On the Complexity of Planning in Transportation and Manipulation Domains

Diplomarbeit

Malte Helmert

Albert-Ludwigs-Universität Freiburg
Fakultät für Angewandte Wissenschaften
Institut für Informatik

March 2001

Acknowledgments

I want to thank all people who (in some way or other) helped me complete this thesis. Specifically, I want to mention the members of the Durham planning group, especially Maria Fox and Derek Long, who spent a lot of time and effort to address my needs while I was in Durham and helped me feel at home in the town and especially in the department. Bernhard Nebel deserves special mention for making this thesis possible and for the e-mail support he provided while I was working in Durham. A number of other people should be mentioned here, and I hope they know that I have not forgotten them.

Erklärung (Declaration)

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

(I hereby declare that I wrote this thesis on my own, only making use of the sources mentioned.)

Durham, im März 2001 (March 2001) Malte Helmert

Contents

Chapter 1

Introduction

In this thesis, we investigate the computational complexity of some well-known benchmark problems from Artificial Intelligence planning. Specifically, we will discuss families of *transportation* and *manipulation* domains, as well as some domains that are closely related to these, but not part of either family.

Although the decision problems we study here have originally surfaced in the AI literature for the most part, this is not of critical importance to our complexity studies. Specifically, we will not discuss typical issues like the encoding of domains in a planning formalism in detail, and readers interested in route planning problems or the FreeCell solitaire game should be able to understand and benefit from the respective proofs without any previous knowledge of AI planning.

1.1 Motivation

However, the main motivation for conducting this work comes from AI planning. Many of the problems discussed in the chapters to come have in the past played an important role in the empirical evaluation of the performance of planning systems. Running time on problems from classical planning domains such as Logistics and Blocksworld has often been and still is often used for comparing the relative merits of planning systems, or, put a bit more provocatively, to draw the line between good and bad ones. However, this kind of comparison is always difficult. If no planning system performs well in a given domain, does that mean that they are all bad, or is that domain intrinsically hard? If they all perform well, is this because of their strength or because of the simplicity of the task?

Or, on a related issue, if a planning system takes significantly more time to solve a given instance of a domain than another one, but the plans it generates are shorter, which one should be preferred? Of course, this question cannot be answered in general, because sometimes high quality plans might be of critical importance whereas in other cases being able to plan (and act) quickly is to be preferred. Still, complexity results can contribute to this discussion: If in a given domain, finding optimal plans is just as hard as finding any plan, then there is no reason to be content with long plans, and if on the other hand it is a far harder problem, then this fact should be taken into account in comparing optimal and non-optimal planning systems.

As a third point, on a more fine-grained level of investigation, methods from complexity theory can not only be used to show that a given problem *is* hard, but also provide some insight as to the *source* of hardness. For instance, if we discovered that in a hypothetical Pac-Man domain, plans can be generated in polynomial time if there is one ghost while the problem of deciding plan existence is **NP**-complete in the presence of multiple ghosts, this would allow us to draw the conclusion that one source of hardness in this domain is the number of ghosts.

1.2 What Makes a Planning Domain Interesting?

It is evident that we cannot hope to analyze the complexity of every single planning domain imaginable. Even if we could, it is doubtful that this would be interesting. So we are forced to select a set of domains that we consider particularly important.

There are two criteria we use in this selection process. Firstly, rather than looking at isolated domains, we prefer analyzing families of *related* domains. In addition to allowing us to make use of a more structured approach than would be possible when looking at each domain in isolation, this makes sense for precisely the reason we have just mentioned: Boundaries between easy and complex domains in the same hierarchy tell us something about the sources of hardness for these domains.

Many well-known planning domains can naturally be associated with certain domain families. To be more specific, we will discuss families of *transportation* domains and *manipulation* domains in some detail, where for the latter family the focus will be on two subfamilies named *construction* and *transformation* domains.

This criterion alone is not sufficient, though. One can easily imagine possible domain families that are not interesting at all, for example because they are artificial and do not reflect real-world problems that people want to have solved. So the second criterion is something like *practical relevance*, which is hard to measure, or rather *popularity*, which is a bit easier to assess. We want to include domains in our analysis that the planning community is interested in, ones that are considered "standard benchmarks". More specifically, we will investigate all the domains from the AIPS planning competitions so far (from 1998 and 2000), listed in Figure 1.1. While this collection includes domains which might rightly be called atypical for planning (like FreeCell), in general being part of a competition can be considered evidence that a domain is actually interesting.

