PAPER NAME	AUTHOR
thesis_cs_msc_Arendes_Tim.pdf	Tim Arendes
WORD COUNT	CHARACTER COUNT
27253 Words	136023 Characters
PAGE COUNT	FILE SIZE
92 Pages	1.3MB
SUBMISSION DATE	REPORT DATE
Jan 25, 2023 12:31 PM GMT+1	Jan 25, 2023 12:33 PM GMT+1

● 10% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

- 9% Internet database
- Crossref database
- 1% Submitted Works database
- 7% Publications database
- Crossref Posted Content database

Chair of Artificial Intelligence
Prof. Dr. Jana Koehler

UNIVERSITÄT DES SAARLANDES

Saarland University
Faculty MI – Mathematics and Computer Science
Department of Computer Science
Chair of Artificial Intelligence

Master Thesis

Solving Flow Shop problems using automatic planning

Tim Arendes

First Reviewer: Prof. Dr. Jana Koehler
Second Reviewer: Dr. Daniel Fišer

Saarbrücken, January 24, 2023

Abstract

Constraint Satisfaction Problems are a widely used tool to solve different problems. They can be used to model these problems, and can then be solved by efficient solvers. A different approach could however yield better results for these problems. PDDL, an action based planning language, could be such an approach. Instead of creating a CSP model for a problem, a PDDL model could be created, for which a plan can then be found.

This thesis focuses on the Flow Shop problem. In a Flow Shop problem, multiple products must be processed at a sequence of machines, all in the same order. The goal is to optimise this processing, so that the overall needed time is minimal.

The goal of this thesis is to see, if a PDDL model of a problem can outperform a CSP model. Flow Shop will be used as the problem to see, if PDDL can solve it better than CSP. For this purpose, a PDDL model of Flow Shop will be created, which will then be used for comparisons to an existing CSP model of Flow Shop.

Contents

1 Motivation

Different approaches to problems can lead to drastically different outcomes. When trying to find a model for a problem, the choice of the model is therefore important. Two possible ways to model problems in informatics are either as a Constraint Satisfaction Problem(CSP), or with the Planning Domain Definition Language(PDDL). In this thesis, the Flow Shop problem will be used as an example to show, how a problem, for which a CSP model exists, can also be solved by PDDL.

The Flow Shop problem has many variations, but all of them consist of different jobs that need to be processed at different machines. The exact characteristics of the problem depend on the specific Flow Shop problem.

The motivation for this is, that such a different model could have advantages over the other model. The PDDL model could outperform the CSP model. This could happen by being easier to solve, so that solutions for the Flow Shop problem can be found faster, or by making it easier to find better solutions. For the Flow Shop problem, better would mean, that less time is needed to process all jobs.

2 Research outline

The overall goal of the thesis is to find out, if PDDL is an alternative to constraint solvers when it comes to solving problems, for which the Flow Shop problems will be used as a case study. This overall goal is split into multiple smaller objectives that will be described further below. They are divided into primary and secondary objectives and are used as a guide on what the exact methods are to compare PDDL to constraint solving.

2.1 Objectives

First, we define the objectives of the Master thesis.

2.1.1 Generating a PDDL model of Flow Shop

As this thesis tries to compare a PDDL model of Flow Shop with the CSP version of it, a PDDL model is needed. This model should be equal to the CSP model of Flow Shop. If this is not possible, the differences between the models should be as small as possible.

2.1.2 Translating instances to the PDDL model

To use this model, instances of the Flow Shop Problem are needed. They are already provided for the CSP model, so they only need to be translated to PDDL. For this, an automatic translation is needed, which takes the information for the CSP model, and uses them to generate a PDDL instance of the Flow Shop problem.

2.1.3 Comparison of the models

The two models need to be compared. To do this, Solvers for the CSP model and planners for the PDDL model are needed. They must be run on the generated instances, to find solutions. This can then be used for comparison. The search time can be compared to see, how much time is needed to find a solution for each model. The solution itself. The solutions themselves can also be compared. The best way to see, which solution is better, is by comparing their time needed to process every job in the Flow Shop. These times can then be compared between the different solvers and planners.

2.2 Planned experiments

As basic experiments, the models for PDDL and CSP will be solved on the same machine, so that the hardware is the same for both and one does not have an unfair advantage because of this. The experiments themselves will consist of find a solution for the models using the corresponding solvers and planners. A Benchmark set will be used, that contains different types of instances. This Benchmark set will contain multiple subsets, which will try to only change one variable of the instances in them. This will be done in order to find out, how changes to the instances affect the outcome. All solvers and planners will try to solve all of the instances.

2.3 Evaluation method

To evaluate the experiments, two main numbers can be used. The first one is the searchtime, which is the time needed to find the solution. This time could vary drastically between the instances and solvers/planners. It will therefore be evaluated, how the searchtimes change. To do this, the searchtimes of the same instance solved by different solvers and planners will be compared. In addition to this, the searchtimes of one single solver or planner will be compared over multiple instances, in which only one part of the Flow Shop changes. With this, it can be seen, how these aspects of the Flow Shop problem affect the time needed to find a solution.

The second evaluated number is the time needed in the solution, to put all jobs through the Flow Shop. This time, also called makespan, is a metric, on how good a solution is. In the Flow Shop problem, it is tried to find a solution, that can put all jobs though the machines as fast as possible. Reducing the makespan as much is possible is therefore an objective. This makespan will then again be compared between the solvers and planners, but also for one specific solver or planner with different instances.

3 Background and related work

This chapter introduces the background for this background. It includes formal definitions for the Flow Shop problem, as well as the necessary background information for CSP and PDDL. Furthermore, it includes past research on the topic, and how this influenced the work on this thesis.

3.1 Flow Shop

This thesis focuses on the Flow Shop Problem. It is a scheduling problem, in which multiple jobs are supposed to be processed on multiple machines. The task is to find an order, in which the jobs can be processed on the machines. This section will introduce already existing Flow Shop definitions, and adds extra parts to these definitions, until we arrive at the definition of Flow Shop that is actually used in this thesis. Along the way, a running example will be introduced, that will be adapted to include the additional aspects added to Flow Shop along the way.

3.1.1 Basic Flow Shop

Flow Shop problems are part of the scheduling problems. A good introduction to them is the book „Scheduling: Theory, Algorithms, and Systems“[1] by Michael L. Pinedo. It introduces the overall description of scheduling problems. They are defined as:

Definition 1 *A scheduling problem is described by a triplet $\alpha|\beta|\gamma$.*

- *α describes the machine environment and contains just one entry.*
- *β provides details of processing characteristics and constraints and may contain no entry at all, a single entry, or multiple entries.*
- *γ describes the objective to be minimized and often contains a single entry”[1, p. 14]*

Together with this, a notation is introduced. The important part for this thesis is:

- “The processing time $p_{i,j}$: This defines how long job j needs to be processed on e machine i”
- “The release date r_j: This defines, when a job arrives at the system, i.e. the earliest time for the job to be processed.”
- “The finishing time $C_{i,j}$: This indicates, at what time job j will finish on machine i”[1, p. 14]

The α which is of interest for us is Fm, as it stands for a Flow Shop with m machines. Pinedo describes this problem as: “There are m machines in series. Each job has to be processed on each one of the m machines. All jobs have to follow the same route, i.e., they have to be processed first on machine 1, then on machine 2, and so on. After completion on one machine a job joins the queue at the next machine. Usually, all queues are assumed to operate under the First In First Out (FIFO) discipline, that is, a job cannot “pass” another while waiting in a queue. If the FIFO discipline is in effect the Flow Shop is referred to as a permutation Flow Shop and the β field includes the entry prmu.”[1, p. 15]

An example for this would be a Flow Shop with two machines and two jobs. All of which have a processing time of 20 units at each machine. With equal wight, and no release dates. A plan for this would be to first send the first job through the machines, with the second one following as soon as the first one finished the first machine. With this, the maximal finishing time, which would be the minimization goal, would be 60 units.

This definition of Flow Shop must however be extended, as more features are used in the given Flow Shop problems. Therefore, additional features of the problem will be introduced.

3.1.2 Flow Shop with operators

One important addition needed for the problem are operators They are needed to process a job at a machine, and without them, no processing can be performed. An example for this can be found in „Scheduling Flow Shops with operators“ [2] by Amène Benkalai, Djamal Rebaine and Pierre Baptiste.

This Flow Shop Problem, denoted as $Fm|res\,1k1, S, e|C_{max}$ introduces additional aspects to the Flow Shop problem:

- $res\,1k1$: This indicates the additional constraint of a resource. $1k1$ meaning, that there is one resource, of which there are k, and one is required for every step. This resource in this step are the operators, of which there are k in the problem, and one is needed for each action at a machine.
- S: This describes, that the problem “is subject to a fixed job sequence S”.[2, p. 2]
- e: This indicates an “end-of-operation changing mode” [2, p. 2], which means that an operator can not stop processing until completion. After that, the operator can start processing at the same or a different location, or remain idle.

Additionally, new notations are introduced in the paper. The important one for this thesis is the operation $S(i, j)$, which “corresponds to the i-th operation of the job at the j-th position on machine M_i”[2, p. 2]. As this gives the opportunity to directly label operations, it will be even more important later on.

With this, we can expand our running example, by setting $k = 1$, so that there is one operator, which needs to process the jobs. If we still have the two jobs and machines with the specified processing times, the maximal finishing time now would be 80 time units, as the jobs can not be worked on in parallel, because there is only one operator.

This is still not the Flow Shop problem which is supposed to be modeled in PDDL, so it again needs to be extended.

3.1.3 Extended Flow Shop with operators/workers

This part will extend the Flow Shop with operators problem. To get a notation that is consistent to the CSP problems later on, a few things will have to be renamed. From now on, operators will be called workers, jobs will be called products, and machines will be stations. This renaming is necessary, as most given Flow Shop problems use the former names, while the Flow Shop Problem modeled in CSP uses the newer names. Apart from that, multiple features to be added to the Flow Shop problem, and some features will have to be changed. These will be described in the following, together with their corresponding entry in the β field in the scheduling problem.

- Buffer: The first thing needed in addition are buffers. These are also introduced in Pinedos "Scheduling, Theory, Algorithms, and Systems" [1, p. 17] the β parameter *block* in a scheduling problem. It describes the constraint, when there is a limited buffer between the stations. If some station finishes, processing a product, and the buffer after it has no available space, then the product has to remain on the station, meaning it can not start processing the next product.
  In this extended Flow Shop, the size of this buffer will always be 1, so that one product can be stored between two stations. Therefore, the entry in the β field for this is $block1$

- Release times: Similar to release dates in "Scheduling, Theory, Algorithms, and System" [1, p. 15], these describe, that some product can not be processed at a station, before its release date r_p. In addition, this also gets added to workers, meaning, that a worker w can not be used for any processing or walk to another station, before the workers release time r_w. Together, they can be used to model, that a worker is still working on a product at the start, by giving both the product and the worker the same release time.
  This will be denoted in the β field as $r_{j,w}$

- Walking times: Workers are now at specific positions, starting at specified starting positions for each worker. So, in contrast to the operators in the Flow Shop with workers, a worker needs to be at a station in order for the processing to take place. In addition to this, walking times are introduced. These are times needed by the workers to go from one station to another. In practice, this means, that if a worker finishes processing at some station and then wants to start processing at a different station, this walking time has to pass between the end of the first processing, and the start of the next one.
  It is specific to the distance between two stations, so that walking from one station to two different ones can take a different amount of time.
  As entry in the β field, wt is used.

- Product types: Each product has a specific type, which defines how long processing the product at each station takes. If there are t types of products in a problem, then each type has a number in $[1, \ldots, t]$. Two products with the same type will take the same time to be processed at a station, while two products of different types can take a different amount of time at the same machine.
  This will be represented by *types* in the β field.

- automatic/manual processing: The processing itself also changes. Instead of always having a product being processed at a station once together with a worker, there can now be a second option. The first one will remain as manual processing, but in addition, there is now an automatic option.
  If a product gets processed automatically at a station, then there are actually three steps. First, the setup part, where the product gets put from the previous buffer onto the station, which also needs a worker. Next up is the actual processing of the product, which in this case will be called automatic processing. As this is meant to mimic an automatic process, no worker is needed for this step. Lastly, there is the takedown part, in which the product gets taken from the station and gets put onto the next buffer. Similar to the setup part, this requires a worker.
  If a product gets processed automatically or manually at a station is defined by the product type, so that one station could process some product types automatically, and others manually, but the same product types will always be treated the same at a station.
  It has to be noted here, that a product can only be taken from the previous buffer and be put onto the station, after the station is completely finished with the takedown of the previous

product.
The Operation $S(i, j)$ operator gets extended here to $S(i, j, k)$, where i and j still represent the i-th operation of a product at the j-th station, and k represents which part of the processing is meant. For manual processing, k can only be 1, as there is only one part in the processing, but for automatic processing, k can be 1 to 3, 1 for the setup, 2 for the actual automatic processing and 3 for the takedown. To write down this in the β field, amp

- Variable starting positions: Products do not necessarily have to be processed at every station, but could also only need processing on later stations, meaning they get treated, as if they already passed some stations. This can be at any point, meaning that, for example, a product can have finished the setup at a station, so that the setup part is not necessary, but only the automatic processing and the takedown.
This will be denoted in the β field as vsp

- Worker skill: The last change is the worker skill. It is a number between 0 and 100, where 100 means, that the worker works as fast a s possible, and 0 means, that the worker needs infinite amounts of time for a workstep. A workstep here means either the manual processing and the three steps of the automatic processing.
For a workstep with the duration t and a worker skill x, we get the actuall processing thime p as:

$$p = \frac{100}{x} \cdot t$$

This gets noted in the β field as ws.

This is the final Flow Shop version, that gets used in this thesis. As a Scheduling problem, this will be written down as $Fm|res\,1k1, S, e, block1, wt, types, amp, vsp, ws|C_{max}$

We can again expand our running example here. In addition to the two stations(machine before), we now have three buffers, one before the first station, one between the stations, and one after the last station. We can set the release times for all workers and products to 0. As a walking time, we can use 10 units between the stations for the worker. As product types we can give both products the same product type. This product type get processed at the first station automatically, with a setup time of 10, a processing time of 80 and a takedown time of 10, and manually at the second station, with a processing time of 100. Variable stating positions can be used to put the first product on the buffer in the middle. We can also give the worker a skill of 100.

With this, a possible solution for our example would be for our worekr to start the automatic processing of product 2 at the first station, and during the automatic processing step, the worker would go to the second station, to manually process the first product. After this, the worker would return to the first station, to perform the takedown part of the process. After one last move to the second station, and a final manual process, both products are at the end. The makespan for this plan would be 250. This consists of the setup(10), the move to station 2(10), the manual processing(100), the move back to station 1(10), the takedown(10), the last move to station 2(10) and the second manual processing(100). The automatic processing of the second product at the first station can be ignored her, as it happens in parallel to the worker going to station 2 and manually processing the first product.

3.1.4 Time complexity

The time complexity of this Flow Shop problem is hard to define. The problem is, that this Flow Shop problem is different from most other Flow Shops. There are however simpler versions of Flow Shop, which are proven to be NP-hard. An example for this can be found in "Scheduling Theory. Multi-Stage System"[3]. They prove, that finding a sub-optimal solution for their Flow Shop is NP-hard. If we can show, that our flow shop can model theirs, then we have also shown, that our Flow Shop is at least NP-hard.

To do this, we need to ignore most of the extra features we added to our Flow Shop. In "Scheduling Theory. Multi-Stage System", a Flow Shop simply consists of multiple jobs, multiple machines in an order and processing times at different machines. To reduce our problem to this, we first need to remove the workers. This can be achieved, by having one worker at every station with a skill of 100. If this is the case, then no walking is needed for workers, and there is always a worker at a station for a process. Furthermore, as every worker has a skill of 100, the worker skill can also be ignored.

The variable starting positions are not a problem, as they also allow for every product to start at the beginning. As The simple model only has one processing at every machine, we can simply only use manual processes at every station. Also, As every product can have different processing times in the simple Flow Shop model, we can simply use one product type for every product. Release times can just be set to 0 to ignore them.

The important last part are the buffer. These do not exist in the simple Flow Shop model. But as the simple Flow Shop does not enforce that a product must directly start processing at the next machine, if it finishes at the previous one, we can assume, that our product is at a buffer during that time. Our model is therefore able to model the simple Flow Shop model of "Scheduling Theory. Multi-Stage System". Because of this, our Flow Shop model must be at least NP-hard.

3.2 CSP

3.2.1 Basics of CSP

A Constraint satisfaction programm (short CSP) is a basic concept on how to describe and implement problems. The core idea is to have variables with domains, and constraints, which must be met in order for a solution to be correct. A basic definition for CSP can be found in “Artificial Intelligence A Modern Approach”[4] by Stuart Russel and Peter Norvig. It defines a constraint satisfaction problem as follows:

Definition 2 *A Constraint satisfaction problem consists of three components, X, D, and C:*

- *X is a set of variables $\{X_1, \dots, X_n\}$*
- *D is a set of domains $\{D_1, \dots, D_n\}$, one for each variable.*
- *C is a set of constraints that specify allowable combinations of values.*

Each domain D_i consists of a set of allowable values, $\{v_1, \dots, v_k\}$ for variable X_i. Each constraint C_i consists of a pair $\langle scope, rel \rangle$, where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation *and enumerating the members of the relation. For example, if X_1 and X_2 both have the domain $\{A, B\}$, then the constraint saying the two variables must have different values can be written as $\langle(X1, X2), [(A, B), (B, A)]\rangle$ or as $\langle(X1, X2), X1 \neq X2\rangle$" [4, p.202f]*

In addition, we write $C\{x_i, \ldots, x_j\}$ for the constraint $c \in C$, so that the variables in the scope of c are $\{x_i, \ldots, x_j\}$.

To find a solution for a CSP, we need to define what an assignment is.

Definition 3 *Let $\gamma = (X, D, C)$ be a CSP.*
An assignment for γ is a function $a : X' \to \cup_{\{x \in X\}} D_x$, where $X' \subseteq X$, and we require, that $a(x) \in D_x$ for every $x \in X$.
If $X' = X$, then a is a total assignment, otherwise it is a partial assignment.[5, p.22]

Additionally, we need to define consistency.

Definition 4 *Let $\gamma = (X, D, C)$ be a CSP, and a be an assignment for γ.*
If there are variables $x_i, \ldots, x_j$, so that $C\{x_i, \ldots, x_j\} \in C$, and the tuple $(a(x_i), \ldots, a(x_j))$ is not part of the relation of $C\{x_i, \ldots, x_j\}$, then the assignment a is inconsistent.
If a is not inconsistent, then it is consistent.[5, p.22]

We can now use this, to define a solution for a CSP.

Definition 5 *Let $\gamma = (X, D, C)$ be a CSP, and a be an assignment for γ.*
If a is a total assignment and also consistent, then a is a solution for γ.[5, p.24]

To understand these definitions an example is helpful. Our example is $\gamma = (X, D, C)$, where $X = \{a, b, c\}$, $D = \{\{1, 2, 3\}, \{1, 2, 3\}, \{1, 2, 3\}\}$ and C allows any combinations, so that the value of any variable must be different from every other variable. This means, that we have the three variables a, b and c, each of them can have valus 1, 2 or 3, and they must all be distinct. One solution for this would be the assignment $a = 1, b = 2, c = 3$, as this is a consistent total assignment for the CSP.

In addition, objectives can be added on top of a CSP. They define something, that should be tried when solving the CSP. A possible objective for the example would be, to minimize the value of c. In this case, our solution from before would still be a correct solution, but when trying to optimize this objective, the solution $a = 3, b = 2, c = 1$ would be better.

3.2.2 Systems used for CSP

For this thesis, two CSP models of Flow Shop will be used. They all implement for different systems, but all encode the same Flow Shop problem described above. The systems are:

- Google OR-Tools CP-SAT solver[6]: The OR-Tools solver by google is a state of the art tool, which includes solvers for CSP. The one used here is the CP-SAT solver. It can, for example, be used in python, to create a model, add all necessary variables and constraints, and then find a solution.
  It works, by using propagation, meaning assigning one variable a specific value and then adding restrictions on the other not yet assigned variables, and backtracking to previous steps, to find all possible solutions[7].

- CPLEX[8]: CPLEX is a system developed by IBM, which can solve multiple types of problems, including CSPs. To define the model, the OPL modeling language is used.

These are two systems, for which Flow Shop models are already given. An additional model exists for MiniZinc[9]. This model did however not work, and was therefore not used. The used models can be found on github[10]. They were not created by the author of this thesis.

3.2.3 Flow Shop in CSP

Flow shop problems are described in both of these systems. In both of them, this description has two main parts:

- The model itself, which defines, how a general Flow Shop problem should look like. It defines, which variables will be created with which domains, and which constraints will be between the variables.
- The instance, which gives the exact instance of the Flow Shop model, so, for example, the number of Products and workers, what the processing times are and so on. It is important to note, that for CP-Sat solver of google, this is given in two separate *.xml files, the configuration file, which contains general information about the amount of stations and their location, and the initial file, which contains information about the workers and products.

As both systems follow this general rule, and the implementation of Flow Shop in CSP is not the focus of this thesis, but the implementation in PDDL, only one of them will be discussed in more detail, which will be the CP-SAT solver model.

As example for the initial file, the “W1-Ex1.xml” file will be used. It looks like this:

```
<?xml version="1.0"?>
-<FlowShopInitialState name="W1-Ex1">
    <ConfigurationFile name="Example"/>
    -<Workers>
        <Worker releasetime="0" location="STATION-1" skill="100" id="WORKER-1"/>
    </Workers>
    -<Products>
        <Product id="PRODUCT-1" variant="VARIANT-1"/>
        <Product id="PRODUCT-2" variant="VARIANT-1"/>
        <Product id="PRODUCT-3" variant="VARIANT-1"/>
        <Product id="PRODUCT-4" variant="VARIANT-1"/>
        <Product id="PRODUCT-5" variant="VARIANT-1"/>
    </Products>
</FlowShopInitialState>
```

We can see, that it contains the following information about Flow Shop:

- There is one worker, which starts at station1, without release time, and has a skill of 100.
- There are five products, all of which are of variant1.

Next up, a look at the “S5-Ex29” file shows, what a configuration file looks like. It looks like this:

```
<?xml version="1.0"?>
-<FlowShopConfiguration name="S5-Ex29">
    -<WalkingTimes>
        <WalkingTime traveltime="0" end="STATION-1" start="STATION-1"/>
        <WalkingTime traveltime="10" end="STATION-1" start="STATION-2"/>
        <WalkingTime traveltime="20" end="STATION-1" start="STATION-3"/>
        <WalkingTime traveltime="30" end="STATION-1" start="STATION-4"/>
        <WalkingTime traveltime="40" end="STATION-1" start="STATION-5"/>
        <WalkingTime traveltime="10" end="STATION-2" start="STATION-1"/>
        <WalkingTime traveltime="0" end="STATION-2" start="STATION-2"/>
        <WalkingTime traveltime="10" end="STATION-2" start="STATION-3"/>
        <WalkingTime traveltime="20" end="STATION-2" start="STATION-4"/>
        <WalkingTime traveltime="30" end="STATION-2" start="STATION-5"/>
        <WalkingTime traveltime="20" end="STATION-3" start="STATION-1"/>
        <WalkingTime traveltime="10" end="STATION-3" start="STATION-2"/>
        <WalkingTime traveltime="0" end="STATION-3" start="STATION-3"/>
        <WalkingTime traveltime="10" end="STATION-3" start="STATION-4"/>
        <WalkingTime traveltime="20" end="STATION-3" start="STATION-5"/>
        <WalkingTime traveltime="30" end="STATION-4" start="STATION-1"/>
        <WalkingTime traveltime="20" end="STATION-4" start="STATION-2"/>
        <WalkingTime traveltime="10" end="STATION-4" start="STATION-3"/>
        <WalkingTime traveltime="0" end="STATION-4" start="STATION-4"/>
        <WalkingTime traveltime="10" end="STATION-4" start="STATION-5"/>
        <WalkingTime traveltime="40" end="STATION-5" start="STATION-1"/>
        <WalkingTime traveltime="30" end="STATION-5" start="STATION-2"/>
        <WalkingTime traveltime="20" end="STATION-5" start="STATION-3"/>
        <WalkingTime traveltime="10" end="STATION-5" start="STATION-4"/>
        <WalkingTime traveltime="0" end="STATION-5" start="STATION-5"/>
    </WalkingTimes>
    -<Variants>
        -<Variant id="VARIANT-1">
            -<WorkSteps>
            -<WorkStep id="WORKSTEP-V1-S1"
            automatic="False" location="STATION-1">
                <Duration>100</Duration>
            </WorkStep>
            -<WorkStep id="WORKSTEP-V1-S2"
            automatic="False" location="STATION-2">
                <Duration>100</Duration>
            </WorkStep>
            -<WorkStep id="AUTOWORKSTEP-V1-S3"
            automatic="True" location="STATION-3">
                <Duration>80</Duration>
                <SetupTime>10</SetupTime>
                <TakedownTime>10</TakedownTime>
            </WorkStep>
            -<WorkStep id="WORKSTEP-V1-S4"
            automatic="False" location="STATION-4">
                <Duration>100</Duration>
            </WorkStep>
            -<WorkStep id="WORKSTEP-V1-S5"
            automatic="False" location="STATION-5">
                <Duration>100</Duration>
            </WorkStep>
            </WorkSteps>
        </Variant>
    </Variants>
    -<ProcessingUnits>
        <ProcessingUnit id="STATION-1" position="1"
        lane="1" type="STATION"/>
        <ProcessingUnit id="BUFFER-1" position="1"
        lane="1" type="BUFFER" capacity="1"/>
        <ProcessingUnit id="STATION-2" position="2"
        lane="1" type="STATION"/>
        <ProcessingUnit id="BUFFER-2" position="2"
        lane="1" type="BUFFER" capacity="1"/>
        <ProcessingUnit id="STATION-3" position="3"
        lane="1" type="STATION"/>
        <ProcessingUnit id="BUFFER-3" position="3"
        lane="1" type="BUFFER" capacity="1"/>
        <ProcessingUnit id="STATION-4" position="4"
        lane="2" type="STATION"/>
        <ProcessingUnit id="BUFFER-4" position="4"
        lane="2" type="BUFFER" capacity="1"/>
        <ProcessingUnit id="STATION-5" position="5"
        lane="2" type="STATION"/>
    </ProcessingUnits>
</FlowShopConfiguration>
```

We can see the following:

- The traveling times are defined between each station will every station (including with itself).
- There is one variant which has five work steps, where only the step at station 3 is automatic, the other ones are manual.
- There are five stations, and four buffers.

