian path P. Notice that G_ϕ without the clause vertices is Hamiltonian so we need to show that if P visits $v_{i,2j-1}$, c_j, v in that order, then $v = v_{i,2j}$. Suppose $v \neq v_{i,2j}$. Then notice that the only vertices coming into $v_{i,2j}$ are $v_{i,2j-1}$, c_j, $v_{i,2j+1}$ and the only vertices coming out of $v_{i,2j}$ are to $v_{i,2j-1}$ and $v_{i,2j+1}$. Hence $v_{i,2j}$ must be visited coming in from $v_{i,2j+1}$, but now the path is stuck and cannot continue since $v_{i,2j-1}$ has already been visited. A similar argument shows that if P visits $v_{i,2j}, c_j, v$ in that order, then $v = v_{i,2j-1}$. So a Hamiltonian path in G_ϕ visits variable vertices in order from the vertices for x_1 to the vertices for x_n, interleaving the clause vertices in between variables associated to the same variable. Therefore, an assignment to the variables x_i is well defined by noticing which direction the path takes along the variable vertices for x_i in G_ϕ. By construction, this assignment will satisfy ϕ since the assignment makes a literal in each clause true.

The reduction runs in $O(mn)$ time which is a polynomial in the length of the input. □

C TSP Formulation

Definition 26 (TSP [20]). *Let $G = (V, E)$ be a graph (directed or undirected) and $\mathbb{F}$ be the family of all Hamiltonian cycles (tours) in G. For each edge $e \in E$ a cost (weight) c_e is prescribed. Then the travelling salesperson problem is to find a tour (Hamiltonian cycle) in G such that the sum of the costs of edges of the tour is as small as possible. For* *convenience we assume G to be a complete graph where missing edges are added with very large cost.*

D ATSP Formulation

Definition 27 (ATSP [20])**.** *Let $G = (V, E)$ be a graph (directed or undirected) and $\mathbb{F}$ be the family of all Directed Hamiltonian cycles (tours) in G. For each edge $e \in E$ a cost (weight) c_e is prescribed. Then the asymmetric travelling salesperson problem is to find a tour (Hamiltonian cycle) in G such that the sum of the costs of edges of the tour is as small as possible. Here we do not assume for $(u, v) \in E$ $c_{(u,v)} = c_{(v,u)}$. Also if $(u, v) \in E$ and $(v, u) \notin E$, then we add edge (v, u) to E with a very large cost.*

E Listings

E.1 List of Algorithms