In fact, their being part of the competitions contributes to the interestingness of these domains, as, in the form of the competition results, the planning community is provided with much useful empirical evidence on the performance of many important planning systems on those domains. These results can be used to compare planning systems to one another or to compare their performance to the theoretical optimal behaviour on these domains as presented here.

This can help us identify shortcomings of current planning systems: If we can prove polynomial complexity results for a given domain, but observe poor performance in the competition results, this might tell us what kind of issues should be addressed in order to improve the overall performance of a planning system.

The trivial Movie domain will not be discussed. It is evident that Movie tasks can be solved optimally in polynomial time.

Year	Domain name	Domain family	Discussed in
1998	Assembly	Construction	Section 4.7
	Grid	Transportation	Section 3.9
	Gripper	Transportation	Section 3.7
	Logistics	Transportation	Section 3.7
	Movie	Other	—
	Mystery	Transportation	Section 3.7
	Mystery'	Transportation	Section 3.7
2000	Blocksworld	Construction	Section 4.5
	FreeCell	Construction	Section 4.6
	Logistics	Transportation	Section 3.7
	Miconic-10	Transportation	Section 3.8
	Schedule	Transformation	Section 4.4

Figure 1.1: Domains from the AIPS 1998 and AIPS 2000 planning competitions.

1.3 Related Work

Most of the work in the AI planning literature concerned with computational complexity results focuses on *domain-independent planning*, whereas we are interested in the complexity of specific domains. There is some published work on the complexity of domain-independent planning if the domain is fixed in advance (which is essentially domain-dependent planning, but for an *unspecified* domain), e. g. [Erol et al., 1995]. The domain-dependent results in that article shed light on the computational complexity of the *hardest* planning domain imaginable.

Many interesting results have been proved with regard to the complexity of different variants of the planning problem and special cases thereof, e. g. in the previously cited article and in [Bylander, 1994]. However, they focus on special cases of planning defined by purely *syntactical* features such as the number of operator preconditions or effects, while we are not concerned about the actual representation of the planning domain. The special cases of planning in transportation or manipulation domains can be seen as a restriction of the planning problem more closely related to *semantical* properties of the input.

There is very little work we are aware of in the planning literature that is heading into exactly the same direction as this thesis. The existing work concentrates on the complexity of Blocksworld, including results for the Blocksworld domain as defined here and some generalizations thereof, e. g. allowing for blocks of different size. The most comprehensive reference for this line of research is [Gupta and Nau, 1992].

Another important reference both with regard to Blocksworld and the complexity of AI planning in general is [Selman, 1994], which also emphasizes the important distinction between optimal and non-optimal planning, including very interesting results on the complexity of near-optimal planning in Blocksworld and some other domains that are not analyzed in this thesis.

Many of the results provided in this thesis focus on the domains from the competitions that formed part of AIPS 1998 and AIPS 2000. The most relevant references for these competitions, introducing the competition domains and presenting the results, are

[McDermott, 2000], [Long et al., 2000], and [Bacchus, 2001].

The usefulness of the idea of partitioning planning domains into families like "transportation" and "manipulation" is pointed out in [Long and Fox, 2000], although in this paper the focus is on the automatic *detection* of transportation domains and the exploitation of some of their features by a planning system, not on complexity aspects.

Finally, most concepts from complexity theory that are used in the following chapters and almost all decision problems used in the reductions are borrowed from the standard reference [Garey and Johnson, 1979].

1.4 Outline

In Chapter 2, we will briefly introduce some aspects of our notation that deviate from the standard. After that, we will formally define planning domains and the underlying concept of search in state spaces and introduce the decision and search problems we are interested in solving. At the end of that chapter, we will provide some very basic general results on the relationship between different variants of the planning problem.

In Chapter 3, we will discuss transportation domains, first defining a new transportation domain called TRANSPORT and some of its special cases, then discussing the relationship between some of the benchmark domains and TRANSPORT. The MICONIC-10 and GRID domains, which are more remotely related to TRANSPORT, are discussed in their own sections.