These files together fully describe, what elements are part of the exact Flow Shop problem. Now we can look, how the Flow Shop model gets designed. The code, which creates the model and then searches for a solution in in written in python.

First up, the variables of the CSP need to be defined. In this model, there are two types of variables. The first type are time variables. One exists for every start and end of every workstep. Their domain goes from 0 up to a horizon that is defined in the code of the model. This horizon is therefore the latest time something can happen in the model, and is set to 1000000 in the code. In addition to these variables, boolean variables are added to describe, if a worker performs some processing. That means, for every possible workstep and worker, there is a boolean variable, that is supposed to be true, if the worker does perform this workstep.

These are all variables used in the model. The constraints will not be discussed in full detail, as that is not the main focus of this thesis. As an example, how the constraints work, we can look at the first one. In enforces, that the end time of each work step is equal to the start time of it plus the duration of the work step. This achieves, that each work step has the correct duration.

Lastly,the the objective has to be added. As for Flow Shop, this objective is to have a makespan as small as possible, the objective added in the model is to minimize the end times of the last product at the last station.
For this model, a solution can then be found. A CSP solution is a consistent assignment to all of the variables mentioned above. A different output is generated, to make this solution more readable. It consists of the following parts:

- Events for workers, that describe, which worker has to be at which station during which time intervals. It is important to note, that, while these intervals are definitely possible, so that no worker needs to be at two positions at the same time, and that there is always enough time between the intervals to allow the worker to walk from one station to another, only the times where the worker actually has to be at a station are in the solution. Times between this, including the walking times, are not part of this.

- Events for the products. They mark, where a product is during a time interval. A product can be either at a station or at a buffer in these events.

- The worksteps for each worker. They show, which worksteps a worker does, and at which time.

This is how the FLow Shop Problem is modeled with the Google OR-Tools, and how the solution is presented. The CPlex model works in a similar way.

3.3 Planning and PDDL

3.3.1 A Planning Problem

The goal of this thesis is, to implement the Flow Shop problem as a planning problem, and comparing the results with the CSP implementations. Therefore a definition of planning problems is needed. For reference, the book “Automatic Planning and Acting” [11] by Malik Ghallab, Danu Nau and Paolo Traverso will be used.
Firstly, we will introduce a state-transition-system. In ”Automatic Planning and Acting”, a state-transition-system is introduced as:

Definition 6 *A state-transition-system (also called a classical planning domain) is a triple $\Sigma = (S, A, \gamma)$ or 4-tuple $\Sigma = (S, A, \gamma, cost)$, where*

- *S is a finite set of states in which the system may be.*

- *A is a finite set of actions that the actor may perform*

- *$\gamma : S \times A \to S$ is a partial function called the prediction function or state-transition function. If (s, a) is in γ ’ domain (i.e., $\gamma(s, a)$ is defined), then a is applicable in s, with $\gamma(s, a)$ being the predicted outcome. Otherwise a is inapplicable in s.*

- *cost* : $S \times A \to [0, \inf)$ *is a partial function having the same domain as* γ*. Although we call it the cost function, its meaning is arbitrary: it may represent monetary cost, time, or something else that one might want to minimize. If the cost function isn't given explicitly (i.e., if* $\Sigma = (S, A, \gamma))$*), then* $cost(s, a) = 1$ *whenever* $\gamma(s, a)$ *is defined."[4, p. 25]*

This allows for a basic definition of Flow Shop, where every state of the state-transition-system represents represents one state in which the Flow Shop can be at every given time. Actions would be every possible change in the Flow Shop, so for example a worker walking somewhere or a job getting processed, where γ defines, how the Flow Shop looks like after the action.
As the definition states, the cost function may represent time, which is what we want to minimize in the Flow Shop problems. This is however not possible in our case, as this would not allow any concurrency of actions. We will therefore only use $\Sigma = (S, A, \gamma)$, and assume a cost of 1 for each action, so we assume uniform costs, as a normal cost function is not useful for us.

This is however not enough to fully describe the problem, as for a problem to be defined, we still need to know, what out state is at the beginning, and what goal we want to have at the end. For Flow Shop, the beginning would have to describe, where everything is at the beginning, so where the workers, jobs etc. are, if any workers or jobs need to get released, and so on. The goal would be for every job to be processed by every machine, so every job must be at the end. We therefore have to use a different approach. For this we introduce the STRIPS planning Task[12]:

Definition 7 *A STRIPS planning task is a 4-tuple* $\Pi = (P, A, I, G)$*, where*

- *P is a finite set of facts.*
- *A is a finite set of actions. Each action* $a \in A$ *is a triple* $a = (pre_a, add_a, del_a)$*, where each of these is a subset of P. They get refered to as precondition, add list and delete list. It is required, that* $add_a \cap del_a = \emptyset$*.*
- $I \subseteq P$ *is the initial state.*
- $G \subseteq P$ *is the goal.*

Again, We assume unit cost, meaning all actions have a cost of 1.

This definition of a STRIPS planning problem now allows us to also define the STRIPS state space[12]:

Definition 8 *Let* $\Pi = (P, A, I, G)$ *be a STRIPS planning problem. The state space of* Π *is* $\Theta_\Pi = (S, A, T, I, S^G)$ *where*

- *S are the states of the state space, with* $S = 2^P$*, so all possible combination of facts*
- *A is the action set of* Π
- *The transitions of the state space are* $T = \{s \xrightarrow{a} s' | pre_a \subseteq s, s' = (s \cup add_a) \setminus del_a\}$
- *I is the initial state of* Π
- *The goal states* $S^G = \{s \in S | G \subseteq s\}$

This definition is similar to the state-transition-system, with the addition of the initial state and the goal states. As we now have a state space, we can also define what a plan looks like. We want a plan to be a sequence of actions, which lead us from the initial state to a goal state. Formally this is defined as:

Definition 9 *Let $\Theta_\Pi = (S, A, T, I, S^G)$ be a STRIPS state space. Then*

- $\pi =< a_1, \dots, a_n >$

is a plan for Θ_Π over the states $\{s_0, \dots, s_n\}$, if:

- *Every $a \in \pi$ is an action of Θ_Π*
- *For every $i \in \{1, \dots, n\} : pre_{a_i} \subseteq s_{i-1}$*
- *For every $i \in \{1, \dots, n\} : s_i = (s_{i-1} \cup add_{a_i}) \setminus del_{a_i}$*
- $s_0 = I$
- $s_n \in S^G$
- The length n of a plan is $|\pi|$, so the amount of actions in the plan.
- The cost c of a plan is the sum of the cost of the actions in the plan. As we assume uniform cost, $n = c$.
- An optimal plan is a plan with minimal cost, meaning there is no other plan, that has lower cost.

3.3.2 Defining a planning problem with PDDL

With a planning problem, together with its state space and plans now defined, we need to introduce, how such problems are encoded. A standard language for this is PDDL[13], the "Planning Domain Description Language". It is a widely used language to define planning problems, one example use of it is the International Planning Competition. The core idea of the language is, to separate a problem into two files, a domain and a problem file.

PDDL domain file

The domain file of PDDL defines, what is part of the planning problem, and what actions there are. There are many things which can be added to it, but this definition will focus on the needed parts of PDDL. The Paper "PDDL - The Planning Domain Definition Language"[13] gives a formal definition for the domain as a EBNF. We will only introduce the needed parts here.

```
<domain>                       ::=    (define (domain <name>)
                                      [<require-def>]
                                      [<types-def>]
                                      [<predicates-def>]
                                      <structure-def>*)

<require-def>                  ::= (:requirements <require-key>)
<types-def>                    ::= (:types <typed list (name)>)
<predicates-def>               ::= (:predicates <atomic formula skeleton>+)
<atomic formula skeleton>      ::= <predicate> <typed list (variable)>
<predicate>                    ::= <name>
<variable>                     ::= ?<name>
<structure-def>                ::= <action-def>
```

All of these arguments, with the exception of (domain ...) can be in any order. <require-key> can be a list of requirements, which will not fully be described here, the important one is :typing, which enables <types-def>.
Regarding the fields themselves, the paper gives the following explanations:

“The :requirements field is intended to formalize the fact that not all planners can handle all problems statable in the PDDL notation If the requirement is missing, [...], then it defaults to :strips.”[13, p. 6]

“The :types argument uses a syntax borrowed from Nisp that is used elsewhere in PDDL (but only if :typing is handled by the planner)

```
<typed list (x)>        ::= x*
<typed list (x)>        ::= x+ - <type> <typed list (x)>
<type>                  ::= <name>
```

“A typed list is used to declare the type of a list of entries; the types are preceded by a minus sign ("-"), and every other element of the list is declared to be of the first type that follows it, or object, if there are no types that follow it.[...]”[13, p. 6]

“The :predicates" field consists of a list of declarations of predicates, once again using the typed-list syntax to describe the arguments of each one.”[13, p. 7]

The EBNF for an action again gets defined as:

```
<action-def>            ::= (:action <action function>
                                 :parameters ( <typed list (variable)>)
                                 <action-def-body>
                             )
<action function>       ::= <name>
<action-def body>       ::= [:precondition <GD>]
                             [:effect <effect>]
```

To the fields themself, the paper states:

“The :parameters list is simply the list of variables on which the particular rule operates, i.e. its arguments, using the typing syntex described above”[13, p. 7]

“The :preconditions is an optional goal description (GD) that must be satisfied before the action is applied. [...] If no preconditions are specified, then the action is always executable.”[13, p. 7]
(GD) gets defined as:

```
<GD>                ::= <atomic formula(term)>
<GD>                ::= <and <GD>*>
<GD>                ::= <literal(term)>

<literal(t)>        ::= <atomic formula(t)>
<literal(t)>        ::= (not <atomic formula(t)>)
```

```
<atomic formula(t)> ::= (<predicate> t*)
<term>              ::= <name>
<term>              ::= <variable>
```

“where, of course, an occurrence of a <predicate> should agree with its declaration in terms of numbers and, when applicable, types of arguments. Hopefully the semantics of these expressions is obvious.”[13, p. 8f]

The description of the effects is given as:

```
<effect>        ::= (and <effect>*)
<effect>        ::= (not <atomic formula(term)>)
<effect>        ::= <atomic formula(term)>
```

“As in STRIPS, the truth value of predicates are assumed to persist forward in time. Unlike STRIPS, PDDL has no delete list - instead of deleting (on a b) one simly asserts (not (on a b)). If an action’s effects does not mention a predicate P then the truth of that predicate is assumed unchanged by an instance of the action.”[13, p. 9]

PDDL problem file

Now, we need to define the problem file. We can again use the definition in “PDDL - The Planning Domain Definition Language”[13]. We cut parts that are not needed in this thesis:

```
<problem>               ::= (define (problem <name>)
                            (:domain <name>)
                            [<object declaration>]
                            [<init>]
                            <goal>
                            )

<object declaration>    ::= (:objects <typed list (name)>)
<init>                  ::= (:init <literal(name)>+)
<goal>                  ::= (:goal <GD>)
```

:objects is a (typed) list of objects in the problem, and :init are the initial predicates that are true.
“A solution to a problem is a series of action such that (a) the action sequence is feasible starting in the given initial situation; (b) the :goal is true in the situation resulting from executing the action sequence[...]”[13, p. 19]
With this, we have a formal definition for a problem in PDDL.

PDDL example: Moving with trucks

This is an example of a domain and problem file for PDDL. In this task, A truck can drive between positions and load and unload packages at the positions. The goal is to bring the packages to specific positions.
We get the domain file:

```
(define (domain logistics)

    (:requirement :typing)

    (:types
        package
        truck_position
    )

    (:predicates
        (truck_at ?truck - truck_position ?position - truck_position)
        (package_at ?package - package ?position - truck_position)
        (connected ?p1 - truck_position ?p2 - truck_position)
        (truck_unloaded ?truck - truck_position)
    )

    (:action drive
        :parameters (?from - truck_position  ?to - truck_position
        ?truck - truck_position)
        :precondition (and (truck_at ?truck ?from) (connected ?from ?to))
        :effect (and (not (truck_at ?truck ?from)) (truck_at ?truck ?to))
    )

    (:action load
        :parameters (?position - truck_position ?truck - truck_position
        ?package - package)
        :precondition (and (truck_at ?truck ?position)
        (package_at ?package ?position) (truck_unloaded ?truck))
        :effect (and (not (package_at ?package ?position))
        (package_at ?package ?truck) (not (truck_unloaded ?truck)))
    )

    (:action unload
        :parameters (?position - truck_position ?truck - truck_position
        ?package - package)
        :precondition (and (truck_at ?truck ?position)
        (package_at ?package ?truck))
        :effect (and (not (package_at ?package ?truck))
        (package_at ?package ?position) (truck_unloaded ?truck))
    )
)
```

We see:

- We have typing as the only requirement.
- We have the types package and truck_position, the second one is used to say, that something is a position or a truck. This has to be defined this way, to allow a package to be at the truck and to be at a position with the same predicate.

- Our predicates are:
  - (truck_at ?truck - truck_position ?position - truck_position): This is used to define, where a truck is.
  - (package_at ?package - package ?position - truck_position): This is used to define, where a package is.
  - (connected ?p1 - truck_position ?p2 - truck_position): This is used to describe, that two positions are connected, so that the truck can drive from the first one to the second one.
  - (truck_unloaded ?truck - truck_position): This is used to define, that the truck is unloaded, and can therefore load a package.
- We have three actions:
  - drive, which can let a truck drive from one position to another, if the two positions are connected.
  - load, which lets the truck load a package, if the truck and package are at the same position, and the truck is unloaded.
  - unload, which lets the truck unload a currently loaded package at its current position.

The problem file looks like this:

```
(define (problem logistics)
    (:domain logistics)

    (:objects
        truck A B C D E F G - truck_position
        package1 package2 - package
    )

    (:init
        (truck_at truck A)

        (package_at package1 C)
        (package_at package2 E)

        (truck_unloaded truck)


        (connected A B)
        (connected B A)

        (connected B D)
        (connected D B)

        (connected C D)
        (connected D C)

        (connected D E)
        (connected E D)
```

```
            (connected D G)
            (connected G D)

            (connected G F)
            (connected F G)
        )

        (:goal
            (and
                (package_at package1 F)
                (package_at package2 F)
            )
        )
    )
```

We see:

- the domain name is the same as the one in the domain file.
- We have 10 objects, 1 truck, 7 positions and 2 packages.
- Initially, the truck is at A, the packages are at C and E respectively, the truck is unloaded, and some positions are connected.
- The goal is a conjunction, which states, that both packages must be at F.

With this, we can compute an optimal plan for this problem. The plan consists of driving to one package, then delivering it to position F, and after that, driving back to the other package and delivering it to F. Which order in which the packages get delivered is arbitrary, as both options result in plans of the same length. An optimal plan looks like this:

```
drive(A, B),
drive(B, D),
drive(D, C),
load(C, truck, package1),
drive(C,D),
drive(D,G),
drive(G,F),
unload(F, truck, package1),
drive(F, G),
drive(G, D),
drive(D, E),
load(E, truck, package2),
drive(E, D),
drive(D, G),
drive(G, F),
unload(F, truck, package2)>
```

In this, $drive(A, B)$ is the drive action, so that the first parameter of the action is the object A, and the second parameter is the object B.

3.3.3 Needed extensions to PDDL: PDDL2.1

With the currently introduced features of PDDL, we get a problem. We can not simply use the cost as a measure for time, as actions in Flow Shop, for example worker going from one station to another, can happen concurrently. If we assume two workers are walking from one station to another, each taking 50 units of time, then after both arrive, 50 units of time have passed, as they both walked at the same time. If we model this as actions with cost 50, then taking both of them has a cost of 100. We therefore need a different way to implement time consumption with PDDL. PDDL2.1[14] offers these features.

PDDL2.1 adds two requirements which we need to enable concurrent actions in a way we need them, the :durative-action requirement and the :fluents requirement. With this, we have to revise the definitions for domain and problem we gave above:

PDDL2.1 domain file

First, we have to change the domain itself. We now have:

```
<domain>                        ::=     (define (domain <name>)
                                        [<require-def>]
                                        [<types-def>]
                                        [<predicates-def>]
                                        [<functions-def>]
                                        <structure-def>*)

<require-def>                   ::= (:requirements <require-key>)
<types-def>                     ::= (:types <typed list (name)>)
<predicates-def>                ::= (:predicates <atomic formula skeleton>+)
<atomic formula skeleton>       ::= <predicate> <typed list (variable)>
<functions-def>                 ::= (:functions <atomic function skeleton>+)
<atomic function skeleton>      ::= (<function> <typed list (variable)>)
<predicate>                     ::= <name>
<function>                      ::= <name>
<variable>                      ::= ?<name>
<structure-def>                 ::= <action-def>
<structure-def>                 ::= <durative-action-def>
```

Here, we add functions, which are similar to predicates, but they can take a value and can not just be true of false. They get initialized in the problem file. Additionally, we get <durative-action-def>, which we now define:

```
<durative-action-def>        ::= (:durative-action <durative action function>
                                     :paramets ( <typed list (variable)>)
                                     :duration (= ?duration <untyped function skeleton>)
                                     <durative-action-def body>
                                 )
<durative action function>   ::= <name>
<untyped function skeleton>  ::= (<function> <list (variable)>)
<durative-action-def body>   ::= [:conditions <timed-GD>]
                                 [:effect <timed-effect>]
```

We see, that we need to define some additional things. First up, a normal list, which behaves like the typed list, but does not allow typing:

```
<list (x)>                ::= x*
<list (x)>                ::= x+ <list (x)>
```

Next up, we need to redefine the goal description GD to the timed goal description timed-GD nad the same for the effect, so to timed effects. For these, we have:

```
<timed-GD>                    ::= <timed-atomic formula(term)>
<timed-GD>                    ::= <and <timed-GD>*>
<timed-GD>                    ::= <timed-literal(term)>

<timed-atomic formula (t)>  ::= (at start <atomic formula(t)>)
<timed-atomic formula (t)>  ::= (at end <atomic formula(t)>)
<timed-atomic formula (t)>  ::= (overall <atomic formula(t)>)

<timed-literal (t)>           ::= (at start <literal(t)>)
<timed-literal (t)>           ::= (at end <literal(t)>)
<timed-literal (t)>           ::= (overall <literal(t)>)

<literal(t)>                  ::= <atomic formula(t)>
<literal(t)>                  ::= (not <atomic formula(t)>)

<atomic formula(t)>           ::= (<predicate> t*)
<term>                        ::= <name>
<term>                        ::= <variable>
```

and:

```
<timed-effect>          ::= (and <effect-timing>*)


<effect-timing>          ::= (at start <effect>)
<effect-timing>          ::= (at end <effect>)


<effect>                ::= (not <atomic formula(term)>)
<effect>                ::= <atomic formula(term)>
```

These changes add the durative-actions, which behave similar to normal actions. The difference is, that they have a :duration, which gets defined by a function. The big change are for the :preconditions, which are now :conditions, and the :effect. Each predicate has as predecessor "at start", "at end" or, for the conditions, "overall".

For the :conditions, this explains, when the predicate must be true, so for "at start", the predicate must be true at the beginning of the action, for "at end" at the end of the action and for "overall", it must be true for the whole duration of the action.

For the :effect, “at start” means, that the changes are applied at the start of the action, and “at end” means, that the changes are applied at the end of the action, so after the duration.

Because the actions now have a duration, multiple actions can happen in parallel, as long as the :conditions are met.

PDDL2.1 problem file

We also need to change the definition of the problem file. We get:

```
<problem>                   ::= (define (problem <name>)
                                (:domain <name>)
                                [<object declaration>]
                                [<init>]
                                <goal>
                                <metric>
                                )

<object declaration>        ::= (:objects <typed list (name)>)
<init>                      ::= (:init <literal(name)>+ <function>+)
<function>                  ::= (= <untyped function skeleton> <integer>)
<goal>                      ::= (:goal <GD>)

<metric>                    ::= maximize <numeric-operation>
<metric>                    ::= minimize <numeric-operation>

<numeric-operation>         ::= <numeric-operation> + <numeric-operation>
<numeric-operation>         ::= <numeric-operation> - <numeric-operation>
<numeric-operation>         ::= <numeric-operation> * <numeric-operation>
<numeric-operation>         ::= (total-time)
<numeric-operation>         ::= (number)
```

The metric describes a parameter of the problem, that is tried to be minimized or maximized. A normal matric in this case would be to minimize the total time, so (:metric minimize (total-time)).

PDDL2.1 plans

As we now additionally have durative actions, we have to adjust our definition of a plan. What we need to add is the start and end time of each action, and we also need to account for the different type of conditions and effects, so for example, if a condition must be true at the start of an action, or the end, or during the whole action. We write a plan as:

```
<[start-time_1] action_1 [end-time_n]
\dots
[start-time_n] action_n [end-time_n]>
```

We have the following conditions for such a plan:

- The start times are in ascending order, so that $start - time_i \leq start - time_j$ for $i < j$.
- For normal action $action_i$, we write down, that $start - time_i = end - time_j$.
- For the effects of $action_i$, that effects with "at start" change at $start - time_i$ and effects with "at end" change at $end - time - i$. The changes of normal action $action_j$ take place at $start - time_j$, because the start and end time are the same for normal actions, so it does not matter, if we pick the start or end time.
- For the conditions of durative actions, we have three possibilities.
  - If a condition of $action_i$ is an "at start" condition, then it must be true at $start - time_i$.
  - If a condition of $action_i$ is an "at end" condition, then it must be true at $end - time_i$.
  - If a condition of $action_i$ is an "overall" condition, then it must be true at all times between $start - time_i$ and $end - time_i$.
- For normal action $action_j$, as $start - time_j = end - time_j$, we require that every precondition is true at $start - time_j$.

PDDL2.1 example: Moving with trucks and durative-actions

As an example, we can adapt the previous example to use durative-actions. We get for the domain file:

```
(define (domain logistics)

    (:requirements :typing :durative-actions :fluents)

    (:types
        package
        truck_position
    )

    (:predicates
        (truck_at ?truck - truck_position ?position - truck_position)
        (package_at ?package - package ?position - truck_position)
        (connected ?p1 - truck_position ?p2 - truck_position)
        (truck_unloaded ?truck - truck_position)
    )

    (:functions
        (drive-time ?from - truck_position ?to - truck-position)
        (load-time ?package - package)
        (unload-time ?package - package)
    )

    (:durative-action drive
        :parameters (?from - truck_position  ?to - truck_position
        ?truck - truck_position)
        :duration (= ?duration (drive-time ?from ?to))
        :condition (and (at start (truck_at ?truck ?from))
        (at start (connected ?from ?to)))
        :effect (and (at start (not (truck_at ?truck ?from)))
        (at end (truck_at ?truck ?to)))
    )

    (:durative-action load
        :parameters (?position - truck_position ?truck - truck_position
        ?package - package)
        :duration (= ?duration (load-time ?package))
        :condition (and (overall (truck_at ?truck ?position))
        (overall (package_at ?package ?position))
        (at start (truck_unloaded ?truck)))
        :effect (and (at start (not (package_at ?package ?position)))
        (at end (package_at ?package ?truck))
        (at start (not (truck_unloaded ?truck))))
    )

    (:durative-action unload
        :parameters (?position - truck_position ?truck - truck_position
        ?package - package)
        :duration (= ?duration (unload-time ?package))
        :condition (and (overall (truck_at ?truck ?position))
        (at start (package_at ?package ?truck)))
        :effect (and (at start (not (package_at ?package ?truck)))
        (at end (package_at ?package ?position))
        (at end (truck_unloaded ?truck)))
    )
)
```

The problem file changes to:

```
(define (problem logistics)
    (:domain logistics)

    (:objects
        truck A B C D E F G - truck_position
        package1 package2 - package
    )

    (:init
        (truck_at truck A)

        (package_at package1 C)
        (package_at package2 E)

        (truck_unloaded truck)


        (connected A B)
        (connected B A)

        (connected B D)
        (connected D B)

        (connected C D)
        (connected D C)

        (connected D E)
        (connected E D)

        (connected D G)
        (connected G D)

        (connected G F)
        (connected F G)

        (= (drive-time A A) 0)
        (= (drive-time A B) 10)
        ...

        (= (load-time package1) 15)
        (= (load-time package2) 20)

        (= (unload-time package1) 5)
        (= (unload-time package2) 10)
    )

    (:goal
        (and
            (package_at package1 F)
            (package_at package2 F)
        )
    )

    (:metric minimize (total-time))
)
```

We can see, that the actions use the functions for durations, which are initialized in the problem file, and that they check their conditions and have their effect at different times during the action. The problem file only adds the functions initialization, and the metric, which tries to minimize the total time of the problem.

The optimal plan is the same as for the normal PDDL problem, but this time, we have to add the times for each action. It therefore looks like this:

```
<[0] drive(A, B) [10],
[10]drive(B, D)[20],
[20]drive(D, C)[25],
[25]load(C, truck, package1)[40],
[40]drive(C,D)[45],
[45]drive(D,G)[55],
[55]drive(G,F)[60],
[60]unload(F, truck, package1)[65],
[65]drive(F, G)[70],
[70]drive(G, D)[80],
[80]drive(D, E)[90],
[90]load(E, truck, package2)[110],
[110]drive(E, D)[120],
[120]drive(D, G)[130],
[130]drive(G, F)[135],
[135]unload(F, truck, package2)[145]>
```

This plan has a makespan of 145, as the last action ends at time step 145.