In Chapter 4, manipulation domains are discussed. Again, the benchmarks are seen as special cases of a newly defined manipulation domain called MANIPULATE. Some of the benchmarks have their own sections devoted to them because they require and deserve special treatment.

Finally, we will summarize our results and draw some conclusions in Chapter 5.

Chapter 2

Terms and Definitions

2.1 Conventions

Although we try to conform to standard mathematical notation wherever possible, there are some cases for which there is no agreed upon common notation. Therefore, we will now introduce some symbols for well-known mathematical concepts which are sometimes defined or written differently in the literature:

$\mathbb{N}_0$	The set of natural numbers (non-negative integers).
sgn	The signum function (-1 for negatives, +1 for positives, 0 for zero).
$\|A\|$	The cardinality of set A.
$\mathbb{P}(A)$	The power set of set A.
$\mathrm{Sym}(A)$	The symmetric group of a set A (the set of bijections from A to A).
A^*	The set of finite-length sequences (words) over the set A.
$[s_1, \ldots, s_n]$	The n-element sequence with i-th element s_i.
$[]$	The empty sequence.
$s +\!\!+\, t$	The concatenation of sequences s and t.
$[a : x]$	$[a] +\!\!+\, x$.
$\mathrm{order}(A)$	The set of orderings of A, i. e. sequences in A^* containing each element of A exactly once.
$\|l\|$	The length of sequence l.
$A \to B$	The set of functions from A to B. A function $f \in A \to B$ is a subset of $A \times B$ containing exactly one pair (a, b) for each $a \in A$. We will use functional notation ($f(a) = b$) and set notation ($(a, b) \in f$) interchangeably.
$A \nrightarrow B$	The set of all partial functions from A to B: $\bigcup_{A' \subseteq A}(A' \to B)$.
$\mathrm{dom}(f)$	The domain of function f: $\{ a \mid \exists b : (a, b) \in f \}$.
$\mathrm{ran}(f)$	The range of function f: $\{ b \mid \exists a : (a, b) \in f \}$.
$f(A')$	For $A' \subseteq \mathrm{dom}(f)$: $\{ f(a) \mid a \in A' \}$.
$f^{-1}(B')$	For $B' \subseteq \mathrm{ran}(f)$: $\{ a \in \mathrm{dom}(f) \mid f(a) \in B' \}$.
$f \equiv b$	f is a constant function with value b: $f(a) = b$ for all $a \in \mathrm{dom}(f)$.
$f \oplus g$	Functional overloading: $(f \setminus (\mathrm{dom}(g) \times \mathrm{ran}(f)) \cup g$.

For partial functions f, we use $f(a) = b$ as a synonym for $(a, b) \in f$. In particular, the truth value of this logical expression is well-defined even if $a \notin \mathrm{dom}(f)$. In this case, it will be false.

From [Garey and Johnson, 1979], we adopt the definition of the symbols **P**, **NP**, **PSPACE**, and **NPSPACE**, and of the terms **NP**-complete, **NP**-easy, **NP**-hard, and **NP**-equivalent. We also adopt their definition of polynomial and Turing reducibility, although we use different symbols, $\leq_p$ for polynomial reducibility and $\leq_T$ for Turing reducibility.

2.2 State Models

Our notion of a planning instance[1] relies on the concept of a state model. The definition we provide here is a slight variation of the one from [Bonet and Geffner, 2000].

Apart from some renaming and rearranging, the main difference between this definition and the one described in the reference lies in the omitted action costs, as they would all equal one for the models we are interested in here.

Definition 2.1 State model
A **state model** *is a five-tuple $M = (S, s_0, S_G, A, \delta)$, where*

- *S is the finite set of* **states***,*
- *$s_0 \in S$ is the* **initial state***,*
- *$S_G \subseteq S$ is the set of* **goal states***,*
- *A is the finite set of* **actions** *and finally*
- *$\delta : S \times A \nrightarrow S$ is the (partial)* **transition function***.*

The set of **applicable actions** *of a state $s \in S$ is defined as $\{\ a \in A \mid (s, a) \in \mathrm{dom}(\delta)\ \}$.*

We will use the symbol $\mathcal{M}$ to denote the set of all state models.

The intuition behind this definition should be fairly obvious to anyone acquainted with the concept of state spaces. It is very similar to the common definition of deterministic finite automata (DFAs) except for the fact that for these, the transition function is usually defined to be total.