3.4 Related work

As one central goal of this thesis is to create a PDDL model that describes our version of Flow Shop, a look at past research is important. It can reveal how such a model could look like, and what design patterns work, and which do not.

3.4.1 Production Planning with IEC 62264 and PDDL

One example where PDDL was used to find a production plan can be found in "Production Planning with IEC 62264 and PDDL"[15]. Here, a PDDL model was generated from a metamodel described with the IEC 62264 standard, which described different production systems.
These production systems did differ from Flow Shop in many ways. For example, a system was used to describe, where the shuttles, which have a similar use as the workers in Flow Shop, could move. Furthermore, the shuttles were only responsible for transporting the products, the actual work was performed at tables by robots, which could not move.
An interesting approach is, that three different types of PDDL models were used. The first one used durative actions and is therefore similar to the planned PDDL model of Flow Shop. The second one did not use durative actions, but instead used normal actions, and the duration was encoded as the cost of these actions. Lastly, a PDDL model which did not use any costs was used, only the number of actions was considered.
The results did show, that such an automatic generation of PDDL models is possible, but that PDDL has problems finding solutions for the problems. The usage of sequential PDDL, so without the durative actions, did seem feasible for small and middle sized problems, but not for big problems. When using durative actions, even small problems were unfeasible. The reason for this is the high memory demand and general slow solution finding of temporal planners, according to the paper. A proposed solution for this is, to find a sequential solution, and then using some post processing to parallelize some actions.

3.4.2 Solving Flow Shop problems using a forward-chaining partial-order planner

A more closely related work is the bachelor thesis "Solving Flow Shop problems using a forward-chaining partial-order planner" [16] by Albin Billman. In it, he created a model for a Flow Shop problem, which did use durative actions.

His type of Flow Shop was however different from the one discussed in this thesis. Instead of workers, a single robot was part of the problem. This robot could move between stations. These stations were either the start, the end, of contained machines for processing. No buffers were present between the stations, and the robot had to manually transfer a product from one station to another. Still, it can be seen, that PDDL with durative actions can be used to find solutions for Flow Shop, but the results did show, that the picked planner can have a big effect on the results. As the planners did not perform optimal search, the makespan did differ, depending on the picked planner. The Temporal Fast Downward planner did have the best results in this regard, as the total makespan over all problems was 3480, while the COLIN planner had a total makespan of 4690.
These better results did however come at a cost, as TFD did need the most time for solving the problems. Therefore, when picking planners, the trade-off between finding a solution fast and finding a solution with a short makespan.

3.4.3 AI Planning Using Constraint Satisfaction Problems

A general different approach would be, to ignore the Flow Shop problem itself, but directly generate a PDDL model from the Constraint satisfaction problems, by using a general CSP to PDDL translator. The different direction of this, so generating a CSP problem from a given PDDL problem was discussed by Debby Nirwan[17].

The idea is, that there is a given PDDL problem, and a fixed plan length k. Then, for every time step $i \in \{1, \dots, k\}$, we have variables to encode, what predicates are true at each step, and a variable to encode, which action is taken at each step.
The example given for this is a domain, in which a robot can be at different location, and move between them. For variables that describe the predicates, one variable $at(i, rob)$ which has the domain consisting of all possible locations is enough to encode all possible states. In the example, there is another predicate that describes, if two locations are adjacent, but as this never changes, it does not have to be described by the variables.
The action variable for each step can be any possible action, and additionally, a no-op action, which is used to add the possibility of doing nothing, which is necessary, if the plan is shorter than k. In the example, this would mean, that the variable $act(i)$, which describes which action can be taken at time step i, can be any move action, or no-op.

Now, the actions, initial state and goal have to be modeled. The initial state is easy to model, as one can use an unary constraint, to enforce, that all variables for time step 0 are set to the value they are in the initial state of the PDDL problem. The same can be done for the goal, so that at time step k, all variables that describe the predicates must have the correct value according to the goal in the PDDL model.
The actions can be modeled, by enforcing that if some action is taken at step i, that all necessary preconditions must be true at step i. This would therefore be a constraint between the action variable of step i and all possible preconditions of actions. In the example, the constraint would be between the action variable $act(i)$ and the at variable $at(i, rob)$, and would for example include the tuple $(move(rob, loc1, loc2), loc1)$, meaning the robot can move from location 1 to location 2, if he is at location 1 at the time step.
The effects can be encoded similarly, by enforcing via constraint, that the variables that encode the values of the predicates at time step $i + 1$ must be correct, according to the effects of the action chosen at time step i.
With this, a PDDL model can be translated into a CSP model, for which solvers can be used. The problem is, that this is the wrong direction of the transformation. We want to translate a CSP model to PDDL and not the other way around.

3.4.4 The RoboCup Logistics League as a Benchmark for Planning in Robotics

The ”RoboCup logistic league”[18] in a competition, in which two teams, each consisting of multiple robots, compete against each other. They try to score more points, which can be earned by the production of objects. Which object should be constructed is announced by a referee box. To finish this production, the robots need to access different machines, to construct a correct product. It is therefore similar to our Flow Shop, because the machines have a strict order for a single product. The robots can be seen as the workers, traveling between the stations.

There are however multiple differences. One important difference are the multiple teams in the RoboCup LL. They compete against each other, and even though they try not to ram into each other, they do not have to make way for the other team. This competitive environment is in complete contrast to the Flow Shop introduced in this thesis, as the Flow Shop has no competing teams. It only has workers that try to achieve the same goal. There are other differences, like no worker skill, and no buffers between the stations. As it still has similarities to the Flow Shop problem of this thesis, it is still interesting.

In the paper ”The RoboCup Logistics League as a Benchmark for Planning in Robotics”[19], it was tried to simulate the RoboCup as a PDDL model. This PDDL model has characteristics that are also needed for the PDDL model of the Flow Shop problem, as the RoboCup and Flow Shop are similar. Some actions from the RoboCup model are however not needed for the Flow Shop model. An example for this would be the deliver action. In Flow Shop, the products change from station to buffers and the other way around with setup and takedown actions, and no special delivery is needed.

3.4.5 A System for Solving Constraint Satisfaction Problems with SMT

There have already been other experiments which tried to translate a CSP model into something different. One example to this can be found in the paper ”A System for Solving Constraint Satisfaction Problems with SMT”[20]. In it, a system gets introduces ”for translating FlatZinc (MiniZinc intermediate code) instances of constraint satisfaction problems to the standard SMT-LIB language”[20, p.1]. Even though SMT is not similar to PDDL, this shows, that CSP models can be translated into something different. The results did also show, that such a translation can show good results.

4 Flow Shop in PDDL

4.1 The PDDL model of Flow Shop

The PDDL model of the Flow Shop problem consists of two parts. Firstly, the domain file, which is the same for every instance of a Flow Shop problem. The second part is the problem file, which defines the characteristics of a specific instance. This chapter will introduce both of them and explain how they function.

As the domain file is the same for every instance, it can be fully shown. The first part of the domain consists of the domain name, the requirements, the types used, and the predicates:

```
(define (domain flow-shop)
(:requirements :durative-actions :fluents :typing)

(:types
    station - object
    buffer - object
    product - object
    worker - object
    variant - object
)

(:predicates
    (product-of-variant ?p - product ?v - variant)
    (end-buffer ?b - buffer)
    (worker-released ?w - worker)
    (product-released ?p - product)
    (worker-at-station ?w - worker ?s - station)
    (product-at-station ?p - product ?s - station)
    (product-at-buffer ?p - product ?b - buffer)
    (buffer-unused ?b - buffer)
    (station-unused ?s - station)
    (station-processing ?s - station)
    (station-not-processing ?s - station)
    (buffer-before-station ?b - buffer ?s - station)
    (buffer-after-station ?b - buffer ?s - station)
    (automatic-process ?s - station ?v - variant)
    (manual-process ?s - station ?v - variant)
    (product-before ?p1 - product ?p2 - product)
    (product-processed ?p - product ?s - station)
)
```

The domain is called flow-shop, but this name is not relevant for the functionality. The first functional part are the requirements. To implement the Flow Shop problem, durative actions and fluents are needed, while typing makes the model a lot clearer. The types of the model can be seen in the next part. They are: station, buffer, product, worker, and variant. Every object of the Flow Shop must be one of these.

The use of each object becomes more clearly, when looking at the predicates. Each of them has a specific use in the actions. The uses are:

- **product-of-variant**: This predicate indicates, that some product belongs to a variant. This is important in actions, when the variant of an product defines aspect of an a action like the duration. This predicate makes sure, that each product gets used together with its correct variant.
- **end-buffer**: This predicate indicates, that some buffer is an endbuffer. It is needed, as the endbuffer has special properties.
- **worker-released**: This predicate indicates, that a worker can work. It is either true from the beginning on, or can get set to true at the start of the plan. This has to be done, if the worker is working on something at the beginning of the Flow Shop. This predicate therefore prevents the worker from doing anything, until this time is over.
- **product-released**: This has the same functionality as worker-released. It prevents a product from being processed, if it is not yet free in the beginning.
- **worker-at-station**: A worker needs to be at a specific station for different actions at this station. Therefore, this predicate indicates, that a worker is at a specific station.
- **product-at-station**: Similar to the worker, a product must also be at a station for specific action. That is indicated by this predicate.
- **product-at-buffer**: If the worker is not at a station, but at a buffer, this predicate gets used to indicate its position.
- **buffer-unused**: As only one product can be at a buffer, an indication is needed to show, that no product is at a buffer yet. This predicate is used for that purpose.
- **station-unused**: Similar to buffer-unused, this predicate indicates, that no product is at a station at a given time.
- **station-processing**: This predicate indicates, if actual processing is happening at a station at a given time. This can then be used to indicate, if processing of another product can begin.
- **station-not-processing**: As using the negation of some predicates was a problem with multiple planners, the negation of station-processing needed to be added. It is always true, if station-processing is not true at some point in time.
- **buffer-before-station**: To have an order over the stations and buffers, predicates are needed. This one indicates, that some buffer is before a station.
- **buffer-after-station**: Similarly to the previous one, this predicate indicates, that a buffer is after a station.
- **automatic-process**: As a variant determines, if processing at a station is automatic or manual, a predicate is needed to indicate this. This is done by this predicate, which shows, that processing the specified variant at the specified station is automatic.

- **manual-process**: Similarly, if the process is manual and not automatic, this predicate is used.
- **product-before**: As the previous product must have been processed at a station, in order for a product to be processed there, an order over the products is needed. This is achieved by this predicate, that indicates, that the first product ?$p1$ is the one before ?$p2$.
- **product-processed**: To complete the requirement, of the previous product having been processed at a station, this predicate is used. It indicates, that a product has been processed at a station.

The domain file next defines the functions:

```
(:functions
    (release-time-worker ?w - worker)
    (release-time-product ?p - product)
    (walk-time ?w - worker ?from - station ?to - station)
    (setup-time ?w - worker ?s - station ?v - variant)
    (automatic-work-time ?w - worker ?s - station ?v - variant)
    (takedown-time ?w - worker ?s - station ?v - variant)
    (manual-work-time ?w - worker ?s - station ?v - variant)
)
```

These are used in the actions to define the duration. Each function defines the duration for a specific action and the specific parameters of the action, that define the duration. For example, as the walk time between station depends on the two stations and the worker-skill, the function uses two stations and a worker.

After this, each action is defined:

```
(:durative-action release-worker
:parameters
    (?w - worker)
:duration
    (= ?duration (release-time-worker ?w))
:condition
    (and
    (at start (not (worker-released ?w)))
    )
:effect
    (and
    (at end (worker-released ?w))
    )
)

(:durative-action release-product
:parameters
```

```
        (?p - product)
    :duration
        (= ?duration (release-time-product ?p))
    :condition
        (and
        (at start (not (product-released ?p)))
        )
    :effect
        (and
        (at end (product-released ?p))
        )
    )
```

release-worker and **release-product** are actions, that are supposed to be used at the start of the plan if needed. Flow Shop allows workers and products to be occupied at the beginning, and only be used after that. These actions are used to model this behaviour. If a worker or product is occupied at the beginning and not yet released, then the respective predicate is not yet true. In that case, these actions can be used, to set their respective predicate to true. The duration is therefore the releasetime of the specific product or worker. As the release of the worker or product happens at the end, it needs to be set to true at the end of the action.

```
    (:durative-action move
    :parameters
        (?from - station
        ?to - station
        ?w - worker)
    :duration
        (= ?duration (walk-time ?w ?from ?to))
    :condition
        (and
        (at start (worker-released ?w))
        (at start (worker-at-station ?w ?from))
        )
    :effect
        (and
        (at start (not (worker-at-station ?w ?from)))
        (at end (worker-at-station ?w ?to))
        )
    )
```

As the workers need to be able to travel between stations, a walk action is needed. The duration is given by a function and depends on the specific worker and the stations from which and to which the worker moves. In this, and all following actions, the worker using the action must be released. Therefore, this is a condition that applies to all following actions. In addition, to move away from a station, the worker needs to be at that station. Both of these need to be true at the start of the action.

If the action is used, then the worker is directly no longer at the station. This is needed, as otherwise, the worker could move from one station to another, and before arriving there, start another move action. When both of these are finished, the worker would be at two different stations at the same time, which would be wrong. Similarly, the worker being at the next station must be set to true at the end of the action.

The rest of the actions deal with the processing of products at stations.

```
(:durative-action automatic-setup
:parameters
    (?s - station
    ?b - buffer
    ?w - worker
    ?p - product
    ?v - variant
    ?pbefore - product)
:duration
    (= ?duration (setup-time ?w ?s ?v))
:condition
    (and
    (at start (product-of-variant ?p ?v))
    (at start (buffer-before-station ?b ?s))
    (at start (worker-released ?w))
    (at start (product-released ?p))
    (at start (station-unused ?s))
    (over all (worker-at-station ?w ?s))
    (at start (product-at-buffer ?p ?b))
    (at start (station-not-processing ?s))
    (at start (automatic-process ?s ?v))
    (at start (product-before ?pbefore ?p))
    (at start (product-processed ?pbefore ?s))
    )
:effect
    (and
    (at start (not(station-unused ?s)))
    (at end (not (product-at-buffer ?p ?b)))
    (at end (station-processing ?s))
    (at end (not (station-not-processing ?s)))
    (at end (product-at-station ?p ?s))
    (at end (buffer-unused ?b))
    )
)
```

The automatic-setup action is used for the first part of an automatic action. It takes a product from a buffer and puts it on the station. As parameters, it has the station where the processing takes place, the buffer before that station, the worker doing the work, the product getting processed, the variant of this product and the product before the processed product. This additional product is needed, to ensure, that the previous one was already processed.

The duration is dependent on the worker, the station and the variant of the product. The conditions make sure, that the setup process can take place at the time the action is taken, according to the Flow Shop definition. For example, they ensure, that the worker is at the station during the whole action, and that the buffer is before the station. They also ensure, that the correct variant is picked for the processed product.

The effects ensure, that no other product can start its processing at this station, that the actual processing step can start, and that the product gets correctly removed from the previous buffer.

```
(:durative-action automatic-work
:parameters
    (?s - station
    ?w - worker
    ?p - product
    ?v - variant)
:duration
    (= ?duration (automatic-work-time ?w ?s ?v))
:condition
    (and
    (at start (worker-released ?w))
    (at start (product-released ?p))
    (at start (product-of-variant ?p ?v))
    (over all (product-at-station ?p ?s))
    (at start (station-processing ?s))
    (at start (automatic-process ?s ?v))
    )
:effect
    (and
    (at end (not (station-processing ?s)))
    (at end (station-not-processing ?s))
    )
)
```

The automatic-work action is used to simulate the actual processing of the product at the station. The conditions ensure, that the product must have used the setup action at this process, that the product/worker are released, and that the variant is correct. At the end, the station is no longer processing, which is indicated by the two predicates.

```
(:durative-action automatic-takedown
:parameters
    (?s - station
    ?b - buffer
    ?w - worker
```

```
        ?p - product
        ?v - variant)
    :duration
        (= ?duration (takedown-time ?w ?s ?v))
    :condition
        (and
        (at start (product-of-variant ?p ?v))
        (at start (buffer-after-station ?b ?s))
        (at start (worker-released ?w))
        (at start (product-released ?p))
        (at end (buffer-unused ?b))
        (over all (worker-at-station ?w ?s))
        (at start (product-at-station ?p ?s))
        (at start (station-not-processing ?s))
        (at start (automatic-process ?s ?v))
        )
    :effect
        (and
        (at end (product-processed ?p ?s))
        (at end (not (buffer-unused ?b)))
        (at end (product-at-buffer ?p ?b))
        (at end (station-unused ?s))
        (at end (not (product-at-station ?p ?s)))
        )
    )
```

After the automatic-work action, the automatic-takedown action can be taken. Most of the conditions are self explanatory, the two important ones here are station-not-processing and buffer-unused. The first one only gets set to true after the automatic-work action, and therefore prevents a premature takedown. The second one ensures, that the buffer after the station is free at the end, so that there won't be two products at the same buffer.

This causes one problem. The buffer after the last station is supposed to be able to hold more than one product. Therefore an extra action is needed.

```
    (:durative-action automatic-takedown-endbuffer
    :parameters
        (?s - station
        ?b - buffer
        ?w - worker
        ?p - product
        ?v - variant)
    :duration
        (= ?duration (takedown-time ?w ?s ?v))
    :condition
        (and
        (at start (product-of-variant ?p ?v))
        (at start (buffer-after-station ?b ?s))
        (at start (worker-released ?w))
        (at start (product-released ?p))
        (at start (end-buffer ?b))
        (over all (worker-at-station ?w ?s))
        (at start (product-at-station ?p ?s))
        (at start (station-not-processing ?s))
        (at start (automatic-process ?s ?v))
        )
    :effect
        (and
        (at end (product-processed ?p ?s))
        (at end (product-at-buffer ?p ?b))
        (at end (station-unused ?s))
        (at end (not (product-at-station ?p ?s)))
        )
    )
```

The automatic-takedown-endbuffer action is the same as the non endbuffer variant, with the exception, that the buffer must not be empty, but must be an endbuffer. Therefore, this action can only be taken at the last station, and enables multiple products to be on the last buffer.

The effects of both the endbuffer and non endbuffer version are nearly the same. They remove the product from the station, changing the predicates accordingly, and place it on the next buffer. The difference is, that with the endbuffer variant, the buffer-unused predicate is not set to false. It is not necessary for the endbuffer, as a product can always be put on the endbuffer.

As the previous product must have been processed in order for the next one to be processed at a station, the automatic-takedown actions all set the predicate indicating that the product has been processed at a station to true.

```
    (:durative-action manual-work
    :parameters
        (?s - station
        ?before - buffer
        ?after - buffer
        ?w - worker
        ?p - product
        ?v - variant
        ?pbefore - product)
    :duration
        (= ?duration (manual-work-time ?w ?s ?v))
    :condition
        (and
        (at start (product-of-variant ?p ?v))
        (at start (product-before ?pbefore ?p))
        (at start (product-processed ?pbefore ?s))
        (at start (buffer-before-station ?before ?s))
        (at start (buffer-after-station ?after ?s))
        (at start (worker-released ?w))
        (at start (product-released ?p))
        (at start (station-unused ?s))
        (over all (worker-at-station ?w ?s))
        (at start (product-at-buffer ?p ?before))
        (at start (station-not-processing ?s))
        (at end (buffer-unused ?after))
        (at start (manual-process ?s ?v))
        )
    :effect
        (and
        (at start (station-processing ?s))
        (at start (not (station-not-processing ?s)))
        (at start (not (station-unused ?s)))
        (at start (not (product-at-buffer ?p ?before)))
        (at start (buffer-unused ?before))
        (at end (not (station-processing ?s)))
        (at end (station-not-processing ?s))
        (at end (station-unused ?s))
        (at end (product-processed ?p ?s))
        (at end (not (buffer-unused ?after)))
        (at end (product-at-buffer ?p ?after))
        )
    )
```

Similar to the automatic processing, there is a action for the manual processing. As the manual processing only consists of one step, only one action is needed. The conditions are a combination of the conditions from the automatic-setup action and the automatic-takedown action. They ensure, that a product with its correct variant gets picked, and that the automatic processing can take place at the time the action is taken.

The effects at the beginning of the action include removing the product from the previous buffer, and indicating, that a product is at the station and is getting processed. At the end of the action, the station gets freed up again, and the product gets placed on the next buffer. The products also gets marked as processed, so that the next product can start processing at the station.

```
    (:durative-action manual-work-endbuffer
    :parameters
        (?s - station
        ?before - buffer
        ?after - buffer
        ?w - worker
        ?p - product
        ?v - variant
        ?pbefore - product)
    :duration
        (= ?duration (manual-work-time ?w ?s ?v))
    :condition
        (and
        (at start (product-of-variant ?p ?v))
        (at start (product-before ?pbefore ?p))
        (at start (product-processed ?pbefore ?s))
        (at start (buffer-before-station ?before ?s))
        (at start (buffer-after-station ?after ?s))
        (at start (worker-released ?w))
        (at start (product-released ?p))
        (at start (station-unused ?s))
        (over all (worker-at-station ?w ?s))
        (at start (product-at-buffer ?p ?before))
        (at start (station-not-processing ?s))
        (at start (end-buffer ?after))
        (at start (manual-process ?s ?v))
        )
    :effect
        (and
        (at start (station-processing ?s))
        (at start (not (station-not-processing ?s)))
        (at start (not (station-unused ?s)))
        (at start (not (product-at-buffer ?p ?before)))
        (at start (buffer-unused ?before))
        (at end (not (station-processing ?s)))
        (at end (station-not-processing ?s))
        (at end (station-unused ?s))
        (at end (product-processed ?p ?s))
        (at end (product-at-buffer ?p ?after))
        )
    )
)
```

Similar to the automatic-takedown actions, there is a special action for the endbuffer. It has the exact same functionality asfor the automatic-takedown action. It can only be used at with the endbuffer, and allows for multiple products to be at the same buffer. Otherwise, it has the same conditions and effects as the normal manual-work action.

The second part of the PDDL model is the problem file. It changes with the instance of the Flow Shop problem, and is therefore, in contrast to the domain file, not always the same. The following is an example of a problem file with

```
    (define (problem flow-shop-01)

    (:domain flow-shop)

    (:objects
        worker1 - worker
        station1 - station
        buffer0 buffer1 - buffer
        helpproduct product1 - product
        variant1 - variant)

    (:init
        (product-of-variant product1 variant1)
        (end-buffer buffer1)
        (worker-released worker1)
        (product-released product1)
        (worker-at-station worker1 station1)
        (product-at-buffer product1 buffer0)
        (buffer-unused buffer0)
        (buffer-unused buffer1)
        (station-unused station1)
        (station-not-processing station1)
        (buffer-before-station buffer0 station1)
        (buffer-after-station buffer1 station1)
        (automatic-process station1 variant1)
        (product-before helpproduct product1)
        (product-processed helpproduct station1)

        (= (release-time-worker  worker1) 10)
        (= (release-time-product  product1) 10)
        (= (walk-time worker1 station1 station1) 0)
        (= (setup-time worker1 station1 variant1) 10)
        (= (automatic-work-time worker1 station1 variant1) 50)
        (= (takedown-time worker1 station1 variant1) 10)
        (= (manual-work-time worker1 station1 variant1) 20)
    )

    (:goal
        (and
        (product-at-buffer product1 buffer1)
        )
    )

    (:metric minimize (total-time))
    )
```

This is a problem file that is as small as possible, meaning there is only one station, two buffer, one worker, and one product that needs to be processed, together with its variant. The model however needs one additional product, here called helpproduct. It is needed, in every instance, as the first product also needs a previous product that needs to have been processed at every station. This is done by the helpproduct. Different instances can have different amounts of objects. For example, if an instance has three workers, then three worker objects would be in the problem file.

The other difference between instances are in the predicates that are initially true. A lot of the predicates just describe the instance and never change. In this case, this would be:

- **product-of-variant product1 variant1**: For each product, one variant must be specified to which it belongs. In this case, product1 belongs to variant1.
- **end-buffer buffer1**: In this case, buffer1 is the last buffer. Therefore, this predicate indicates, that it is the endbuffer.
- **buffer-before-station buffer0 station1 / buffer-after-station buffer1 station1**: Here, they specify, that buffer0 is before station1 and buffer1 is after station1.
- **automatic-process station1 variant1**: This indicates, that every product of variant1 gets processed at station 1 with an automatically.
- **product-before helpproduct product1**: As the helpproduct is needed to be defined as the product before product1, this predicate must be true at the start. If more products were part of the instance, then there would be more of these predicates, for example for the first and second product.
- **product-processed helpproduct station1**: As the helpproduct is not a real product that gets processed, but is only needed for the processing of product1, the product is already processed at station1. If more stations would be part of the instance, then this helpproduct would have been processed at all of them.

In addition to these predicates, there are multiple other ones, that are true initially to describe the instance, but can get set to false by actions later on, or are only true initially in this case. These are:

- **worker-released worker1 / product-released product1**: In this case, both the worker and product are initially released, meaning they can start processing or being processed directly. Once they are true, they never become false.
- **worker-at-station worker1 station1**: This indicates, that initially, worker1 is at station1. If there were more stations, the move action could set this to false, and set it to true for a different station.
- **product-at-buffer product1 buffer0**: The product must have an initial position. In this case, it starts at buffer0. It could also start at a different buffer, or at a station.
- **buffer-unused buffer0 / buffer-unused buffer1**: Initially, every buffer without a product is unused. As buffer0 is the first buffer, it can also be set to unused, as no product can ever be placed on it.
- **station-unused**: Similar to buffer-unused, as there is no product at the station, the station is initially unused.

These are all the predicates that are initially true. In addition to this, the functions must be set to contain the correct duration for each action. Calculations for worker skill are all done during the translation, so that the value of the function is the correct duration for each action.

The goal of the Flow Shop problem is to move every product through the stations and to the last buffer. Therefore, the goal in the problem file requires all products to be at the last buffer, in this case, product1 must be at buffer1. If there were more products or more buffer, then the goal would require every product to be at the last buffer.

The last part of the problem file is the metric, which indicates to the planner, what should be optimized. As the Flow Shop problem wants to minimize the makespan, the metric defined in the problem file indicates, that the total-time needed for the plan, which is the makespan of the solution, should be minimized. With this, the problem file and the PDDL model is complete.

4.2 Translation

The translation from a Flow Shop instance given in a CSP model to a PDDL version only needs to create the problem file. The domain file is always the same and does not need to be translated for each instance. For the problem file, the parts that differ for each instance are the objects, the initial state and the goal. How they are generated will be discussed in this section.

To get the values of an instance, the files for the Google OR-tools model will be used. From these files, every needed value can get used. The first part that differs for different instances are the objects. The number of workers, stations, products and variants can be taken, and the correct amount of objects can then be added. For example, if there are five workers in the instance, then five worker objects are added. Each get assigned a number, so in this case, there will be worker1 to worker5. The same is done for the other objects. The number of buffers can be taken from the number of stations, as there must be one more buffer than station. Here, the first buffer gets the number 0, and the following buffer gets the next number. When this is finished, the objects are complete.

After this, the initial state needs to be defined. The first half of this consists of the initially true predicates. For each type of predicate, there are different rules to add them:

- **product-of-variant**: The files of the Google OR-tools model define for each product, to which variant it belongs. Therefore, this information can be used to add this predicate for each product together with its corresponding variant.
- **end-buffer**: This predicate needs to be added once for the last buffer. The number of the last buffer gets defined by the amount of stations.
- **worker-released**: The files define, if a worker is released from the start. If this is the case, then this predicate is added for the worker, otherwise not.
- **product-released**: Similar to worker-released, if a product has no leftover time, then it can be used directly from the start, so that this predicate can be added to the initial state.
- **worker-at-station**: Each worker starts at a station. This is again defined by the files, and therefore, this predicate can be added for each worker together with its corresponding station.
- **product-at-station**: For a product to be at a station, there are two possibilities. The first one is, that the first action the product must make is not the first action at a station, meaning it was at least put on the station during the setup phase. The second possibility is, that a product still gets processed at a station. In this case, as the product gets processed, it must be on a station. If either of this is true, then this predicate gets set to true in the initial state for the product and its corresponding station.
- **product-at-buffer**: The same that was done for the stations needs to be done for the buffers. If a product does start with the first action at a station, and does not have any leftover time, then it is at a buffer at the start. Therefore, this predicate can then be added for every product which starts at a buffer, together with the correct buffer.

- **buffer-unused**: Each buffer, which is not used by any product gets added as unused. If a buffer is used can be found out in the same way as for product-at-buffer, by checking every product.
- **station-unused**: The same thing that was done for the buffers can also be done for the station, so that this predicate is added for all unused stations.
- **station-processing**: To check, if a station is processing, every product needs to be checked, if it is currently being processed on the station. If this is the case for some station, then this predicate must be added for this station.
- **station-not-processing**: If station-processing is not added for a station, then this predicate gets added instead, as it is the negation.
- **buffer-before-station**: As the order of stations and buffers is given simply by the amount of station, this predicate can be set for every buffer with the corresponding station. For example, buffer0 is before station1, and buffer1 is before station2.
- **buffer-after-station**: This can be done in the same way as buffer-before-station. Therefore, this predicate gets set to true for each buffer with the correct previous station, for example buffer1 and station1.
- **automatic-process**: The CSP model defines, how many automatic worksteps each variant has at each station. If this number is bigger than zero, then it must be an automatic process. Therefore, for each variant and station, if this is the case, this predicate gets added.
- **manual-process**: Similarly, if the number of automatic worksteps is zero, then this variant gets processed manually. Therefore, this predicate can then be added to the initial state.
- **product-before**: Which product is before which other one is given by only the number of products. Therefore, this can be added without any further information of the instance. The only important part here is, that the helpproduct must also be before product1, so that the the predicate (product-before helpproduct product1) must be added.
- **product-processed**: This gets added in two parts. Firstly, the helpproduct gets added as processed at every station. After that, for every product, the starting location gets checked. For every station before the starting position, this predicate gets added with the specific station and product. If a product is at a station in the beginning, this station does not get added as having finished processing the product. This is because the product must still finish processing at the station.

When this is done, all predicates that are initially true are part of the initial state. After that, the functions must be added. For each type of function, Each combination of objects the object can have must be added, together with the correct value. These values can be taken from the instance files. Any calculations are done in advance, so that only the correct value gets added to the initial state. For example, the takedown time for worker1, station1 and variant1 can be calculated by the values given in the instance files. The formula is given as $t \cdot \frac{100}{s}$, where t is the base takedown time for a product of variant1 and s is the worker skill of worker1.

After all function values are added, the initial state is complete. After this, the goal can be constructed. To do this, the information about how many products are part of the instance, and what the last buffer is, is needed. With this, the goal can be constructed by adding together the predicates, that every product must be at the last buffer. If the number of products is known, then all products can be created by enumerating from one to this number. With this done, the rest that does not change can get added, so after the goal the metric. After that, the instance got translated into PDDL, and can be solved by a PDDL solver.

4.3 Theoretical characteristics of the PDDL model

In order for this PDDL model to have any value when comparing it to the CSP models, we must show, how they are similar. To do this, multiple things will be shown:

Firstly, we will show correctness. This means, that, given an CSP instance of a Flow Shop problem, together with a solution for this CSP instance, we can construct a solution in the PDDL model for the same instance.

Next up, we will show completeness. This will be shown, by showing the other way around, meaning if we have a solution for the instance in the PDDL model, we can construct a solution in the CSP model for the same instance.

4.3.1 PDDL model difference

The PDDL model of the Flow Shop does have one aspect, in which it is different from the CSP models. To understand this difference, one can look at the Google Or-Tools model. The constraint for the automatic stations states, that the start time of a takedown action at a station must be equal to the endtime of the processing of the same product at the same station. This means, that the takedown must follow directly after the automatic processing. This is different to the PDDL model. Here, after the automatic processing, time can pass, until the takedown needs to be performed.

This difference is important. In the PDDL model, a worker can start the automatic processing, and then go work with other products, until he return to perform the takedown action. In the CSP model, this might not always be possible, as the worker could return too late to perform the takedown action in time. The two models therefore allow different solutions to instances, that should be the same.

The problem here is, that the used version of PDDL does not have a feature, that enforces one action to take place directly after a different one. There are also no other easy fixes for this problem. Trying to combine these two actions into one, which could be seen as the two actions happening directly after another, does not work, as these actions have different requirements. The automatic work action, for example, does not need a worker, while the takedown action does. Therefore, this approach would not work correctly. Using predicates to enforce this does also not seem like a good idea, as it is questionable, what such a predicate should even try to do. This is ultimately a problem of PDDL itself. The models are however still close enough, that a comparison does make sense.

4.3.2 Correctness

We will now show correctness. As the difference in the models does only allow the PDDL model to do more actions, this is not a problem for correctness. To show correctness, the Google OR-tools model will be used, as it was already described earlier. We assume, we have the CSP and PDDL models, an instance of a Flow Shop problem, and a solution for the CSP model. We now need to construct a plan for the PDDL model, and need to show, that this plan is valid.

To construct the plan, we can take a look at theoutput of the Google OR-tools model. We have three parts:

- Worker events, that state, during which times a worker is at which station
- Product events, that state, during which times, a product is at which station or buffer.
- Worksteps, that state, when each workstep is performed at which station.

Such a solution looks like this:

```
<FlowShopSolution configuration="Example1" initialState="Example1">
<Events>
<Workers>
<Event worker="WORKER-1" station="STATION-1" start="0" end="100" />
<Event worker="WORKER-1" station="STATION-2" start="110" end="210" />
<Event worker="WORKER-1" station="STATION-3" start="220" end="320" />
<Event worker="WORKER-1" station="STATION-4" start="330" end="430" />
<Event worker="WORKER-1" station="STATION-5" start="440" end="540" />
</Workers>
<Products>
<Event product="PRODUCT-1" location="STATION-1" start="0" end="100" />
<Event product="PRODUCT-1" location="BUFFER-1" start="100" end="110" />
<Event product="PRODUCT-1" location="STATION-2" start="110" end="210" />
<Event product="PRODUCT-1" location="BUFFER-2" start="210" end="220" />
<Event product="PRODUCT-1" location="STATION-3" start="220" end="320" />
<Event product="PRODUCT-1" location="BUFFER-3" start="320" end="330" />
<Event product="PRODUCT-1" location="STATION-4" start="330" end="430" />
<Event product="PRODUCT-1" location="BUFFER-4" start="430" end="440" />
<Event product="PRODUCT-1" location="STATION-5" start="440" end="540" />
</Products>
</Events>
<Schedule objective="540.0" solver="OR-Tools" status="OPTIMAL"
solvingTime="0.01074711699038744" gap="0">
<Worker id="WORKER-1">
<AssignedWorkSteps>
<Workstep workstep="WORKSTEP-V1-S1" station="STATION-1"
product="PRODUCT-1" start="0" end="100" />
<Workstep workstep="WORKSTEP-V1-S2" station="STATION-2"
product="PRODUCT-1" start="110" end="210" />
<Workstep workstep="AUTOWORKSTEP-V1-S3" station="STATION-3"
product="PRODUCT-1" start="220" end="230" />
<Workstep workstep="AUTOWORKSTEP-V1-S3" station="STATION-3"
product="PRODUCT-1" start="310" end="320" />
<Workstep workstep="WORKSTEP-V1-S4" station="STATION-4"
product="PRODUCT-1" start="330" end="430" />
<Workstep workstep="WORKSTEP-V1-S5" station="STATION-5"
product="PRODUCT-1" start="440" end="540" />
</AssignedWorkSteps>
</Worker>
</Schedule>
```

```
</FlowShopSolution>
```

We can use this, to construct our PDDL solution. To do this, we will go over all possible actions, state, when they are added to the plan, and show, that they can be used in these cases.

The first actions are the release actions for the workers and products. We can just put them at the start of the plan for every worker or product, that is not directly released at the start. Because the released predicate is only added to the initial state for every worker or product that is released in the translation, this is always possible. The CSP model furthermore requires this release time to be over before a worker or product can do anything in the CSP model. Because of this, we can assume, that for every later action, if a worker or product is part of that action, that this release action can finish before that later action. We can therefore assume, that the worker-released and product-released predicate is true for every later action when needed.

After this, we need to add the move actions. For this, we can simply look at the worker events. They state, when a worker needs to be at which station. We can simply put a move action between each following pair of these events. In the example, worker1 is at station1 from 0 to 100, and is at station2 from 110 to 210. We therefore need to add a move action from station1 to station2 that starts at 100. The duration of this action will at most be 10, as the CSP constraints require the movement time to be between the worker being at two different stations. The action will also always be possible when needed this way, as we know from before, that the worker will be released. The worker-at-station predicate will also be true for the correct station, as the translation sets the predicate with correct station to true initially, and each following move action sets the correct predicate to true at the end. As the CSP constraints enforce, that a worker must be at a station, if he performs something at that station, we can also assume, that the worker-at-station predicate is true for all following actions.

The next action is the automatic setup action. This action will have to be added to the plan for every first autoworkstep of a product at a station. In this case, one would have to be added starting at 220 at station3 with product1. The duration of this action will be the same as the duration of the workstep in the CSP solution. This action can also always be taken at this time. We already said, that the worker and product must be releaed at this time, and that the worker must be at this station, because of the CSP constraints. A lot of the other predicates must also be set to true trivially. They are product-of-variant, buffer-before-station, automatic-process and product-before. All of these must be true, as we assume that the correct variant, buffer and before product get picked for this action, and because we know, that this variant must be processed automatic at this station, as it also gets automatically processed in the CSP model. We can assume these to be true, as the translation must set them to true in the initial state. As the CSP constraints enforce, that every previous product must have already been processed at the station in order for the next product to be processed there, we can also assume product-processed to be true for the previous product. In this case, as this previous product is the helpproduct, this is trivially true due to the initial state. For all other predicates, we can see, they must be true, as the CSP constraints also enforce them to be true in this moment. As station-unused is only set to false when a product is put on the station and set to true, when it leaves the station, and the CSP constraints enforce, that no product can be at the station during this time, we can also assume this predicate to be true. Similarly, as the CSP constraints enforce, that the previous station must have finished processing the product, we can also assume the product to be at the buffer before the station, so that product-at-buffer must be true. the last predicate is station-not-processing. As this only gets set to false during the automatic work action or during a manual work action, and we already know, that no other product is at the station, we know, that this must also be true. Therefore, all predicates of this action are always true when needed, so that we can use it in these times.

The automatic work action needs to be placed between every two autoworkstep events at each station. This is possible, as the duration for this action is defined by a function, that has a value that gets calculated in the same way, the duration of the automatic workstep gets calculated in the CSP model. We therefore only need to show, that the predicates in the condition are true at the correct times. Most of them are again trivially true, just like in the setup action. The two, which are not trivially true are product-at-station and station-processing. these however both need to be true at this point in time. An automatic workstep can only happen after the setup, which sets both of these predicates to ture. We can therefore also assume, that both of these predicates must be true at this point in time.

The last actions regarding the automatic processing are the two possible takedown actions. We need to add them to the plan for every second autoworkstep action at a station. If the station is the last station, then we use the endbuffer variant, otherwise the normal version. The duration is again correct with the same argument as before, so that we again only need to show, that all predicates in the condition are true. The only two non trivial predicates for the normal version are buffer-unused and station-not-processing. As the CSP constraints directly require, that the previous product must have already started processing at the next station with the buffer constraint, we can assume, that the previous product must have already been used with a setup or manual work action, both of which make the buffer empty, so that this predicate must be true. station-not-processing must also be true, as the takedown action must take place after an automatic work action, which sets this predicate to true. As all predicates are true, we can always use the normal takedown action when needed. The endbuffer version is also always possible when needed. As we only use it at the last station, we can assume the only different predicate being the end-buffer to be true, as the initial state will set this predicate to true for the buffer according to the translation. We can therefore also always use the endbuffer variant when needed.

The last set of actions are the manual work actions. We need to add them to the plan for every workstep event in the solution, so in this case four. The first one would start at 0 with a duration of 100. We again assume this duration to be correct because of the way, the translation creates the functions. All of the predicates in their conditions are again true with the same arguments as for the previous actions. Most of them are trivially true. station-unused, product-at-buffer and station-not-processing must all be true because of the same reasons as stated in the setup action, and buffer-unused must be true because of the same reasons as stated in the takedown action. When looking at the endbuffer variant of the action, we again need to use it for the last station. The endbuffer predicate is also again true due to the same reasons stated for the takedown endbuffer variant.

With this, the plan is complete, and we know, that such a plan is possible with the conditions of the domain. WE now only need to show, that this plan also fulfills the goal. This is however easy. As the CSP solution must have processed every product at the last station, we know, that either the takedown action of manual work action was performed at the last station for every product. Therefore, every product must be at the last buffer, so that the goal is fulfilled. With this, we have shown correctness.

4.3.3 Completeness

The same now has to be done with the other way around. We again assume, we have both models, and this time, we have a plan for the PDDL model. We now need to construct a solution for the CSP model of this instance. This is not directly possible because of the difference in the models.

If we however assume, that the takedown action is performed directly when the automatic work action is finished, we can create a solution for the CSP model.

Creating the solution itself is relatively easy in this case. Each of the variables for each workstep directly corresponds to an action in the plan. We therefore just need to find this corresponding action and set the first variable to the starttime of the action and the second one to the starttime plus the duration of the action. for each workstep for a product at a specific station, there must be an action in the plan, for the same product at the same station, as the PDDLplan must also process every product at every station. To set the boolean variables that define, which worker does which workstep, we just need to look, which worker is responsible for which workstep. The varaiable fo this worker and workstep can then be set to ture, while the variables for every other worker gets set to false.

This generates a full assignment for our CSP, we now only need to show, that this assignment is consistent. To do this, we need to check every constraint and need to to show, that it must be fulfilled according to the PDDL model. The first constraint requires, that the endtime for each task must be equal to the start time plus the duration of the task. The way in which the durations of the actions gets translated does mean, that every action takes exactly as long as the duration in the CSP model. Therefore, this constraint is fulfilled for every task.

The next constraint enforces, that every task can only be done by one worker. As each action can only have one worker, this constraitn is also trivially fulfilled.

The next constraint is more complex. It requires, that a product can only be performing a task, after its leftover time. This is enforced in the PDDL model with the release-product action. If a product has leftover time at the start of the instance, the release-product action must first be used. As this action has the same duration as the leftover time, no other action with this product can happen in the plan before this action. Therfore, all starttimes are also after this leftover time, so that this constraint is fulfilled.

The next constraint enforces, that for a product to be processed at a station, it must have been processed at the previous station, if any tasks at this stations existed. This also needs to have happened in the PDDL plan, as the only way for the product to get to a station is to be put on the buffer before it, which can only be done by being processed on the previous station. Therefore, this constraint is fulfilled.

A similar constraint enforces, that a product must have been processed at a station, before the next product can be processed there. This is directly enforced by PDDL thorugh the product-before and product-processed predicates, so that this condition is fulfilled.

The next constraint enforces a buffer size of one. It requires a product to have started being processed at a station, before before the next one can finish processing at the previous station. Again, this is enforced by the PDDL model, as each buffer can only hold one product with the exception of the first and last buffer, which are not important for this constraint. As they can only hold one product, this product must start being processed at the next station, before the next product can finish the previous station. Therefore, this constraint is also fulfilled.

The automatic station condition is the one, which would not be fulfilled directly by the PDDL model. It requires the start of the takedown to be at the time of the end of the setup plus the duration of the automatic work. If we however also enforce, that this takedown must directly happen in the PDDL plan, then this condition is also fulfilled.

The last constraint enforces, that a worker must have time between tasks to move between stations if needed. As the PDDL model does require a direct movement action, which has the correct duration because of the translation, this constraint can also be seen as fulfilled.

As we could construct a consistent full assignment for the CSP model of the Flow Shop instance out of the PDDL solution with the extra condition mentioned before, we have also shown completeness. As we have also shown correctness, we know, that the models do describe the same problem, and that a comparison between them makes sense.

4.4 Advantages of the PDDL models

As it is mostly possible to translate the CSP model to PDDL, it is of question, why such a translation would make sense. Performance is one possibility, meaning the PDDL model could find solutions faster, the solutions could have a smaller makespan, or it could simply solve larger instances. These are all possibilities, two other are discussed in this section.

4.4.1 Readability

The PDDL model can have the advantage of readability. This both applies to the model files, so the domain and the problem file, but also to the plans. The CSP models create variables that stand for the start and end time of different tasks and if a worker does perform a task, and then add constraints to them. These constraints can be hard to understand. Taking the Google OR-Tools model as an example, the constrains are often inside multiple loops and conditions. What exactly these conditions are is not directly clear. PDDL has the advantage here, that every action, predicate and function can be names according to its function. It therefore becomes clear, what each action does in the problem, and what a predicate is used for.

This also applies to the solutions. The Google OR-Tools solution for example does not show, when a worker needs to change stations, but only when a worker is at which station. The PDDL plan in contrast would show the actions of the worker, meaning the move actions the worker uses to move between the stations. This could be more readable and easier to understand.

4.4.2 Adaptability

Another advantage could be the adaptability. If the problem is not fixed and changes later on, PDDL has multiple options to adapt the model to the new problem, without having to design a completely new model. If, for example, another possible processing can happen at a station, then this can simply be added to the PDDL model with additional actions and, if needed, additional predicates and functions. In the CSP model of Google OR-Tools, this would be harder to implement. New constraints would have to be added, also new variables, and the old constraints would have to be checked, if they conflict with the new possible processing.

It is possible, that some change is not possible to add to the PDDL model, similar to the translation problem, but apart from that, the model is adaptable to change.

5 Experiments

To compare the PDDL version of Flow Sop to the CSP versions experiments are needed. This chapter introduces thedifferent planners, the benchmark set, and discusses the experiments.

5.1 Planner

Multiple planners could have been used to solve the PDDL instances. A problem for this was, that a lot of planners were not available. A good start to find planners that support durative actions would have been the International Planning Competition 2018[21]. The problem here was, that the website for the temporal tracks was down, so that no information about planners that support durative actions could be found here. However, a set of temporal planners could be found. Some of them did however not work for the experiments, and others had to be used with planutils[22], a tool that allows to use multiple planners locally and handles the necessary installments. With this, multiple temporal planners could be used.

5.1.1 TemPorAl

TemPorAl[23] is a portfolio planner, that competet in the IPC2018. It uses multiple other temporal planners and assigns time to them equally. It however did not work locally, and could therefore not be used for the experiments.

5.1.2 CP4TP

CP4TP[24] is also a portfolio planner. The idea behind the planner is, to first try and solve the problem sequentially, and if this fails, try more complex solutions, which don't have to be sequential anymore. This planner could be run locally, and was used to solve the instances.

5.1.3 POPF

POPF[25] is another temporal planner. It uses forward search, together with linear programming. It was possible to make this planner run with planutils, and can therefore be used for the experiments.

5.1.4 Temporal Fast Downward

Temporal Fast Downward, or TFD[26], is another temoral planner. It performs "heuristic search in a temporal search space" [26, p. 1]. As it was also provided in planutils, it can be used for the experiments.

5.1.5 Differences of the planners

The important difference between the planners is the type of plan they generate. CP4TP generates mostly sequential plans, meaning not many actions can happen at the same time. The other used planners, so POPF and TFD allow for more parallel actions. The effect of this will be seen in the experiments, but it is important to know this.

5.2 Benchmark set

To run the experiments, a benchmark set was needed. To understand, how different types of instances get solved by either the CSP solvers and the planners, different types of subsets were created. For most of them, all characteristics of the instances were tried to be kept the same, with the exception of one, which was increased over the instances. There were overall three types of these sets:

- Benchmarks, in which the number of stations and workers stayed the same, and the amount of products increased.
- Benchmarks, in which the number of products and workers stayed the same, but the amount of stations increased.
- benchmarks, in which the amount of stations and products stayed the same, but the amount of workers increased.

For all of them, a small and a larger set were used. This means, for example, for increasing number of products, one set only had a few stations and workers, while the other set had more stations and workers. Lastly, an extra set was added, that contained large problems, to see, how good the solvers and planners perform, if the instances get big.

5.3 Running the solvers and planners

To run CPlex, the free CPlex solver by IBM was used. The standard parameters were used, meaning a maximal search time of 300 seconds. It also had a maximal input size, so that too big instancescould not be solved, as they contained too many variables and constraints.

Google OR-tools could be run locally in python. It was run with a maximal time of 1000 seconds. This time was used to generate the model and to find a solution, but in most cases, this time was nearly fully used to find the solution.

CP4TP was also run locally. It was run with a memory limit of 6000MB, and a time limit of 1000 seconds.

POPF and TFD were run in planutils. They were run with a time limit of 1000 seconds.

5.4 Results

The results focus on two main features: The makespan of the solution, and the time needed to find the solution. The searchtime is always given in seconds. Regarding the searchtime of CPlex and Google OR-Tools, multiple things are important to note in advance: Both often reached the maximum search time when trying to find a solution. Increasing this time however, would not have made a lot of differences. In most cases where the whole time was used, CPlex and Google Or-Tools could find a solution with the final makespan in a few seconds, and spend the rest of the time trying to improve the solution. The problem was therefore not finding an optimal solution, but proving, that this solution is optimal. More search time was therefore not necessary.

When CPlex could not find a solution for a problem, the issue was never that it ran out of time, but that the input was too big. The used version of CPlex did only allow problems up to a certain size, limited by the number of variables and constraints in the instance. CPlex not finding a solution was therefore never a problem of time, but a problem of input size.

In contrast to this, the Google OR-Tools could handle every problem in regards of size. If no solution could be found with it, then no solution at all was found in the 1000 seconds. In these cases, the instance was not too big to even create the model.

The planners had different reasons why they could not find a solution. Either, the problem got too big and exceeded the memory limit, or they could not find a solution after their time limit of 1000 seconds. For most planners, the progress could be observed during the search. The way this was presented differed between planners, but if no solution was found in the 1000 seconds, then in most cases, no progress could be seen at all, indicating, that the planner didn't get close to finding a solution.

5.4.1 Changing products with small problems

The first benchmark set consisted of small problems. The difference between the instances is an increase in the number of products. Each instance had one worker, five stations, six buffers and one variant. The amount of products started with one, and went up to 15. The makespan can be seen in table 5.1. Comparing the CSP solvers, we can see, that both always have the same makespan. This happens, because both always find an optimal solution. Looking at the PDDL makespans, we can see, that the solutions are not a whole number, but decimal numbers. To understand this, a look at an actual plan is usefull. This is part of a plan, generated by CP4TP on the instance of this benchmark set with five products:

```
0.0000: (MANUAL-WORK STATION1 BUFFER0 BUFFER1 WORKER1 PRODUCT1
VARIANT1 HELPPRODUCT) [100.0000]
100.0002: (MOVE STATION1 STATION2 WORKER1) [10.0000]
110.0004: (MANUAL-WORK STATION2 BUFFER1 BUFFER2 WORKER1 PRODUCT1
VARIANT1 HELPPRODUCT) [100.0000]
210.0006: (MOVE STATION2 STATION1 WORKER1) [10.0000]
220.0008: (MANUAL-WORK STATION1 BUFFER0 BUFFER1 WORKER1 PRODUCT2
VARIANT1 PRODUCT1) [100.0000]
320.0010: (MOVE STATION1 STATION3 WORKER1) [20.0000]
```

The number before each action is the time it begins, and the number at the end is its duration. The first action is a manual work action at station1, after which a move action is used. As the manual-work has a duration of 100, the move action could start at exactly 100. The planner however adds a delay of 0.0002. This delay is not necessary, and could simply be removed by letting the action start at directly 100, and not 100.0002. As the effect of this is not big in relation to the overall makespan, it can however be ignored.