Definition 2.2 Augmented transition function
Let $M = (S, s_0, S_G, A, \delta)$ be a state model.

The **augmented transition function** *$\hat{\delta} : S \times A^* \nrightarrow S$ is defined as the partial function with minimal domain satisfying:*

$$\begin{aligned} \hat{\delta}(s, []) &= s \\ \hat{\delta}(s, [a : x]) &= \hat{\delta}(\delta(s, a), x) \text{ if } (s, a) \in \mathrm{dom}(\delta) \wedge (\delta(s, a), x) \in \mathrm{dom}(\hat{\delta}) \end{aligned}$$

This definition captures the important notion of applying a sequence of actions to some state.

[1]We speak of planning *instances* rather than planning *problems* to avoid confusion with the concepts of *decision problems* and *search problems* from complexity theory.

2.3 Plans and Planning Domains

Definition 2.3 Plan
Let $M = (S, s_0, S_G, A, \delta)$ be a state model.
$l \in A^*$ is called a **plan** *for M if and only if $(s_0, l) \in \mathrm{dom}(\hat{\delta}) \wedge \hat{\delta}(s_0, l) \in S_G$.*
It is called **optimal** *if its length is minimal amongst all plans.*
M is called **solvable** *if there exists a plan for it.*

In our terminology, a *plan* (or *sequential plan*) is what is often called a *solution* in the literature. A sequence of actions that can be applied to the initial state but does not lead to a goal state will simply be called an *applicable action sequence.*

From a mathematical point of view, we have now defined all there is to planning instances: their properties are defined by their state models, and the solutions we are interested in are (possibly optimal) plans.

However, from a complexity theory point of view, one important aspect has not been mentioned yet, and that is the *description length* of a planning instance. In general, this length does not correspond in any obvious way to the state model of the instance as presented before. Thus, our definition of a planning domain will include an explicit notion of instance encodings, and indeed it is this encoding rather than the mathematical model that we will call a *planning instance.*

Definition 2.4 Planning domain
A **planning domain** *is a function $\mathcal{D} : L \to \mathcal{M}$, where $L \subseteq \Sigma^*$ for some alphabet Σ.*
L is called the **encoding language** *of the domain.*[2]
$w \in L$ is called a **planning instance** *and $|w|$ is called its* **encoding length**.

In practice, planning instances are usually encoded in PDDL ([McDermott, 2000]), often restricted to the subset of that language identified as "basic STRIPS[3]" or the more general "ADL" subset.

Since we are interested in proving complexity results with respect to this encoding, we will assume that instances are given in PDDL when talking about their encoding length. However, we see no point in actually *using* PDDL ourselves when defining planning domains, as this does not facilitate understanding the definitions. The state models we will use for the specific planning instances will not be based on the usual logical representations either, as this would lead to an increased complexity of proofs for purely technical reasons.

Instead, we will describe planning instances using more "natural" parameters, not explicitly defining L, and *assume* a sensible encoding length without providing the actual encoding. Because we want our proofs to apply to PDDL encodings, the assumed encoding lengths will always be polynomially equivalent to the length of the corresponding PDDL encoding (in characters). An example of a suitable assumed encoding length is the number of PDDL objects in the instance.

[2]In the chapters to come, it is understood that deciding whether a given word is a valid encoding or not can always be done in polynomial time, which is important for complexity theory proofs. It should be evident that the encoding languages we are going to use satisfy this requirement and therefore it will not be explicitly mentioned.

[3]This should not be confused with the closely related, but more expressive language used by the planning system of the same name, cf. [Fikes and Nilsson, 1971] and [Lifschitz, 1987], or [Allen et al., 1990] for a reprint of both references.

The reader should have no trouble in verifying that our definitions of the planning domains in the following sections are indeed semantically equivalent to their corresponding PDDL descriptions (again, we refer to [McDermott, 2000] and [Bacchus, 2001]). That said, in a few cases we *will* indeed use models that are semantically different. We will only do this where we think that there are errors in the PDDL descriptions, and explicitly point out the differences and justify the decisions we made.