When ignoring the numbers after the decimal point, CP4TP does also find a solution that has a makespan equal to the CSP solvers. It is however not optimal, because of the translation error. When looking at the TFD and POPF makespan, this becomes obvious. POPF starts with a bigger makespan than every other solver and planner, while TFD has the same makespan as the CSP solvers, when ignoring the digitsa after the decimal point. As the amount of products increases however, both get makespans that are smaller than the makespans of CPlex and Google OR-Tools.

This happens, because the plans let the worker do other work, while a product gets automatically processed. The CSP models can not do this, as then the worker would return too late, and could not perform the takedown part of the processing in time. With more products, this effect can be used more. Therefore, This effect is biggest with 15 products. The makespan of POPF is 1080 smaller than the makespan of CPlex and Google OR-Tools. This is however not the planners performing better than the CSP solvers, but the result of the translation error.

CP4TP does not use this however. The reason for this is, that CP4TP tries to find sequential plans first. In such sequential plans, this change can not be used, as the worker can not move during the automatic processing. It therefore only finds plans with the same makespan as the CSP solvers.

The planners perform very well on these problems. Even though an exact comparison is not possible because of the translation error, they find very good solutions. If the solutions of POPF are optimal is clear, but no solution, with the exception of the POPF solution to the first problem, has a worse makespan than a CSP solver solution. A reson fir this could be, that the problem overall allows for very little parallelism. With only one worker, the only way to have things in parallel is to have an automatic process at a station, during which a worker does something else. Because of this, the fan out of the problem is small, as the only possible actions are regarding one worker, and during his actions, nothing is possible.

This performance can also be seen, when looking at the searchtime in table 5.2. CP4TP never needs more than a fraction of a second to find a solution. TFD and POPF also find solutions really fast, with TFD needing at most 11.8345 seconds for the last problem and POPF needing at most 9.41 seconds. the solvers need much more time, with CPlex using the full time as soon, as there are six products in the instance. Google OR-Tools never needs the full time, but needs significantly more time than the PDDL planners. The most time is needed for the instance with twelve problems, where Google OR-Tools needed about 253 seconds. It is however important to note, that for both of these planners, this seachtime was not fully needed to find the solution. In all cases, a solution with the same optimal makespan was found nearly instantly, and the rest of the time was spend trying to improve this solution. The error here was, that the CSP solvers could not find out, that their solution is optimal.

The graph 5.1 shows how the makespan changes with an increasing number of products. All Planners and solvers behave nearly identical. POPF has a smaller slope, as it uses the possible parallelism the most. TFD only starts using it at then products. Apart from that, the makespan of all solutions is nearly identical.

The searchtime is more interesting. The graph 5.2 shows, how the searchtime changes with an increase of products. CPlex quickly rises, using the full searchtime at six products. Google OR-Tools also quickly rises, using much more time than the PDDL planners. The searchtime of the planners can be better senn in graph 5.3. POPF needs the most time, until the last instance, where TFD needs the most time. CP4TP needs nearly no time at all for any of problems, most likely, because it does find sequential plans. The increase in searchtime for the other planners however indicates, that with bigger problems, they could have problems finding a solution.

5.4.2 Changing products with larger problems

The second benchmark set also increased the number of products, but used more stations and workers. In each instance, three workers and ten stations were used, while the number of products did

Number of products	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	540	540	540.002	540.102	660.005
2	1120	1120	1120.0048	1120.22	1080.003
3	1700	1700	1700.0072	1700	1580.003
4	2280	2280	2280.0096	2280.46	2080.003
5	2860	2860	2880.012	2860.58	2580.003
6	3440	3440	3440.0144	3440.7	3080.003
7	4020	4020	3920.0168	4040.83	3580.003
8	4600	4600	4600.0192	4600.94	4080.003
9	5180	5180	5180.0216	5181.06	4580.003
10	5760	5760	5760.024	5381.27	5080.003
11	6340	6340	6340.0264	5921.4	5580.003
12	6920	6920	6920.0288	6501.48	6080.003
13	7500	7500	7500.0312	7141.63	6580.003
14	8080	8080	8080.0336	7801.75	7080.003
15	8660	8660	8660.036	8401.87	7580.003

Table 5.1: Makespan of small problems with increasing number of products

Number of products	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	0.09	0.01	0	0.00863	0.13
2	1.68	0.018	0.004	0.0220441	0.24
3	2.68	0.0012	0.004	0.03	0.38
4	6.52	0.2705	0.00799	0.033	0.6
5	106.75	10.3643	0.01199	0.0539	0.9
6	300.5	39.0863	0.012	0.54913	1.19
7	300.02	14.6224	0.01198	0.0963931	1.64
8	300.02	32.4881	0.01999	0.136355	2.25
9	300.08	41.2264	0.0279	0.195998	2.92
10	300.08	65.1023	0.03199	0.653239	3.58
11	300.06	77.8673	0.04399	0.984936	4.55
12	300.07	253.3134	0.052	1.64686	5.6
13	300.09	90.6608	0.06798	3.87247	6.97
14	300.04	90.3739	0.0719	7.87123	8.03
15	300.06	120.7731	0.08399	11.8345	9.41

Table 5.2: Searchtime in seconds of small problems with increasing number of products

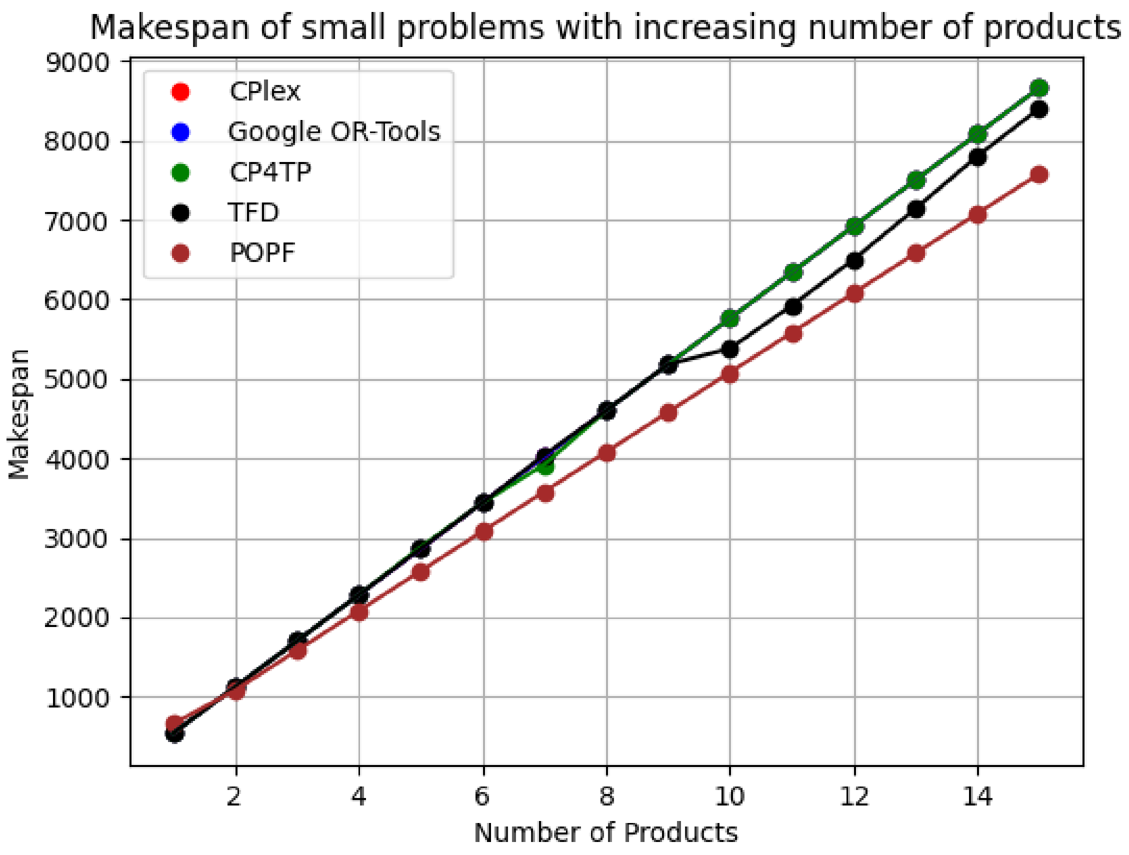

Makespan of small problems with increasing number of products
CPlex
Google OR-Tools
CP4TP
TFD
POPF
Makespan
9000
8000
7000
6000
5000
4000
3000
2000
1000
2
4
6
8
10
12
14
Number of Products

Figure 5.1:

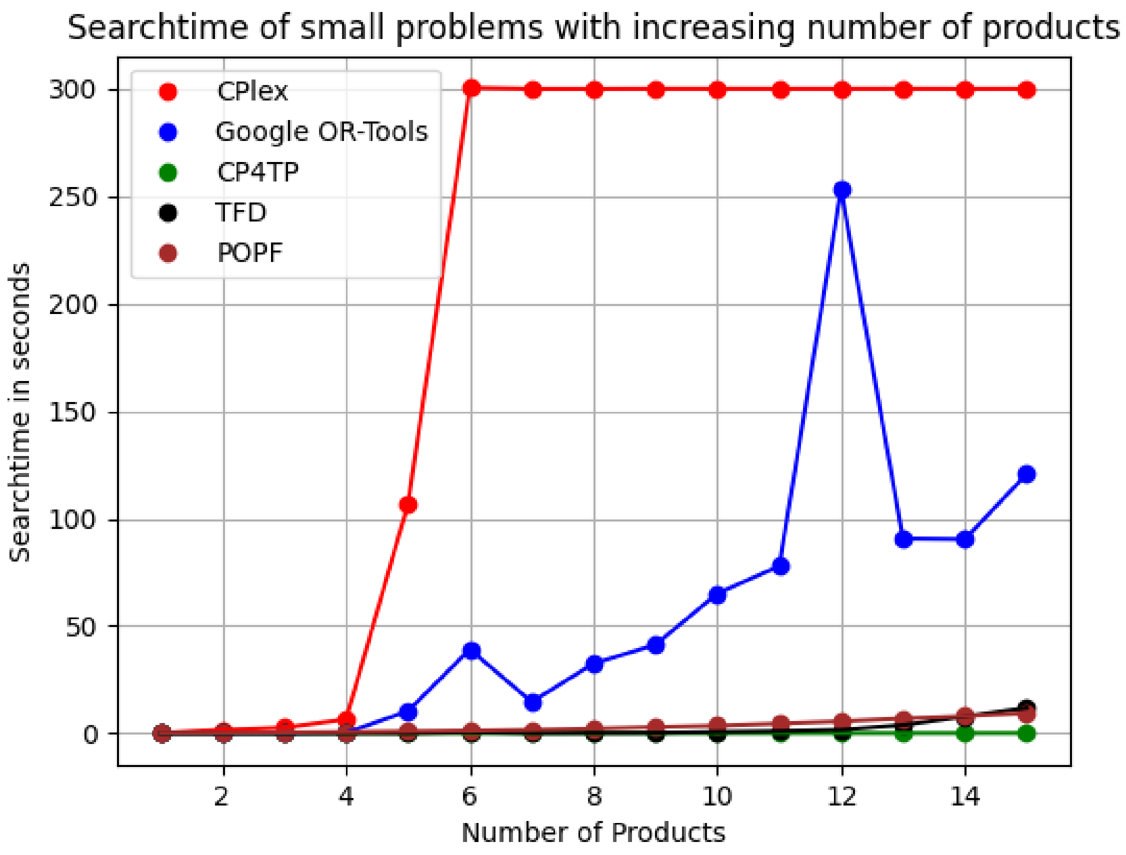

Searchtime of small problems with increasing number of products
CPlex
Google OR-Tools
CP4TP
TFD
POPF
Searchtime in seconds
Number of Products
0
50
100
150
200
250
300
2
4
6
8
10
12
14

Figure 5.2:

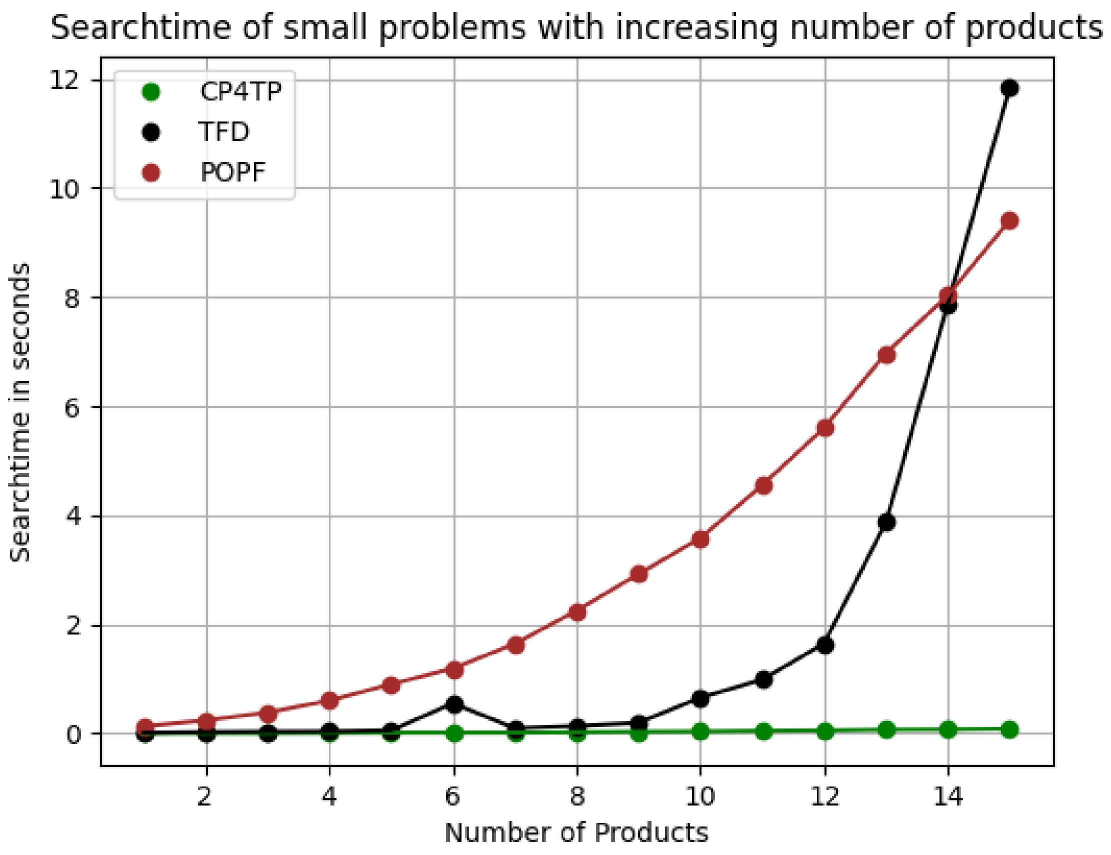

Searchtime of small problems with increasing number of products
CP4TP
TFD
POPF
Searchtime in seconds
Number of Products
0
2
4
6
8
10
12
14

Figure 5.3:

again increase. The smallest problem had one product, while the largest one had ten products.

The makespan can be seen in table 5.3. This time, multiple instances could not be solved. CPLex could only solve instances with up to four products, after which the problems got too big. TFD could not solve any of the instances, and POPF could only solve the first four. Google OR-Tools and CP4TP could however solve all of them. The reason why TFD and POPF had problems solving the bigger could be, because they allow more parallelism. As these instances have three workers, more actions could happen at the same time. For example, all of them could be used in an manual work step at a station at the same time. TFD could not solve these instances at all, indicating, that it can not handle much parallelism.

Looking at the makespan in the graph 5.4, one can see, that POPF does perform well in the beginning, but gets worse than Google OR-Tools at four products, before it can not solve any problems at all. CP4TP had a similar makespan as Google OR-Tools with only one product, likely, becuase not much parallelism is possible with only one product. As soon as the number of products increased, the makespan got worse when comparing it to Google OR-Tools. At ten products, it needed nearly three times the makespan Google OR-Tools needed. CPlex however could always solve the instance optimally together with Google OR-Tools. The limiting factor for it was therefore most likely the input size, as it could likely solver bigger instances, if a bigger inpout was allowed.

When looking at the searchtime in table 5.4, we can see, that CPlex and Google OR-Tools quickly used up all their given time. CPlex needs all the time as soon as the instance contains three products, and Google OR-Tools with four products. CP4TP does need more time for the bigger problems with more products, but even the largest one gets solved fast. Only 0.311747 seconds are needed. POPF needs more time than CP4TP, and depending in the instance even more than the CSP solvers. with two products, POPF needs nearly ten seconds, while both CSP solvers still need less than one second to solve the instance. The biggest solved instance by POPF was however solved in about 22 seconds, while CPlex and Google OR-Tools both used all their time, which was significantly more. This can also be seen in graph 5.5. With only one or two products, all solvers and planners need nearly no time, but starting with three products for CPlex, and four for Google OR-Tools, the CSP solvers needed much more time than the PDDL planners. The difference between between the PDDL planners can be better seen in graph 5.6. The searchtime of CP4TP stays low, while the searchtime for POPF does rise significantly. This rise in searchtime, together with the not solved bigger instances of POPF does show, that POPF does begin to struggle, once more products get added to the instances. The reason for this is most likely the possibility of parallelism in these problems. As CP4TP does not try to use any actions in parallel, its searchtime stays small.

5.4.3 Changing stations with small problems

The next benchmark set increases the amount of stations, while keeping the number of workers and products the same. In these instances, the number of worekrs was always one and the number of products was always five. The amount of stations increased from one, up to ten.

Table 5.5 shows the makespan for the different instances. CPlex and Google OR-Tools could both not solve all the instances, both only being able to solve instances with up to eight stations. After that, the input was too big for CPlex, and Google OR-Tools could not find a solution in the provided time. The planners could all find solutions for all instances.

The makespan for both CPlex and Google Or-Tools is the same for all instances, as both could always find the optimal solution. When comparing their makespans with CP4TP, we can see, that

Number of products	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	1000	1000	1260.0028		100.011
2	1100	1100	2680.0068		1250.008
3	1410	1410	4170.0094		1510.007
4	1770	1770	3100.0098		2530.008
5		2200	5200.0124		
6		2730	6420.016		
7		3180	7270.0188		
8		3600	8320.0206		
9		4070	9410.0228		
10		4650	13340.0326		

Table 5.3: Makespan of larger problems with increasing number of products

Number of products	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	0.04	0.1584	0.006699		0.22
2	0.06	0.5315	0.0142255		9.46
3	300.06	19.7255	0.0262697		16.61
4	300.04	1001.2908	0.0452973		21.81
5		1001.0992	0.06824		
6		1001.5309	0.0662		
7		1001.2679	0.092		
8		1001.2219	0.1584		
9		1000.7189	0.203471		
10		1002.1019	0.311747		

Table 5.4: Searchtime in seconds of larger problems with increasing number of products

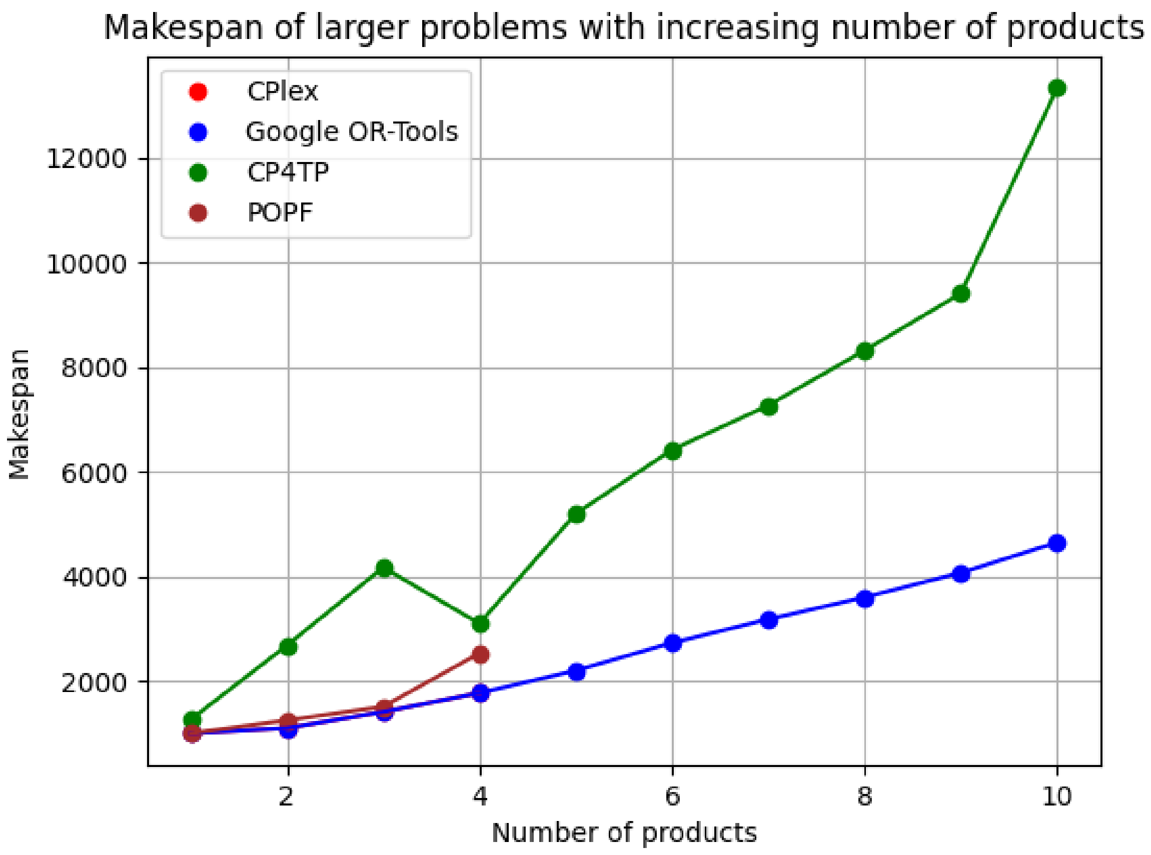

Makespan of larger problems with increasing number of products
CPlex
Google OR-Tools
CP4TP
POPF
12000
10000
8000
6000
4000
2000
Makespan
2
4
6
8
10
Number of products

Figure 5.4:

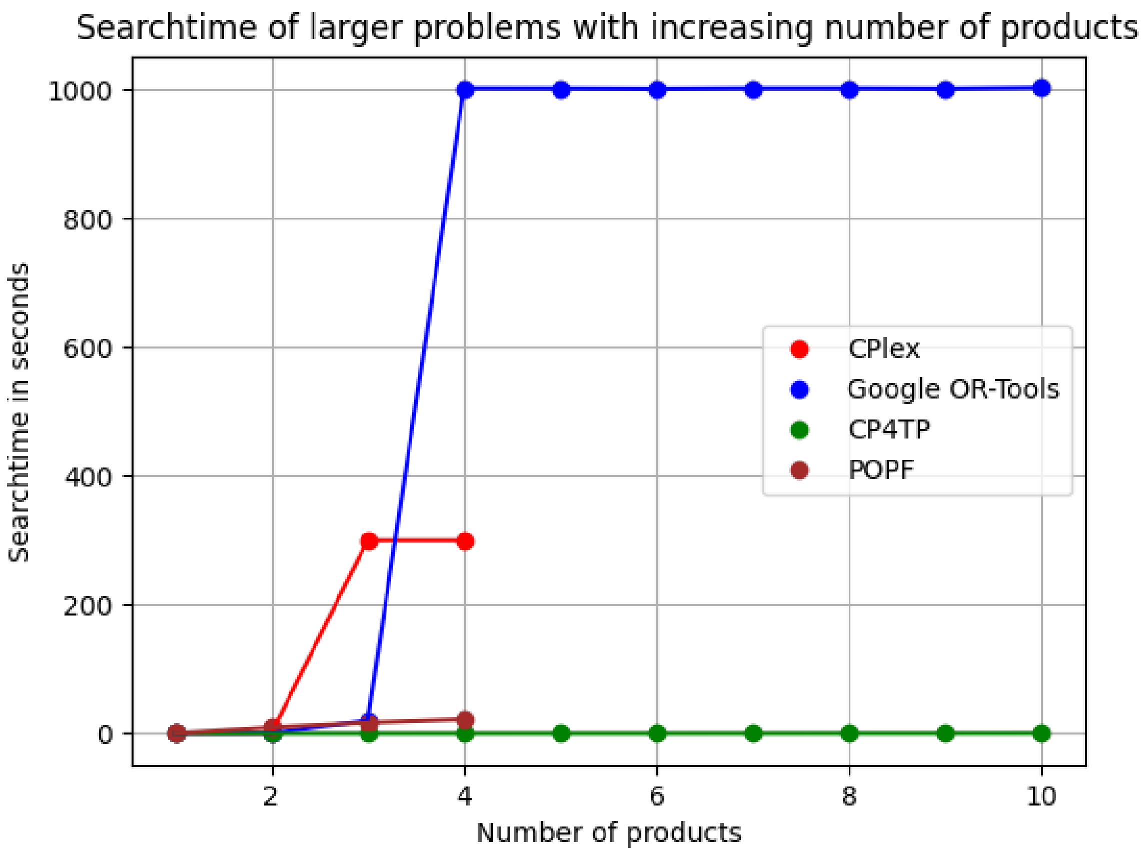

Searchtime of larger problems with increasing number of products
Searchtime in seconds
Number of products
CPlex
Google OR-Tools
CP4TP
POPF
0
200
400
600
800
1000
2
4
6
8
10

Figure 5.5:

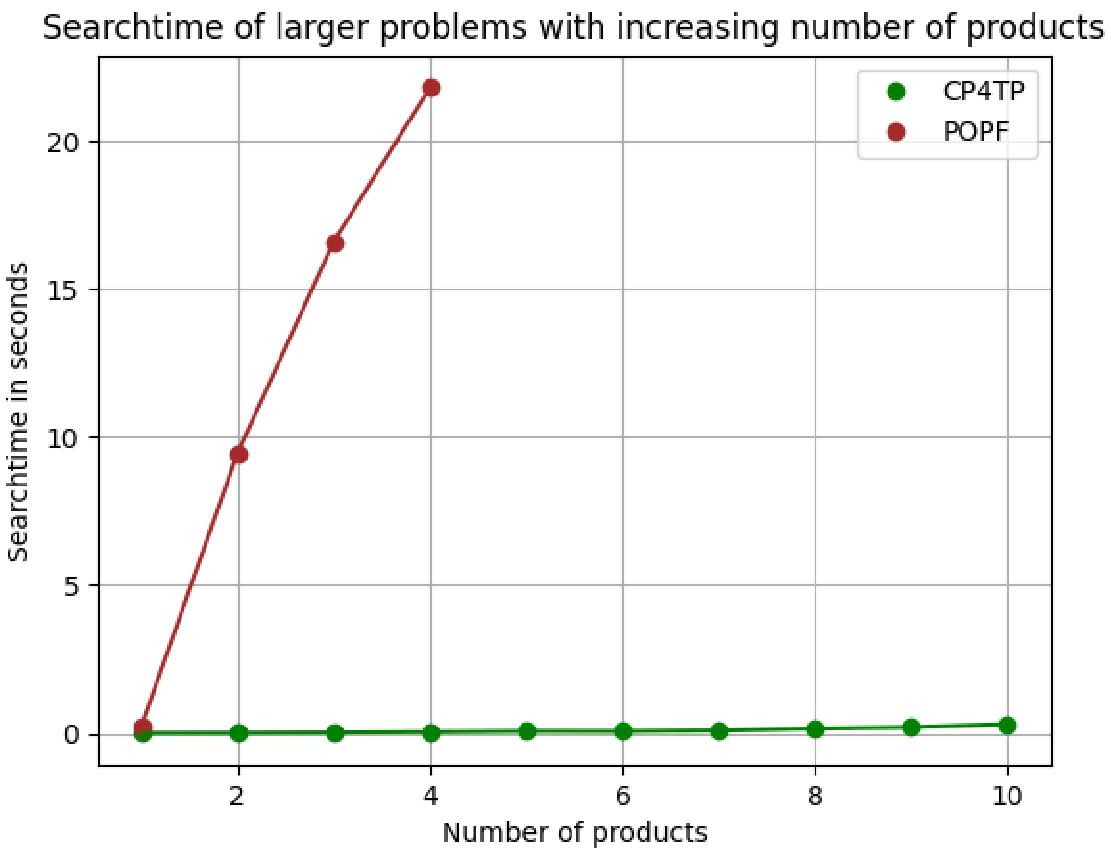

Searchtime of larger problems with increasing number of products
Searchtime in seconds
Number of products
CP4TP
POPF
0
5
10
15
20
2
4
6
8
10

Figure 5.6:

CP4TP could find solutions with the same makespan as the CSP solvers. The CP4Tp solutions are however not optimal, as the PDDL planners have more options because of the translation error. This can be observed when looking at the instances with three stations. CP4TP has a makespan of about 2270, while POPF found a solution with a makespan of 1970. This happens, because POPF does allow more parallelism. More parallelism can however also lead to worse makespans, as the instances with six stations show. Here, CP4TP has a smaller makespan than the CSP solvers, because even in sequential mode, CP4TP can use some parallelism and therefore used the translation error. POPF and TFD however, which both allow more parallelism, have a bigger makespan than CP4TP and the CSP solvers. For the biggest instances with ten stations however, more parallelism does pay of, as TFD and POPF have a smaller makespan than CP4TP. This can also be seen in graph 5.7. The makespan for all planners and solvers is similar, at different amounts of stations different planners have the advantage. This effect is however small, so that it is not clear, if this is not just coincidence.