One final remark: From a complexity point of view, PDDL encodings do not always satisfy the requirement of being *reasonably concise* [Garey and Johnson, 1979]. For example, instances of the GRIPPER domain could be specified by encoding an integer in binary, denoting the number of balls to be moved. Under this encoding, plan length would be exponential in the length of the input and thus plan generation would require exponential time. However, using the PDDL description, plan generation can easily be done in polynomial time in the length of the input. This is due to the fact that the common PDDL encoding of numbers uses an amount of space linear (rather than logarithmic) in their magnitude. In cases where it is not evident, we will point out whether or not the PDDL encoding is reasonably concise.

2.4 Variants of the Planning Problem

What is the complexity of a planning domain? There are several answers to that question. In some domains, it might be easy to find a plan, but hard to generate an optimal one. In other domains, it might be easy to prove that a plan exists but hard to generate it, for example because it is very long. Specifically, we will focus on the following problems:

Definition 2.5 The PLANEX-DOMAIN decision problem
Given a fixed planning domain (which will be mentioned as part of the decision problem, as in PLANEX-MYSTERY*),* PLANEX-DOMAIN *is the language of all solvable instances.*

Definition 2.6 The PLANGEN-DOMAIN search problem
Given a planning instance in a fixed planning domain, output a pair (L, P), where P is a sequential plan of length $L \in \mathbb{N}_0$ solving the instance, or return "no" if no plan exists.[4]

Definition 2.7 The PLANLEN-DOMAIN decision problem
Given a fixed planning domain, PLANLEN-DOMAIN *is the language of all pairs (I, L), where I is a planning instance in that domain, and $L \in \mathbb{N}_0$ such that there exists a sequential plan for I with length at most L.*

Definition 2.8 The PLANOPT-DOMAIN search problem
Given a planning instance in a fixed planning domain, output a pair (L, P), where P is an optimal sequential plan of length $L \in \mathbb{N}_0$ solving the instance, or return "no" if no plan exists.

Note that (once a domain has been fixed), the first and third of these problems are decision problems, while the other two are search problems. Although in practice we are primarily interested in solving the search problems, we will only prove results for

[4]The length of the plan is prefixed to the actual plan in this and the other search problem for technical reasons, cf. Lemma 2.9.

the decision problems. The implications of these results for the complexity of the search problems are investigated in the rest of this section.

Both decision problems are closely related to the respective search problems introduced after them. This is no coincidence. While it is generally harder to theoretically analyze search problems, complexity theory allows us to prove results on decision problems and then draw conclusions with regard to the complexity of the related search problems. We will often do this in the chapters to come, mainly relying on the following lemmas. All of these assume a fixed domain named Domain.

Lemma 2.9 Planning problems relationship
Unconditionally, it is true that:
PlanEx-Domain $\leq_T$ PlanGen-Domain
PlanLen-Domain $\leq_T$ PlanOpt-Domain
PlanGen-Domain $\leq_T$ PlanOpt-Domain

If there exists a polynomial p such that for all planning instances of encoding length n, a state can be encoded in binary using no more than $p(n)$ units of space[5]*, then:*
PlanEx-Domain $\leq_p$ PlanLen-Domain
PlanEx-Domain $\leq_T$ PlanOpt-Domain

Proof: A planning instance I is in PlanEx-Domain if and only if PlanGen-Domain does not return "no" on I, proving the first result. Note that even though consulting the PlanGen-Domain oracle can cause an exponentially long string to be written on the oracle tape, this is all done in one computation step (cf. [Garey and Johnson, 1979]), and it is only necessary to look at a constant-length prefix of the output to decide whether it is equal to "no" or not.

Similarly, a pair (I, L) is in PlanLen-Domain if and only if PlanOpt-Domain returns a plan of length at most L (and not "no"), proving the second claim. Again, despite the possibly exponentially long solution to PlanOpt-Domain, this can be done in polynomial time using an oracle, as only the first part of the PlanOpt-Domain output, denoting the length of the generated plan, needs to be examined.[6]

Algorithms for generating optimal sequential plans can be seen as a special case of algorithms producing *any* plan, giving the third reduction.

If states can be encoded in binary using space bounded by $p(n)$, then there are no more than $2^{p(n)}$ states, and whenever a plan exists, there is a plan of length at most $2^{p(n)} - 1$, as it is not necessary to traverse the same state twice, and thus $I \in$ PlanEx-Domain if and only if $(I, 2^{p(n)} - 1) \in$ PlanLen-Domain. This is a polynomial reduction, as only space $p(n)$ is needed for encoding this number.