The searchtimes, visible in table 5.6 show, that CPlex and Google OR-Tools did not use the full time for the biggest instance they could solve. CPlex needed the full time for instances with six and seven, but for the instance with eight instances, only 100.09 seconds were used. Similarly, Google OR-Tools only needed 531.3118 seconds for the largest instance in this case. This indicates, that even finding a solution for the bigger incstances here is a problem for Google OR-Tools. In the previous benchmarks, most of the time was used trying to improve the solution, but with increasing numbers of stations, Google OR-Tools needs a lot of time to even find a solution. This is in contrast to the PDDL planners. They found their solutions really fast, CP4TP never needing even 0.1 seconds for any instance. The maximum time needed for any instance was POPF for the instance with ten stations. This shows, that while the PDDL solutions might be suboptimal in some cases, the solutions are found much faster.

This aspect can be seen even better in graph 5.8. The searchtime for all PDDL planners stays near zero, while the CSP solvers need much more time for the later instances. When comparing the Planners agains each other in graph 5.9, it becomes visible, that CP4TP needs the least amount of time. TFD needs more time for the larger instances, and POPF needs more time than the others even for the smallest instance. This shows the tradeoff, that is most visible at the largest instances. POPF and TFD need more time to find a solution, but they find a better solution with a smaller makespan. With problems in this size, the extra search time needed is insignificant when comparing it with the searchtime needed by the CSP solvers. When comparing the instance with eight stations, POPF needs 26.8 times the searchtime that CP4TP needs, but CPlex needs nearly 1500 times the searchtimetime and Google OR-Tools needs nearly 8000 times the searchtime of CP4TP. It is therefore possible, that CP4TP can find solutions for much larger instances than Google OR-Tools.

5.4.4 Changing stations with larger problems

To check, if larger instances favor the PDDL planners even more, the next benchmark set again keeps the amount of workers and products the same, and increases the amount of stations. In this case, every instance had three workers and ten products. As three workers were present, the smallest instance started with three stations, with one worker at every station. The bigger instances had four, five, ten and twenty stations.

The makespan of the solutions in table 5.7 shows, that in this case, CPlex could solve the instances up to the size of five stations. After that, the input became too big. Google OR-Tools could however solve all instances, even the one with 20 stations. The same is true for CP4Tp, which could also solve all instances. TFD could solve the two smallest instances with three and four stations, and POPF the instances up to five stations. This is in contrast to the previous benchmark set, in

Number of stations	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	500	500	500.0008	500.041	500.004
2	1090	1090	1090.0036	1090	1090
3	1680	1680	1680.0078	1680	1360.006
4	2270	2270	2270.01	2270	1970.003
5	2860	2860	2860.012	2860.58	2580.003
6	4350	4350	4230.0182	4630.92	4630.03
7	5120	5120	5100.0202	5461.06	5700.031
8	5650	5660	5650.0222	6311.17	6290.032
9			6400.0242	6641.25	7060.033
10			8670.029	8331.51	7590.043

Table 5.5: Makespan of small problems with increasing number of stations

Number of stations	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	0.01	0.0143	0.004	0.00073	0.11
2	0.03	0.0168	0.00194	0.0082	0.15
3	0.51	0.0376	0.00399	0.0075	0.3
4	2.29	0.0723	0.00399	0.0245	0.51
5	42.63	2.1635	0.012	0.04897	0.91
6	300.06	56.6958	0.02	0.1973	1.06
7	300.14	100.0859	0.0359	0.3479	1.46
8	100.09	531.3118	0.0679	0.4853	1.82
9			0.06799	0.64471	2.44
10			0.096	0.7493	3.72

Table 5.6: Searchtime in seconds of small problems with increasing number of stations

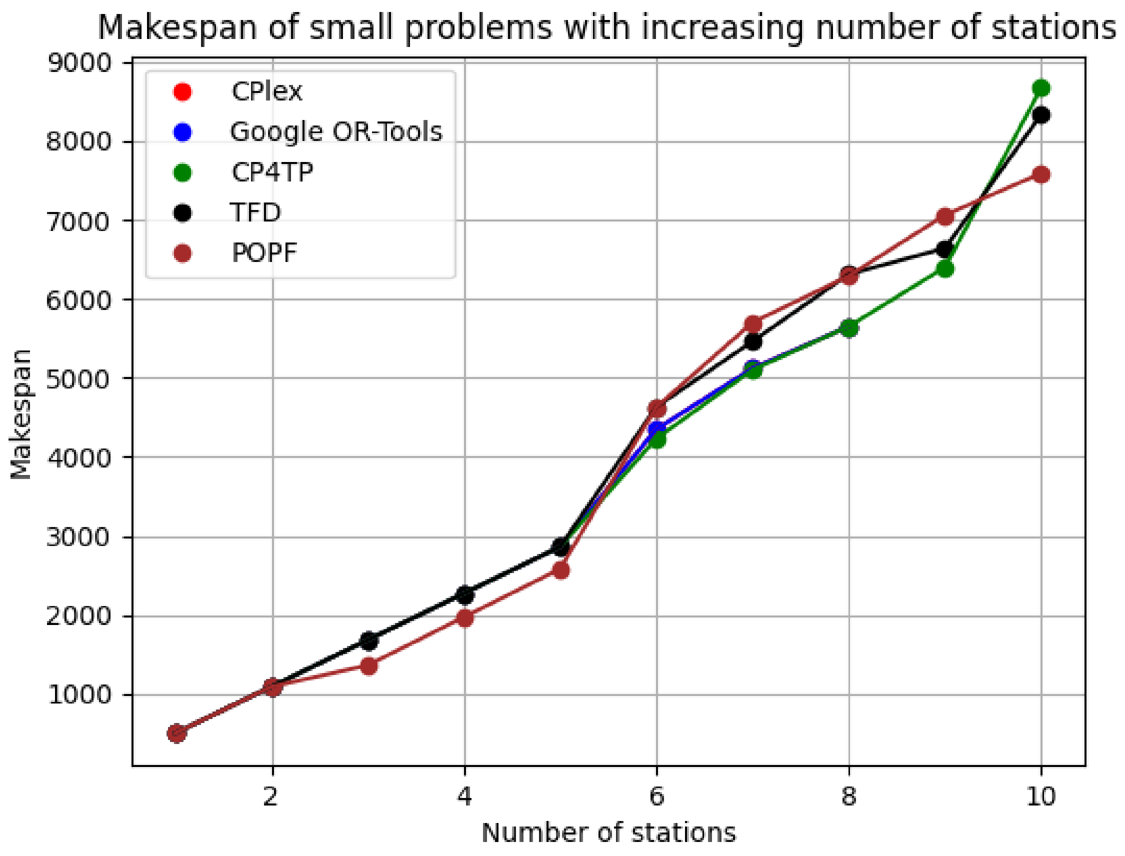

Makespan of small problems with increasing number of stations
CPlex
Google OR-Tools
CP4TP
TFD
POPF
Makespan
9000
8000
7000
6000
5000
4000
3000
2000
1000
2
4
6
8
10
Number of stations

Figure 5.7:

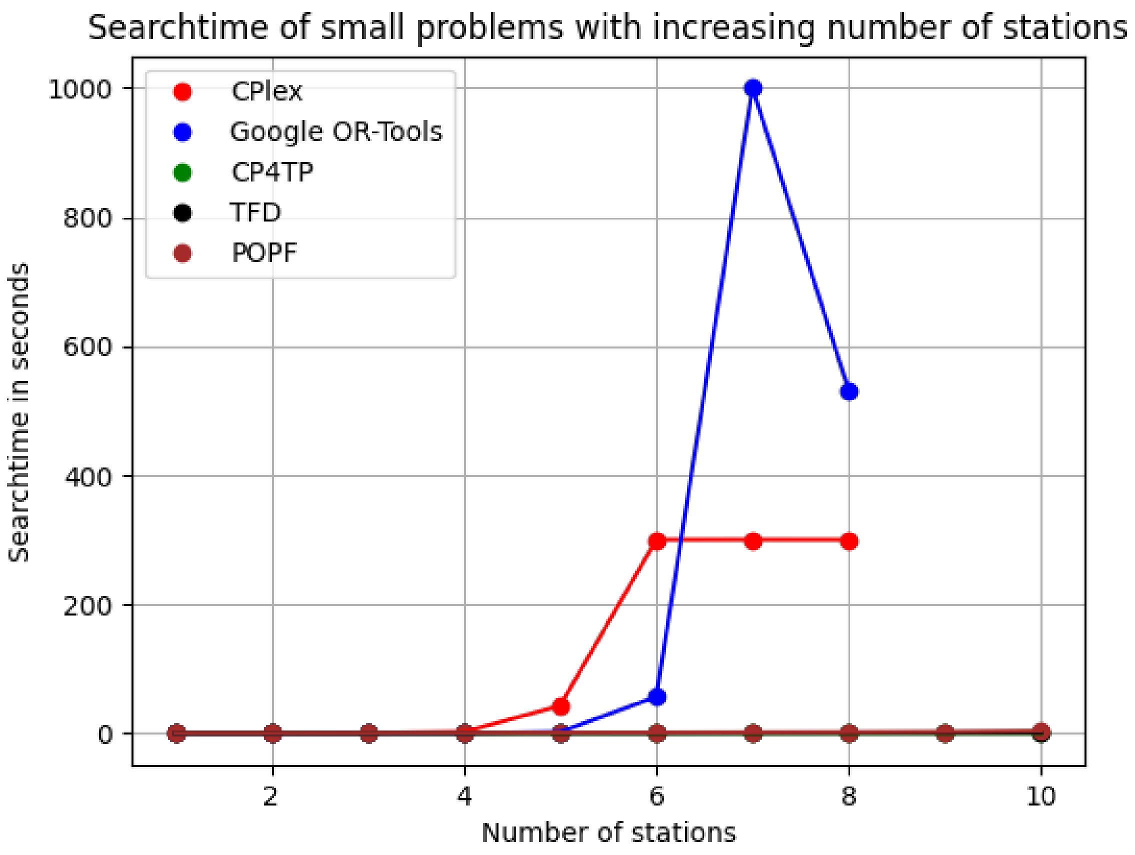

Searchtime of small problems with increasing number of stations
CPlex
Google OR-Tools
CP4TP
TFD
POPF
Searchtime in seconds
Number of stations
0
200
400
600
800
1000
2
4
6
8
10

Figure 5.8:

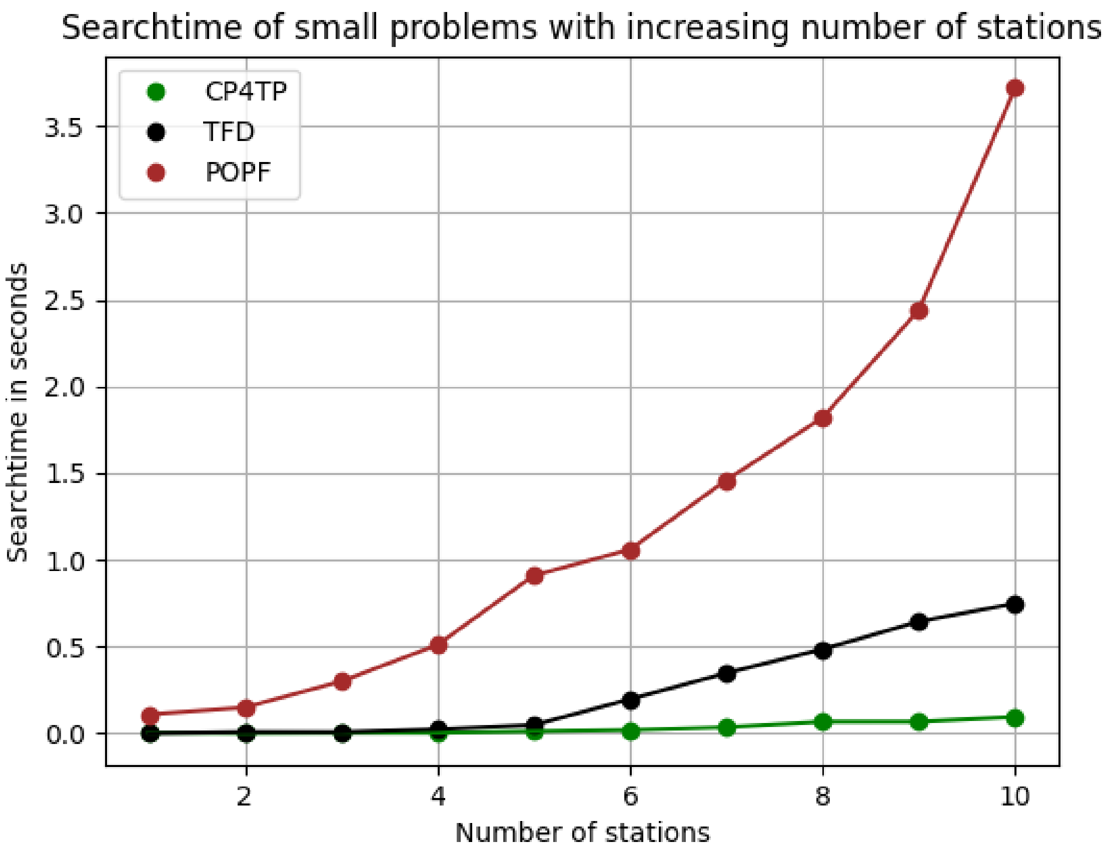

Searchtime of small problems with increasing number of stations
CP4TP
TFD
POPF
Searchtime in seconds
Number of stations
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
2
4
6
8
10

Figure 5.9:

Number of stations	CPlex	Google OR-Tools	CP4TP	TFD	POPF
3	1200	1200	1200.007	2320.74	1310.035
4	1330	1330	2290.0136	4021.21	1670.035
5	1450	1450	2810.015		2290
10		4540	10320.0318		
20		7519	17333.0548		

Table 5.7: Makespan of larger problems with increasing number of stations

which the PDDL planners could solve larger instances than both CSP solvers. Here, only CP4TP could solve all instances.

CPlex and Google OR-Tools do have the same makespan for all problems they solve, as they again solve all of them optimally. CP4TP only has this optimality in the smallest instance with three workers, after which, the makespans are much bigger than the ones of Google OR-Tools. When looking at the biggest instance with 20 stations, the makespan of CP4TP is more than double the makespan of the Google OR-Tools solution. TFD directly has a bigger makespan than CP4TP or the CSP solvers, as tHe makespan of its solution is nearly double the size of the other solutions. POPF also has bigger makespans than the CSP solvers, but with four or five stations, the makespan is smaller than the makespan of CP4TP. This happens most likely, as POPF allows more parallelism than CP4TP. This difference in makespans can be seen in graph 5.10. The difference of the makespans of CP4TP and Google OR-Tools with the big instances of ten and 20 stations becomes even more visible. Looking at the instances with the smaller amount of station in graph 5.11 shows, that even for the smaller instances, the solutions from the CSP solvers have a much smaller makespan than the plans from the PDDL solvers.

This better makespan does however again come at the cost of longer searchtimes. Looking at the table 5.8 shows, that CPlex needed the full time as soon, as four stations were part of the problem. Google OR-Tools could solve the three smaller instances in less time, but the two big ones with ten and 20 stations needed the full 1000 seconds. CP4TP did solve every instance faster than Google OR-Tools, only needing about seven seconds for the biggest instance. TFD and POPF both needed more time than Google OR-Tolls for he instances they could solve, even tough they did need less time than CPlex for the instances with four and five stations. But CPlex did find the optimal solution in much less time than the 300 seconds, the rest of the time was used trying to optimize the solution, which was not possible. The big difference in searchtime is therefore misleading here, as CPlex could have stopped much earlier. The graph 5.12 does again show these differences, as Google OR-Tools and CPlex use much more time than the PDDL planners on most of the instances. When comparing only in the PDDL solvers in graph 5.13, it becomes obvious, that CP4TP does not increase much in searchtime when compared to POPF, which has a significant rise in the for the instance with five stations, which was the last it could solve.

CP4TP trades optimality of the solution for a much faster searchtime. Even though the solutions of the biggest instances have a much bigger makespan when compared to the Google OR-Tools solution, the searchtime of CP4TP is much smaller than the one of Google OR-Tools. This is a trend that could already be seen in the other benchmarks. As the other planners can not solve the biggerinstances, it becomes obvious, that planners that allow more parallelism, have more trouble solving bigger instances. In a lot of cases however, they do find solutions with smaller makespans. This in not guarenteed, as TFD does find plans with a bigger makespan than the plans of CP4TP.

Number of stations	CPlex	Google OR-Tools	CP4TP	TFD	POPF
3	0.13	0.8086	0.0287	1.15	2.96
4	300.1	2.4872	0.0555	6.88	7.09
5	300.06	7.7146	0.113564		125.96
10		1002.4188	0.9691		
20		1002.1777	6.939		

Table 5.8: Searchtime in seconds of larger problems with increasing number of stations

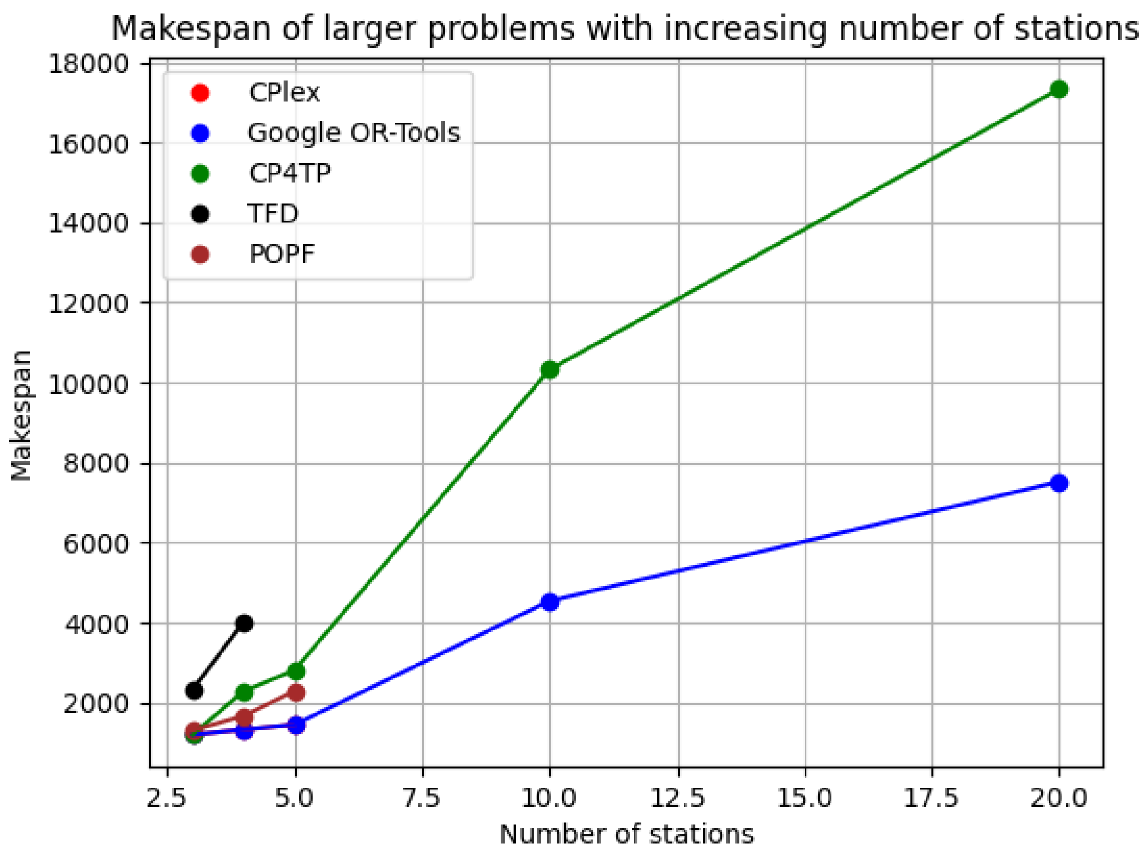


Figure 5.10:

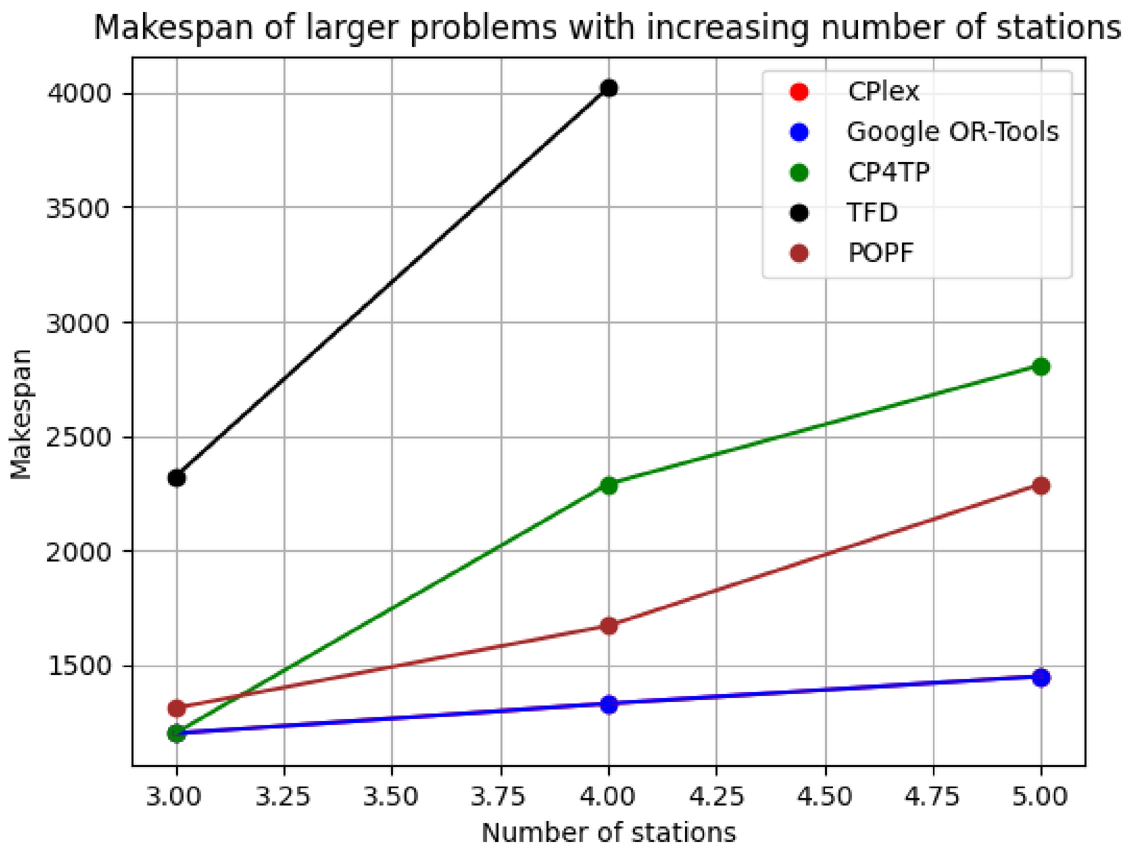

Makespan of larger problems with increasing number of stations
Makespan
4000
3500
3000
2500
2000
1500
3.00
3.25
3.50
3.75
4.00
4.25
4.50
4.75
5.00
Number of stations
CPlex
Google OR-Tools
CP4TP
TFD
POPF