The last result is a consequence of the fourth and second. ∎

The main message of this lemma is that the search problems are at least as hard as their corresponding decision problems. The following lemma states that, under some restricting assumptions, they are in a way "no harder".

[5]This will be true for all the domains we will discuss, without us mentioning it explicitly.

[6]This is the reason for the somewhat artificial definition of the output of PlanOpt-Domain. While the same problem does *not* arise for PlanGen-Domain, we chose to define it similarly for symmetry reasons.

Lemma 2.10 Plans of polynomial length
Let p be a polynomial and Domain *be a planning domain such that each state can be represented using space bounded by $p(n)$, there are at most $p(n)$ actions, the result of applying the transition function to a given state and action can be computed in time bounded by $p(n)$, and membership of a state in the set of goal states can be tested in time bounded by $p(n)$, where n is the encoding length of the instance.*[7]

If for each solvable planning instance of this domain with encoding length n, there exists a plan of length at most $p(n)$, then:
PlanEx-Domain $\in$ **NP**.
PlanLen-Domain $\in$ **NP**.
PlanGen-Domain *and* PlanOpt-Domain *are* **NP**-*easy.*

Proof: PlanEx-Domain and PlanLen-Domain can be solved by guessing an optimal sequential plan and verifying that it is indeed a plan. This involves polynomial time for writing down the initial state, polynomial time for applying each of the (polynomially many) actions, in sequence, to the initial state, and polynomial time to check that the resulting state is a goal state. This proves the first two results.

Note that this is still true if the initial state is given as (the third) part of the problem instance.[8] We show that PlanOpt-Domain can be Turing-reduced to that generalized version of PlanLen-Domain, thus showing **NP**-easiness for this search problem (and PlanGen-Domain, applying the previous lemma). The reduction is described as follows:

Let I be a planning instance from Domain with initial state s_0.

If $(I, p(n), s_0) \notin$ PlanLen-Domain, return "no".

Otherwise, let $len := \min\{\ n \in \mathbb{N}_0 \mid (I, n, s_0) \in$ PlanLen-Domain $\}$. This can be calculated using $\Theta(log_2(p(n)))$ invocations of PlanLen-Domain with a binary search strategy.

The *current state* s is initialized as s_0, the *remaining plan length* l as len. As long as $l > 0$, a check is made for each action $a \in A$ that is applicable in state s to see if $(I, l-1, \delta(s,a)) \in$ PlanLen-Domain. There must be such an action, so any action a satisfying this can be chosen and added to the (initially empty) action sequence that is to be returned. Then s is updated to $\delta(s,a)$ and l is decreased by one. This is repeated until l becomes zero, at which stage a goal state has been reached and the calculated action sequence is returned.

The remaining plan length l will be decreased at most a polynomial number of times. Each time this is done, a polynomial number of checks for applicability, state transitions, and invocations of PlanLen-Domain are performed, so this is a polynomial Turing transformation. ■

Domains that satisfy the important requirement of the existence of "short" plans in the previous lemma will be referred to as *domains with polynomial length plans* in the following.

[7]These properties are trivial to verify in all the planning domains in this thesis and we will not discuss them explicitly when applying this lemma.

[8]With our definition of a planning domain, the initial state is a function of the planning instance which will usually not allow for arbitrary initial states. For example, in the Transport domain from Section 3.1 all mobiles must start out unloaded.

We now put the two previous results together:

Lemma 2.11 NP-complete decision problems
If DOMAIN *satisfies the requirements stated in the previous lemma (particularly, if it is a domain with polynomial length plans), the following is true:*
If PLANEX-DOMAIN *is* **NP**-*complete, then so is* PLANLEN-DOMAIN.
If PLANEX-DOMAIN *is* **NP**-*complete, then* PLANGEN-DOMAIN *is* **NP**-*equivalent.*
If PLANEX-DOMAIN *is* **NP**-*complete, then* PLANOPT-DOMAIN *is* **NP**-*equivalent.*
If PLANLEN-DOMAIN *is* **NP**-*complete, then* PLANOPT-DOMAIN *is* **NP**-*equivalent.*
Proof: This follows directly from the previous two lemmas. ∎

As a consequence of this lemma, we will not discuss the search problems further in the chapters to come, although these are the ones we are primarily interested in. As **NP**-completeness results carry over, we can focus our attention on decision problems.