Figure 5.11:

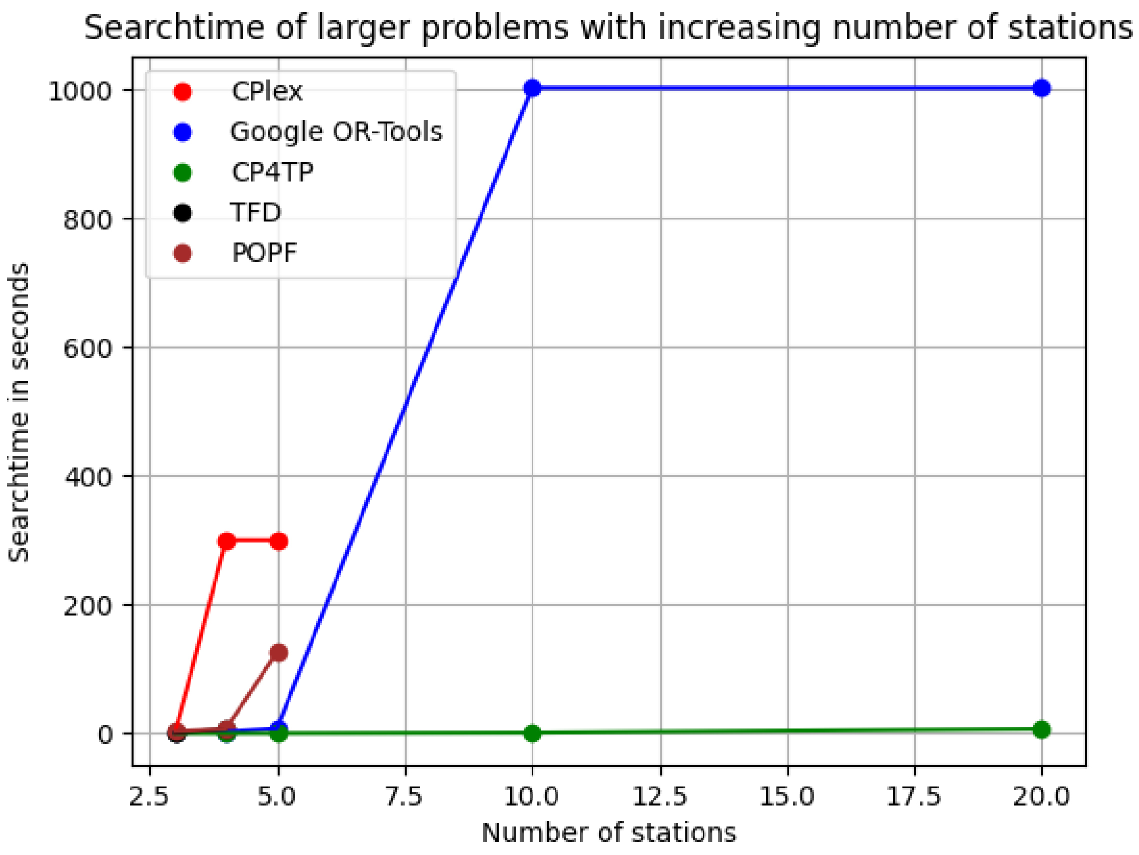

Searchtime of larger problems with increasing number of stations
CPlex
Google OR-Tools
CP4TP
TFD
POPF
Searchtime in seconds
Number of stations
0
200
400
600
800
1000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Figure 5.12:

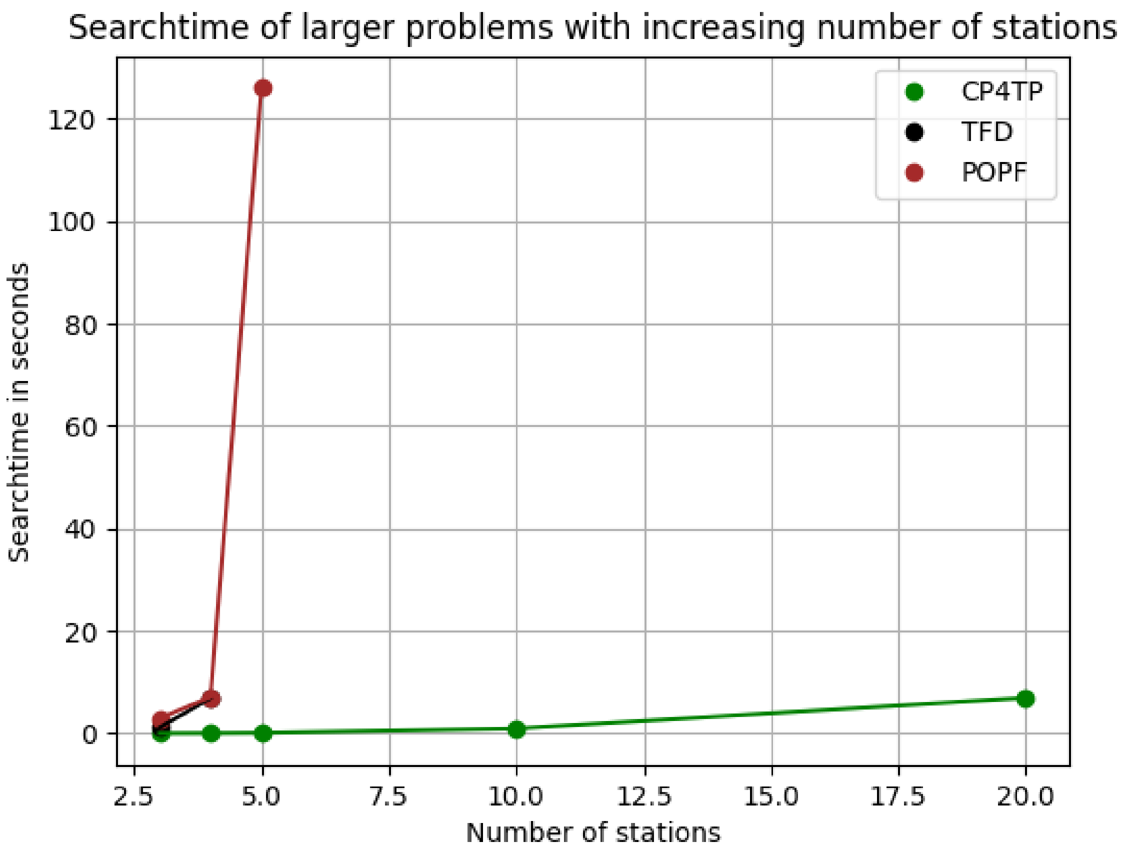

Searchtime of larger problems with increasing number of stations
CP4TP
TFD
POPF
Searchtime in seconds
0
20
40
60
80
100
120
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Number of stations

Figure 5.13:

5.4.5 Changing worker with small problems

The next benchmark set keeps the amount of stations and products the same, but increases the number of workers over time. All instances have five stations and five products, and the number of workers changes from one to five. This benchmark set is especiall interesting when looking at the parallelism. When only one worker is part of the instance, not many actions can be done in parallel. With rising numbers of workers, the amount of possilbe parallel actions increases. The full five workers is also interesting, as a worker is present at each station, no no movement is needed by any of the workers.

When looking at the makespans in table 5.9, the increase of parallelism becomes obvious. POPF cound only solve the instance with one worker, as soon as a second worker was part of the problem, POPF cound not solve the instance. All other planners and CSP solvers could solve every instance of this benchmark set. CPlex and Google OR-Tools both always have the same makespan, as both always find an optimal solution. The planners do also find an optimal solution in the instance with one worker, but as soon as multiple workers are part of the problem, both find suboptimal solutions. One noteable eception to this is the instance with as many workers as stations. In that case, TFD could also find an optimal solution and has the same makespan as the CSP solvers. The reason for this is most likely, that no worker movement was needed in this instance, as there is a worker at every station. This makes the problem drastically more easy. This can again be seen in the graph 5.14. Both CP4Tp and TFD perform worse for the bigger problems, except for the biggest one, wher TFD has the same makespan as the CSP solvers.

The searchtime here is interesting. CPLex does not need the full searchtime with only one worker, even though the needed searchtime is much bigger than every other solver or planner. If all stations have a worker, then the searchtime gets reduced dratically down to 0.09 seconds. For Google OR-Tools, a similar effect can be observed. When switching from one to two workers, the searchtime does go up, but not as much as for CPlex. After that however, more workers mean faster searchtimes. CP4TP needs less time to find a solution, the more workers are part of the instance. A reason for this could be, that, if more workers are part of the instance, less movement actions are needed. TFD on the other hand first needs more searchtime when adding more workers up to three workers, after which the searchtime reduces. A reason for this could be an overall tradeoff, as more workers mean more possible parallel actions, but also mean less movements of the worekrs needed. POPF could only solve one problem, but it needed less time than any of the CSP solvers, but still more time than the other PDDL planners. The reason for this could again be the additional parallelism that POPD allows. As soon as two workers are part of the problem, too many actions could happen in parallel, so that no solution is found. These results can again be seen in the graphs, where graph 5.15 Shows, how much more time CPlex needed in comparison to the other solvers and planners. Graph 5.16 shows the same graph without CPlex, and showcases, that CP4TP needs much less time for most of the problems. More importantly, the time needed stays nearly constant. An assumption that could be made here is, that CP4TP could also solve much bigger instances, even though not optimally.

5.4.6 Changing worker with larger problems

The assumption, that CP4TP can also solve bigger instances can be tried on the next benchmark set. In this set, every instance had ten stations and ten products. The different instances had different amount of workers, starting with one and going up to five. One additional instance had ten workers.

Table 5.17 shown, that for this benchmark set, TFD could solve no instance, and POPF could only

Number of workers	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	2860	2860	2860.012	2860.58	2580.003
2	1320	1320	2180.0072	2120.62	
3	1030	1030	1580.0048	2520.74	
4	940	940	1760.004	2030.73	
5	900	900	1300.0028	900.188	

Table 5.9: Makespan of small problems with increasing number of workers

Number of workers	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	46.21	2.0455	0.012	0.04	0.89
2	300.01	6.9292	0.012	1.64	
3	300.01	0.9184	0.008	3.97	
4	300.01	0.8317	0.008	0.4	
5	0.09	0.3349	0.004	0.03	

Table 5.10: Searchtime in seconds of small problems with increasing number of workers

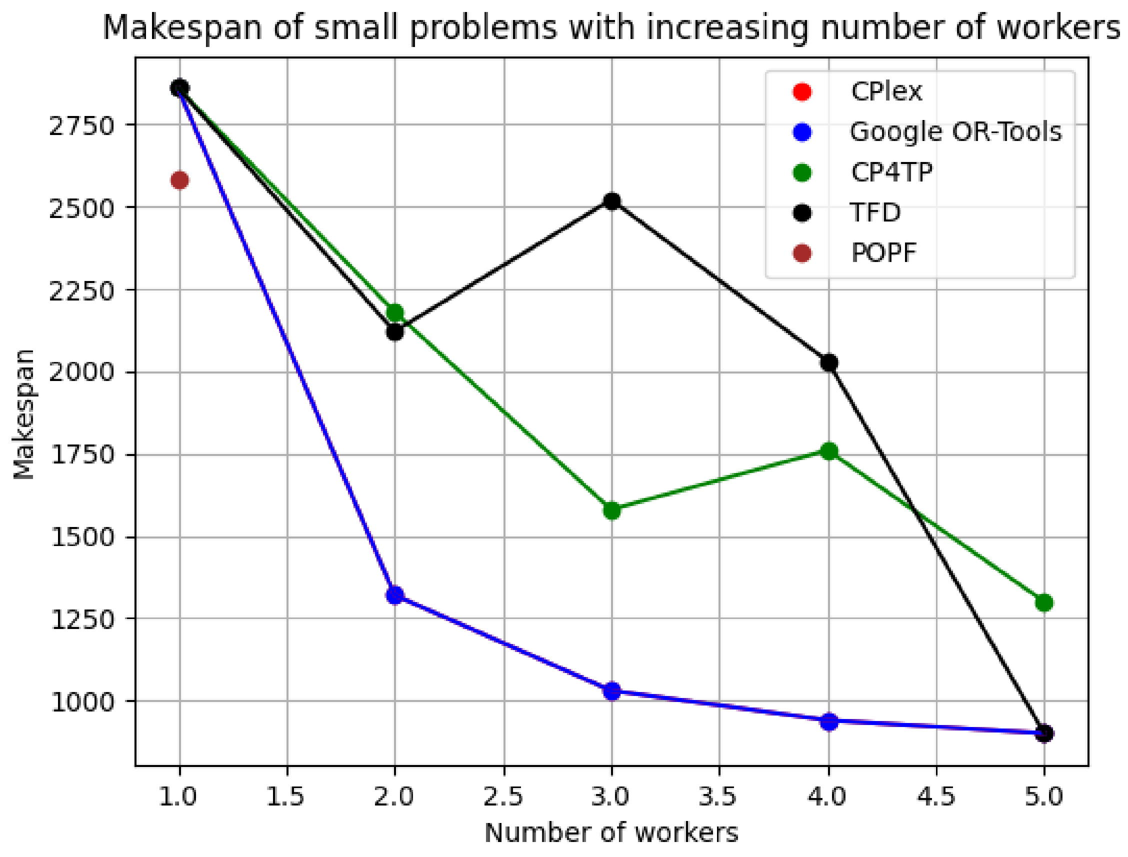
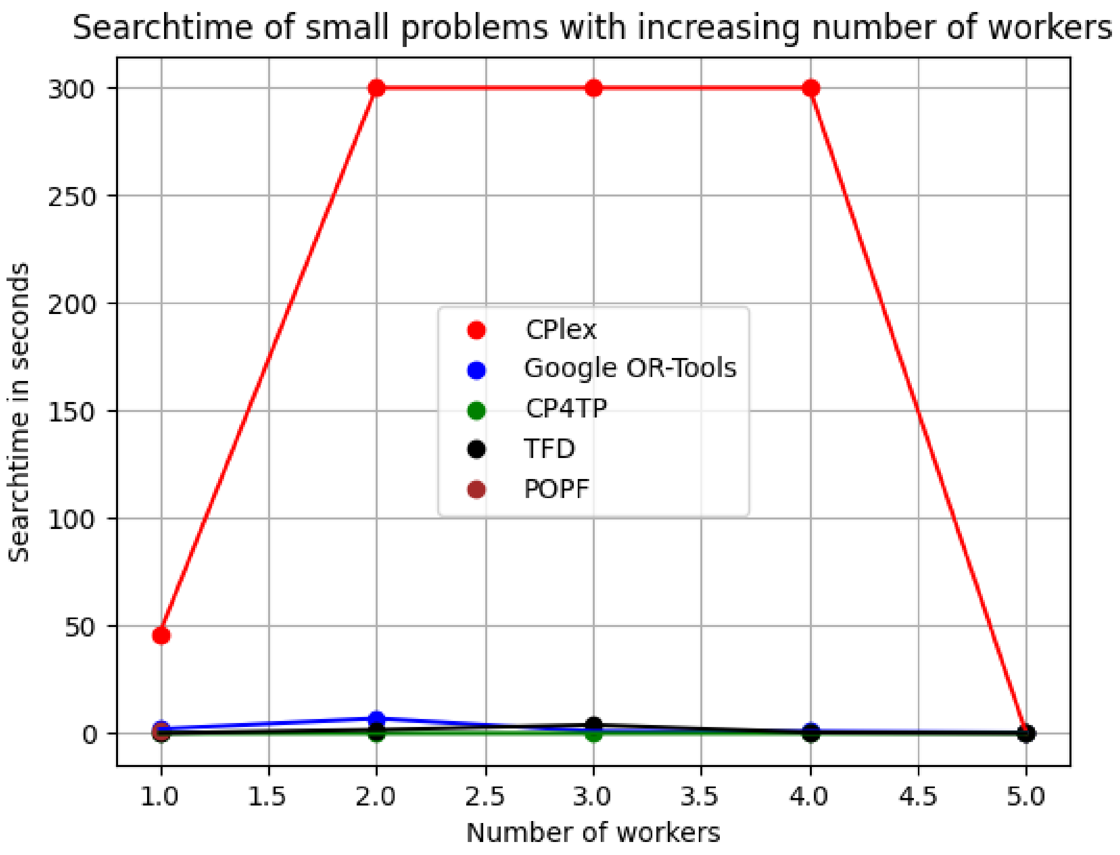


Figure 5.14:



Searchtime of small problems with increasing number of workers
Searchtime in seconds
Number of workers
CPlex
Google OR-Tools
CP4TP
TFD
POPF

Figure 5.15:

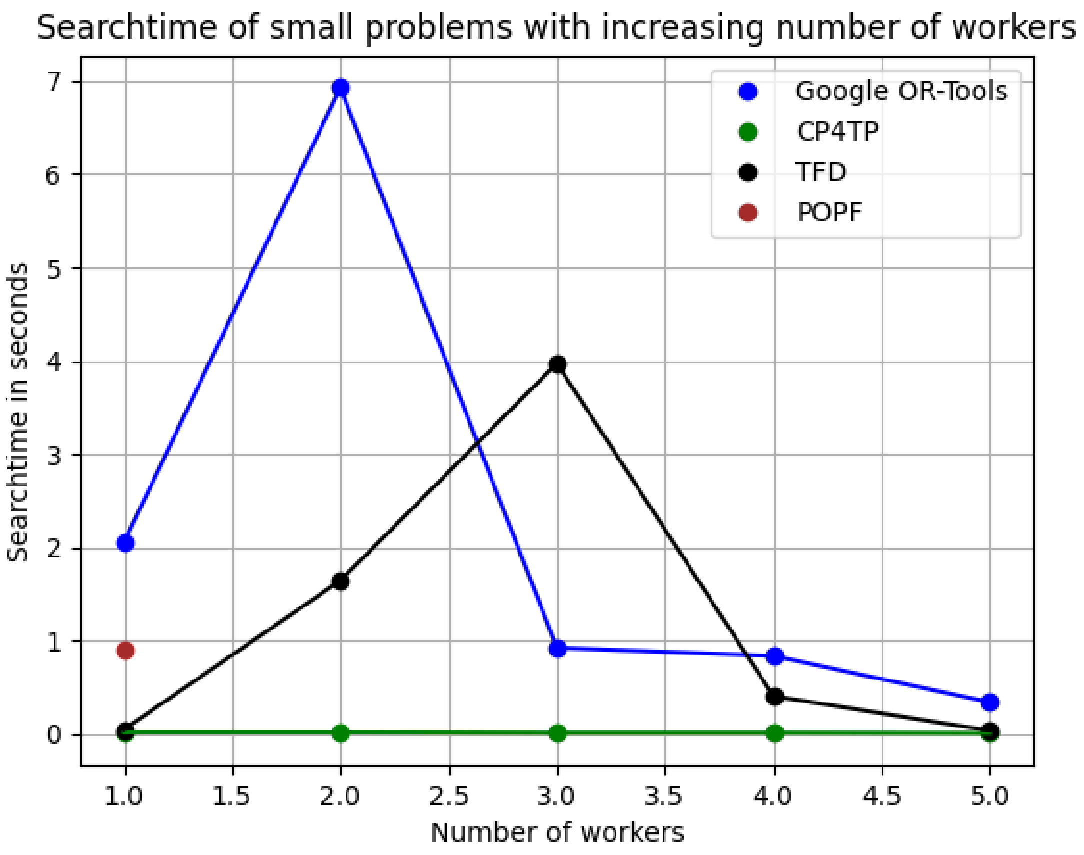

Searchtime of small problems with increasing number of workers
Google OR-Tools
CP4TP
TFD
POPF
Searchtime in seconds
0
1
2
3
4
5
6
7
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Number of workers

Figure 5.16:

Number of workers	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	13510	13510	8660.036		14650.003
2	6420	6910	16230.0452		
3	4220	4720	14340.0326		
4	3300	3650	9580.0244		
5		2980	7140.0166		
10		1900	2800.0058		

Table 5.11: Makespan of larger problems with increasing number of workers

Number of workers	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1	300.15	1000.3	0.3511		44.64
2	300.07	999.046	0.4000886		
3	300.04	1000.4617	0.31888		
4	300.06	1001.3445	0.370885		
5		100.4745	0.21866		
10		41.0978	0.135342		

Table 5.12: Searchtime in seconds of larger problems with increasing number of workers

solve the first instance with one worker. CP4TP and Google OR-Tools could solve all instances, while CPlex could not solve the two biggest instances with five and ten workers respectively. One important observation her is, that Google OR-Tools could not find the optimal solution in all instances here. This can be seen in the instance with four workers, as the CPlex solution is better than the Google OR-Tools solution. The solutions are however much better than the CP4TP solutions, or the one solution of POPF. One exception to this is the first instance, where CP4TP could find a solution with a smaller makespan than the solutions from CPlex or Google OR-Tools. This becomes more visible when looking at the graph 5.17. One interesting aspect here is, that more workers should only reduce the makespan, as one more worker does allow more parallel actions. But CP4TP's solution for two, three and four workers has a bigger makespan than the solution for one worker. This shows, how unoptimal these solutions are, as they should only decrease with the number of workers.

The searchtime in table 5.12 shows, that CPlex always used all the given time. Google OR-Tools also used the maximum time for most of the instances, with the exception of the last instance with ten workers, where only 41 seconds were needed. In contrast to this, CP4TP could solve all instances in under 0.5 seconds. POPF needed 44.64 seconds for the one instance it could solve, less than both CSP solvers, but their searchtime is again inflated by the time they tried to optimize their solutions while already having found an optimal solution. The searchtimes can also be seen in graph 5.18. Every solver and planner needs nearly the same time for all instances, with the exception of Google OR-Tools, which can solve the last instance faster.

This benchmark set again shows, that CP4TP is the only planner, that can solve the biggest problems. Both, TFD and POPF can not find solutions for bigger instances. The searchtime for CP4TP is also much better than the searchtime of the CSP solvers, but this better time comes at the cost of solutions with a bigger makespan.

5.4.7 Large problems

The last benchmark set contains five big instances. The instances have the following characteristics:

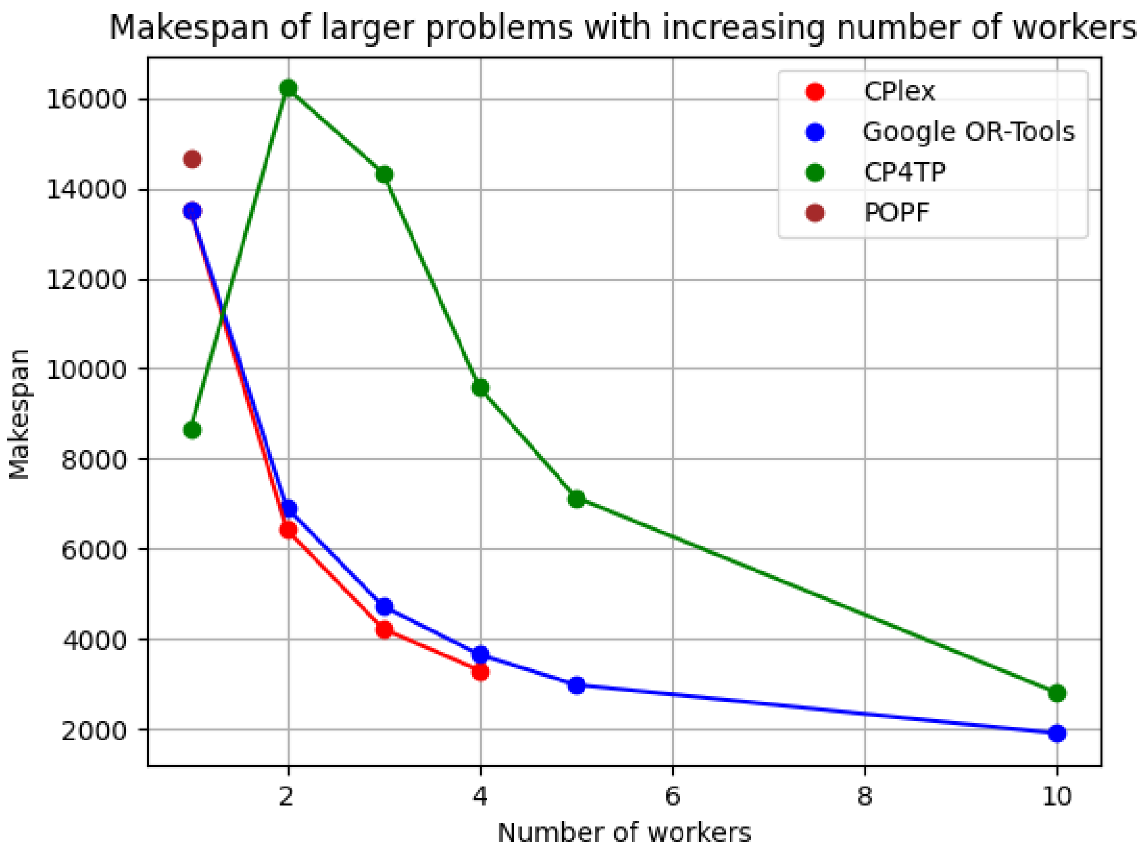

Makespan of larger problems with increasing number of workers
Makespan
16000
14000
12000
10000
8000
6000
4000
2000
2
4
6
8
10
Number of workers
CPlex
Google OR-Tools
CP4TP
POPF

Figure 5.17:

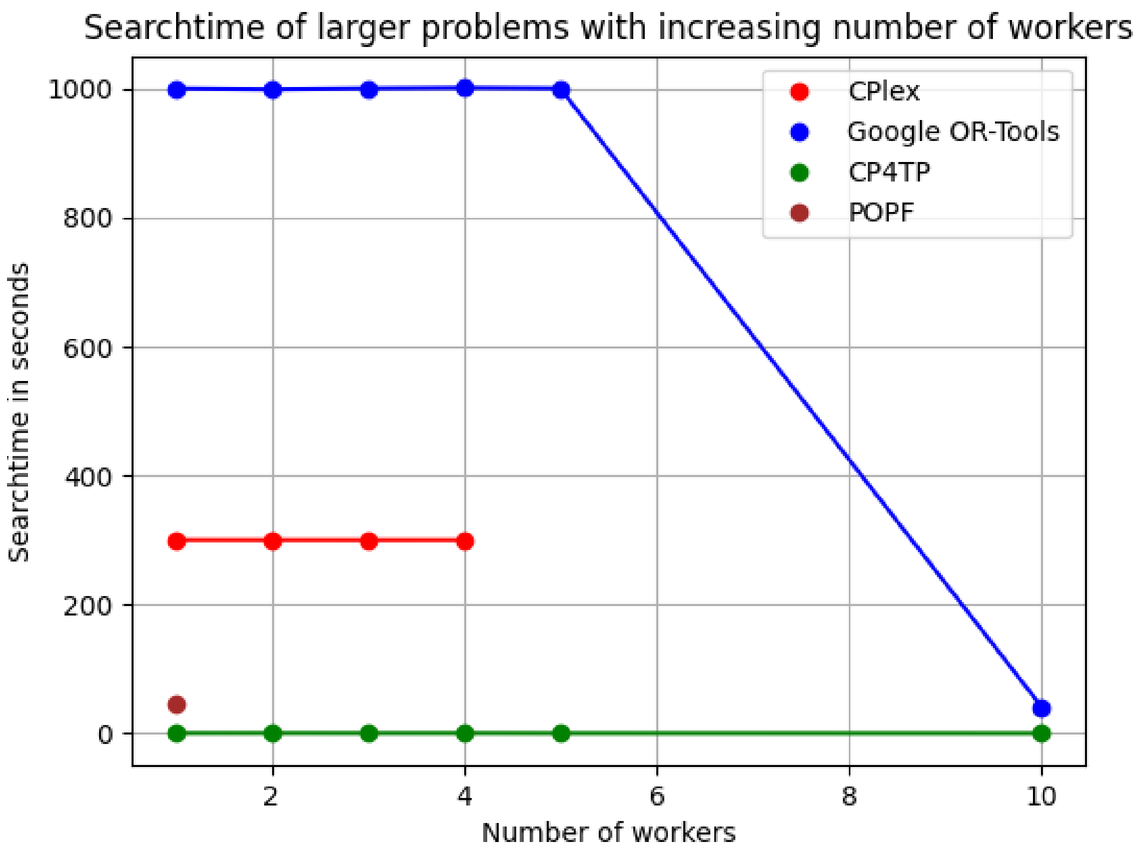

Searchtime of larger problems with increasing number of workers
CPlex
Google OR-Tools
CP4TP
POPF
Searchtime in seconds
Number of workers
0
200
400
600
800
1000
2
4
6
8
10

Figure 5.18:

Problem number	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1		7107	13151.01		
2		9234	24055.0332		
3		8314	21256.0756		
4			39435.0344		
5			12535.0098		

Table 5.13: Makespan of large problems

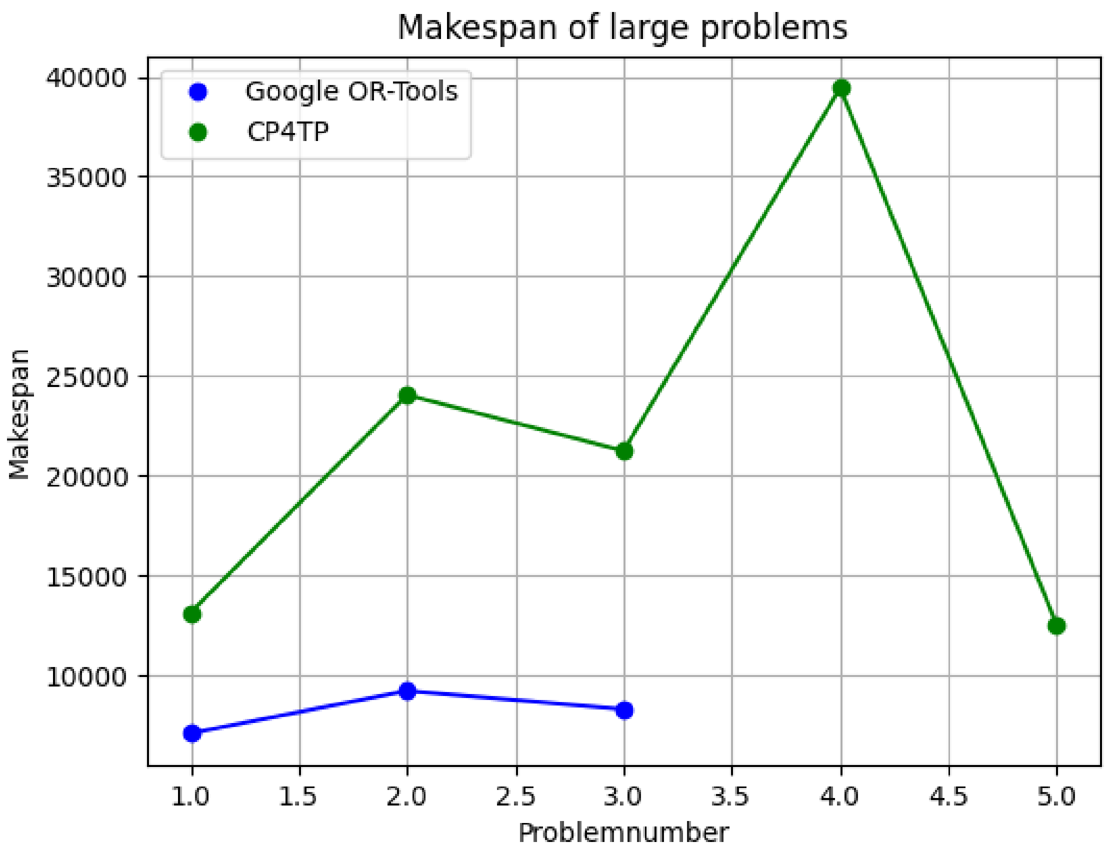
Problem number	CPlex	Google OR-Tools	CP4TP	TFD	POPF
1		1010.4826	0.4293		
2		1004.744	2.38557		
3		1007.6657	36.5136		
4			28.8961		
5			2.5373		

Table 5.14: Searchtime in seconds of large problems

- **1**: ten workers, ten stations and ten products
- **2**: five workers, ten stations and 20 products
- **3**: five workers, 20 stations and 15 products
- **4**: ten workers, 20 stations and 15 products
- **5**: 20 workers, 20 stations and 15 products

The makespans in table 5.13 show, that only Google OR-Tools andCP4TP could solve any of these instances. CPlex could not solve them, as the input was too big, and TFD and POPF did not find a solution in time. Even though Google OR-Tools did find some solutions, but the biggest instances could not be solved. In contrast to this, CP4TP could solve all of the instances, even though the makespan was much worse for all of them when compared to the makespan of the Google OR-Tools solution. In instance three for example, the makespan is nearly three times as big as the makespan of the Google Or-Tools solution. This difference can be seen more clearly in graph 5.19. CP4TP finds solutions that have a much bigger makespan than the solutions of Google OR-Tools.

In contrast to this, the searchtime of CP4TP is much better than the searchtime of Google OR-Tools. They can be seen in table 5.14 CP4Tp needed, at most, 36.5136 seconds, while Google OR-Tools always used the full 1000 seconds. This difference is also visible in the graph 5.20. Google OR-Tools always needed much more time than CP4TP. This again shows the tradeoff, that CP4TP can find bigger solutions and find them faster, but the solution the planner finds have a worse makespan than the solutions of the CSP solvers.



Makespan of large problems
Google OR-Tools
CP4TP
Makespan
Problemnumber
40000
35000
30000
25000
20000
15000
10000
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

Figure 5.19:

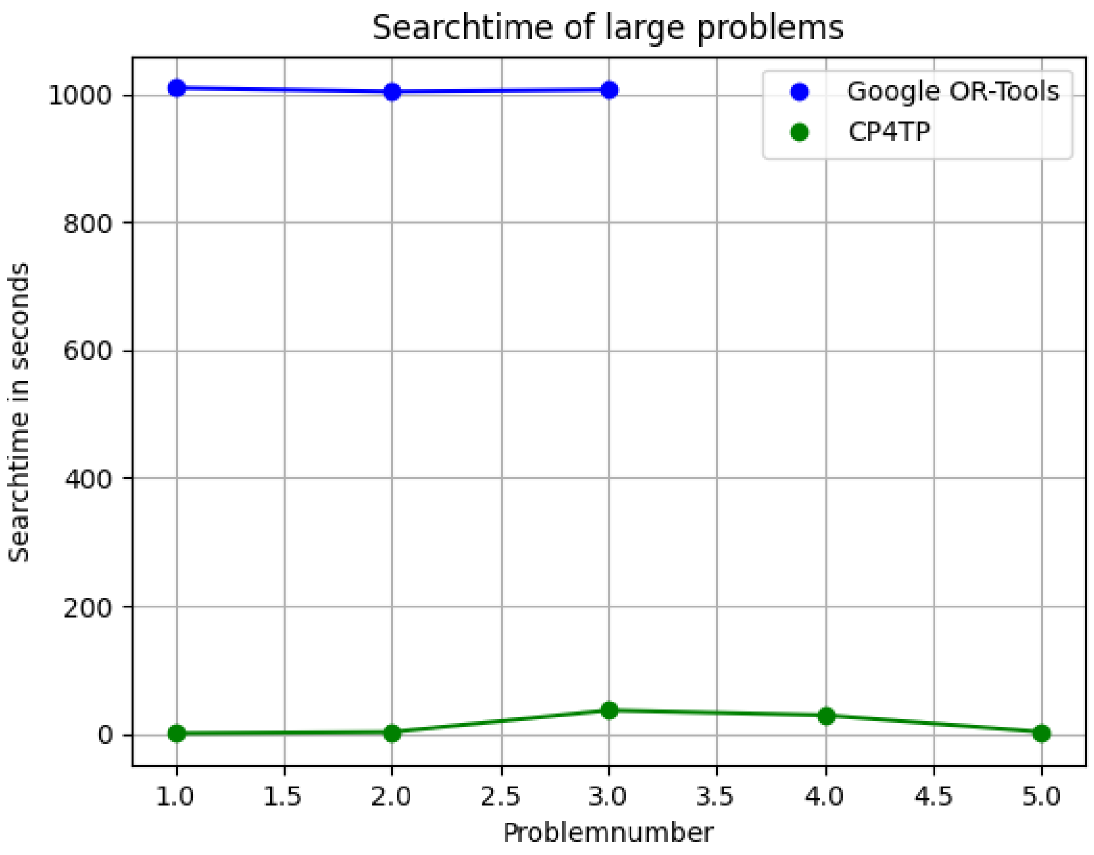

Searchtime of large problems
Google OR-Tools
CP4TP
Searchtime in seconds
1000
800
600
400
200
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Problemnumber

Figure 5.20:

6 Conclusion

The goal of this thesis was to see, if it is possible, to translate a problem modeled in CSP to PDDL. For this purpose, the Flow Sop problem was modeled in PDDl. This problem was already modelled as CSP models. Multiple instances were automatically translated into PDDL, and were then solved by multiple PDDL planners to see, how good this translation was solvable.

The translation did show, that problems can arise when trying to translate a problem, given as a CSP model, to PDDL. In this case, the problem was, that the problem enforced, that an action must be immediately taken, which is not directly possible to enforce in PDDL. Apart from this problem, the problem could be modelled correctly in PDDL. The single instances could also be automatically generated from files meant for the CSP model. The PDDL model did have some advantages over the CSP model. The examples shown in this thesis were a better readability and a good adaptability.

The experiments with different CSP solvers and PDDL planners did show, that both models can be used to find solutions. The CSP solvers did, in most cases, find solutions with a smaller makespan. The PDDL solutions had a larger makespan in most cases. One reason for this was, that they did not allow much parallel actions, which were needed for a solution with smaller makespan. If a planner did allow more parallel actions, then it got solutions with smaller makespans in most cases. As a tradeoff, they needed more time to find these solutions, and could not solve a lot of instances, planners that created sequential plans could solve. CP4TP, a planner that first tries to create a sequential plan, could even solve bigger instances of the Flow Shop problem than any of the CSP solvers.

Overall, modeling the problem in PDDL resulted in suboptimal solution, but these solutions were found fast. An important aspect of this was, that in the flow Shop problem, a lot of actions can happen in parallel. Therefore, such an translation into PDDL is possible, but it is more useful with problems, where no parallel actions are needed. In these cases, durative actions would not be needed, so that planners that do not support durative actions can be used. These planners can then have more desired features, such as finding optimal solutions.

Bibliography

[1] M. L.Pinedo, *Scheduling: Theory, Algorithms, and Systems (Fifth Edition)*. Springer, 2016.

[2] I. Benkalai, D. Rebaine, and P. Baptiste, “Scheduling flow shops with operators,” *International Journal of Production Research*, vol. 57, no. 2, pp. 338–356, 2019.

[3] V. S. Tanaev, Y. N. Sotskov, and V. A. Strusevich, *Scheduling Theory. Multi-Stage Systems*. Springer, 1994.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson, 2016.

[5] J. Hoffmann, D. Fiser, D. Höller, and S. Saller, “Artificial intelligence 6. csp part1: Basics and naive search.” University Lecture, 2022.

[6] “Google or-tools website.” `https://developers.google.com/optimization/cp`. Accessed: 2022-07-22.

[7] “Google or-tools website describing propagation and backtracking.” `https://developers.google.com/optimization/cp/queens#4queens`. Accessed: 2022-07-22.

[8] “Ibm website for cplex.” `https://www.ibm.com/de-de/analytics/cplex-optimizer`. Accessed: 2022-07-22.

[9] “Minizinc website.” `https://www.minizinc.org/`. Accessed: 2022-07-22.

[10] smfs4, “Workerflowshop.” `https://github.com/smfs4/WorkerFlowShop`, 2022.

[11] D. N. Malik Ghallab and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[12] R. E. Fikes and N. J. Nilsson, “STRIPS: A new approach to the application of theorem proving to problem solving,” *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[13] M. et. al., “PDDL - the planning domain definition language,” *he AIPS-98 Planning Competition Comitee*, 1998.

[14] M. Fox and D. Long, “PDDL2.1: An extension to PDDL for expressing temporal planning domains,” *Journal of Aritificial Intelligence Research (JAIR)*, vol. 20, pp. 61–124, 12 2003.

[15] B. Wally, J. Vyskočil, P. Novák, C. Huemer, R. Šindelář, P. Kadera, A. Mazak, and M. Wimmer, “Production planning with iec 62264 and pddl,” in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1, pp. 492–499, 2019.

[16] A. Billman, “Solving flow shop problems using a forward-chaining partial-order planner,” 2015. Bachelors thesis at Mälardalens Högskola School of Innovation, Design and Technology.

[17] D. Nirwan, “AI planning using constraint satisfaction problems.” `https://towardsdatascience.com/ai-planning-using-constraint-satisfaction-problems-eb1be5466af6`, 2021. Accessed: 2022-05-29.

[18] "Robocup logistics league website." `http://www.robocup-logistics.org/`. Accessed: 2023-01-17.

[19] G. L. Tim Nielmueller and A. Ferrein, "The robocup logistics league as a benchmark for planning in robotics," *Planning and Robotics (PlanRob-15)*, 2015.

[20] M. Bofill, J. Suy, and M. Villaret, "A system for solving constraint satisfaction problems with smt," 2010.

[21] "International planning competition 2018." `https://ipc2018.bitbucket.io/`. Accessed: 2023-01-15.

[22] "Planutils github." `https://github.com/AI-Planning/planutils`. Accessed: 2023-01-15.

[23] I. Cenamor, T. de la Rosa, M. Vallati, F. Fernández, and L. Chrpa, "Temporal: Temporal portfolio algorithm," *Proceedings of ICAPS International Planning Competition*, 2018.

[24] D. Furelos-Blanco and A. Jonsson, "Cp4tp: A classical planning for temporal planning portfolio," 2018.

[25] A. Coles, A. Coles, M. Fox, and D. Long, "Forward-chaining partial-order planning," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 20, pp. 42–49, May 2021.

[26] P. Eyerich, R. Mattmüller, and G. Röger, "Using the context-enhanced additive heuristic for temporal and numeric planning," 2009.

10% Overall Similarity

Top sources found in the following databases:

- 9% Internet database
- 7% Publications database
- Crossref database
- Crossref Posted Content database
- 1% Submitted Works database

TOP SOURCES

The sources with the highest number of matches within the submission. Overlapping sources will not be displayed.

1	http://134.60.68.50/ki/Edu/Vorlesungen/GdKI/WS0203/pddl.pdf Internet	<1%
2	toplinessolution.com Internet	<1%
3	projects.laas.fr Internet	<1%
4	nt.uni-saarland.de Internet	<1%
5	link.springer.com Internet	<1%
6	strathprints.strath.ac.uk Internet	<1%
7	inf.ed.ac.uk Internet	<1%
8	dspace.cvut.cz Internet	<1%

9	epdf.pub Internet	<1%
10	plg.inf.uc3m.es Internet	<1%
11	kwarc.info Internet	<1%
12	web.archive.org Internet	<1%
13	kclpure.kcl.ac.uk Internet	<1%
14	fai.cs.uni-saarland.de Internet	<1%
15	courses.cs.washington.edu Internet	<1%
16	pecan.srv.cs.cmu.edu Internet	<1%
17	essays.se Internet	<1%
18	mcc.ufc.br Internet	<1%
19	tandfonline.com Internet	<1%
20	informatik.uni-ulm.de Internet	<1%

21	Coles, A.. "Managing concurrency in temporal planning using planner-s... Crossref	<1%
22	essay.utwente.nl Internet	<1%
23	Michael L. Pinedo. "Scheduling", Springer Science and Business Media ... Crossref	<1%
24	mccormick.northwestern.edu Internet	<1%
25	mafiadoc.com Internet	<1%
26	Michael Cashmore, Maria Fox, Tom Larkworthy, Derek Long, Daniele M... Crossref	<1%
27	ais.informatik.uni-freiburg.de Internet	<1%
28	cms.sic.saarland Internet	<1%
29	rairo-ro.org Internet	<1%
30	pdfs.semanticscholar.org Internet	<1%
31	Antonio Garrido, Eva Onaindia. "chapter 2 Extending Classical Planning... Crossref	<1%
32	waseda.repo.nii.ac.jp Internet	<1%

33	cs-cs.yale.edu Internet	<1%
34	imae.udg.edu Internet	<1%
35	saarlandes on 2020-12-16 Submitted works	<1%
36	mrg.dist.unige.it Internet	<1%
37	Fabian Peller, Mirko Wachter, Markus Grotz, Peter Kaiser, Tamim Asfo... Crossref	<1%
38	Julia Schmitt, Michael Roth, Rolf Kiefhaber, Florian Kluge, Theo Ungere... Crossref	<1%
39	d-nb.info Internet	<1%
40	deepai.org Internet	<1%
41	dokumen.pub Internet	<1%
42	dspace.lboro.ac.uk Internet	<1%
43	Lai Tsung-Chyan, Y.N. Sotskov, N.Yu. Sotskova, F. Werner. "Optimal m... Crossref	<1%
44	docplayer.net Internet	<1%

45	ifaamas.org Internet	<1%
46	tidel.mie.utoronto.ca Internet	<1%
47	Bozejko, W.. "Solving the flow shop problem by parallel programming", ... Crossref	<1%
48	José Alexandre de Souza Crippa, Antonio Waldo Zuardi, Geraldo F. Bus... Crossref	<1%
49	Zeid Kootbally. "Industrial robot capability models for agile manufactur... Crossref	<1%
50	nozdr.ru Internet	<1%
51	rtime.ciirc.cvut.cz Internet	<1%
52	"Metaheuristics for Production Scheduling", Wiley, 2013 Crossref	<1%
53	ideals.illinois.edu Internet	<1%
54	semanticscholar.org Internet	<1%
55	Gaby Pinto, Yariv T. Ben-Dov, Gad Rabinowitz. "Formulating and solvin... Crossref	<1%
56	Giuseppe DellaPenna, Daniele Magazzeni, Fabio Mercorio. "A universal... Crossref	<1%

57	Kerem Bülbül. "Flow shop scheduling with earliness, tardiness, and inte... Crossref	<1%
58	Li Xu. "Backward-Chaining Flexible Planning", Lecture Notes in Comput... Crossref	<1%
59	Sapena, O.. "A distributed CSP approach for collaborative planning sys... Crossref	<1%
60	Stefan-Octavian Bezrucav, Burkhard Corves. "Modelling Automated Pla... Crossref	<1%
61	Convex Functions and Optimization Methods on Riemannian Manifolds... Crossref	<1%
62	archive.org Internet	<1%
63	ntrs.nasa.gov Internet	<1%
64	Niklas Sebastian Eltester, Alexander Ferrein, Stefan Schiffer. "A Smart ... Crossref	<1%
65	diva-portal.org Internet	<1%
66	citeseerx.ist.psu.edu Internet	<1%
67	discovery.ucl.ac.uk Internet	<1%
68	mail.prz-rzeszow.pl Internet	<1%

69	psychology.wikia.com Internet	<1%
70	Lecture Notes in Computer Science, 2010. Crossref	<1%
71	Liji Shen, Jatinder N. D. Gupta. "Family scheduling with batch availabilit... Crossref	<1%
72	Ribas, I.. "Review and classification of hybrid flow shop scheduling pro... Crossref	<1%
73	Safa Hachani. "Business Process Flexibility in Service Composition: Ex... Crossref	<1%
74	cognitivesystems.org Internet	<1%
75	dissertations.umi.com Internet	<1%
76	icaps17.icaps-conference.org Internet	<1%
77	ndl.ethernet.edu.et Internet	<1%
78	ras.papercept.net Internet	<1%
79	F. Parreño. "Ahybrid GRASP/VND algorithm fortwo-andthree-dimensio... Crossref	<1%
80	Lecture Notes in Computer Science, 2012. Crossref	<1%

81	Lee, W.C.. "A two-machine flowshop problem with two agents", Compu... Crossref	<1%
82	Lukáš Chrpa, Daniele Magazzeni, Keith McCabe, Thomas L. McCluskey... Crossref	<1%
83	Natalia Shakhlevich. "Minimizing total weighted completion time in a p... Crossref	<1%
84	Oakes, William C.. "Engineering Your Future", Oxford University Press Publication	<1%
85	Peter Brucker, Silvia Heitmann, Johann Hurink. "Flow-shop problems w... Crossref	<1%
86	dspace.mit.edu Internet	<1%
87	pdffox.com Internet	<1%
88	cs.ust.hk Internet	<1%
89	csd.abdn.ac.uk Internet	<1%
90	ebooks.cambridge.org Internet	<1%
91	hybrid-reasoning.org Internet	<1%
92	researchgate.net Internet	<1%

93	teachmint.com Internet	<1%
94	www2.informatik.uni-freiburg.de Internet	<1%
95	Allaoui, H.. "Scheduling two-stage hybrid flow shop with availability co... Crossref	<1%
96	Ana G. D. Correa, Laisa C. C. De Biase, Erich P. Lotto, Roseli D. Lopes. "... Crossref	<1%
97	B. Baki, M. Bouzid, A. Ligęza, A. I. Mouaddib. "A centralized planning te... Crossref	<1%
98	F. Maris. "TLP-GP: New Results on Temporally-Expressive Planning Be... Crossref	<1%
99	Jacob W. Crandall. "When autonomous agents model other agents: An ... Crossref	<1%
100	Jen-Shiang Chen, Jin-Shan Yang. "Model formulations for the machine ... Crossref	<1%
101	Moritz Tenorth, Michael Beetz. "Representations for robot knowledge i... Crossref	<1%
102	Scheduling Algorithms, 2004. Crossref	<1%
103	christian.buerckert.eu Internet	<1%
104	jair.org Internet	<1%

105	kluedo.ub.uni-kl.de Internet	<1%
106	par.nsf.gov Internet	<1%
107	pure.uva.nl Internet	<1%
108	saarlandes on 2021-02-09 Submitted works	<1%
109	"Towards Service Robots for Everyday Environments", Springer Scienc... Crossref	<1%
110	Eugene Levner, Vladimir Kats, David Alcaide Lopez De Pablo. "Chapter ... Crossref	<1%
111	Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, Christian Muise. "An... Crossref	<1%
112	powcoder.com Internet	<1%
113	saarlandes on 2019-06-26 Submitted works	<1%
114	Antony Thomas, Fulvio Mastrogiovanni, Marco Baglietto. "MPTP: Moti... Crossref	<1%
115	Bing Li, Xingsheng Gu. "Earliness/Tardiness Scheduling Problem unde... Crossref	<1%
116	Francesco Leofante, Erika Abraham, Tim Niemueller, Gerhard Lakemey... Crossref	<1%

117 Janusz Kacprzyk, Margarita Knyazeva, Alexander Bozhenyuk. "Chapter... <1%
Crossref

118 Scheduling, 2016. <1%
Crossref