An "easiness" result on the other hand, e. g. proving that some decision problem can be solved in polynomial time, does not necessarily imply that the same is true for the corresponding search problem. However, whenever showing such a result, we will do so by giving a constructive proof which can easily be turned into a (polynomial time) algorithm for generating plans.

For domains where plan lengths are not polynomial, however, it is quite possible that there is a difference in complexity between the decision and search problems. Although deciding the existence of a plan might be easy, plans can never be generated in polynomial time in such domains because of the length of the output. Indeed, if shortest plan lengths can be exponential in the input size, no planning system can ever generate a plan in less than exponential time, and thus we will in those cases be content with stating the fact that plans are exponentially long and not investigate the decision problems in depth, as further results would be of no consequence to planning algorithms.

Chapter 3

Transportation Domains

We start by analyzing transportation domains because this type of domain seems to be most wide-spread. This shows in the fact that many important benchmark domains fall within the transportation family, including the benchmark domain that is probably best-known, LOGISTICS, but domains like GRIPPER, MYSTERY, MYSTERY', MICONIC-10 and GRID are also easily identified as having a transportation theme.

While the usual PDDL transportation domains do have their shortcomings, e. g. not being able to take distances between locations into account, they are still close enough to real-world concepts to be relevant.

By a *transportation domain*, we understand a planning domain exhibiting the following features in its instances:

- *Locations* are connected by *roads*, building a graph called a *roadmap*.
- *Mobiles* are moving around this map.
- *Portables* are being carried by the mobiles.
- The goal is to move (a subset of) the portables to their desired *final destinations*.

In our definition, we require the sets of locations, mobiles and portables to be disjoint. Specifically, mobiles cannot be carried by other mobiles.

There are some other characteristics that a given transportation domain may or may not exhibit, namely:

- *Dynamic roadmap*: The roadmap can be changed by the application of actions establishing or removing links between locations, e. g. doors being opened or closed.
- *Capacity constraints*: Mobiles are restricted in the number of portables they can carry at the same time.
- *Fuel constraints*: Mobiles are restricted in the number of movement actions they may perform.
- *Movement constraints*: A given mobile is only allowed to traverse a subgraph of the overall roadmap, to model ships not being allowed to leave the water, or trains not being allowed to leave the railway tracks. More precisely, the roadmap that each mobile may traverse is a subgraph of the general roadmap.

We will not analyze dynamic roadmaps in general, as this concept may appear in a domain in very different ways. However, we will discuss two types of dynamic roadmaps in the context of the MICONIC-10 and GRID domains.

The other three characteristics, however, can be treated in a fairly general way, so we will consider them in our general analysis.

3.1 TRANSPORT

We will first introduce a transportation domain called TRANSPORT which exhibits capacity, fuel and movement constraints and investigate some special cases of this domain. These results can then be applied to the benchmark domains we are mainly interested in.

While the next two definitions might seem a bit lengthy, we think that in complexity theory, it is vital to define the problem at hand in a formal and unambiguous way. The reasoning in the proofs will take place at a somewhat higher level of abstraction, but once the formal definitions have been presented, they can be used to verify any intuitive claim that seems doubtful. This would be impossible if the semantics of the planning domain were not properly defined.

Definition 3.1 TRANSPORT **instance**
A TRANSPORT **instance** *is a 10-tuple*
$I = (V, E, M, P, P_G, loc_0, loc_G, cap, fuel, road)$, *where*

- (V, E) *is a graph with a finite number of vertices called* **locations***; the edges are called* **roads***,*
- M *is a finite set of* **mobiles***,*
- P *is a finite set of* **portables***,*
- $P_G \subseteq P$ *is the set of* **goal portables***,*
- $loc_0 : (M \cup P) \to V$ *is the* **initial location** *function,*
- $loc_G : P_G \to V$ *is the* **goal location** *function,*
- $cap : M \to \mathbb{N}_0$ *is the* **capacity** *function,*
- $fuel : V \to \mathbb{N}_0$ *is the* **fuel** *function and finally*
- $road : M \to \mathbb{P}(E)$ *is the* **movement constraints** *function.*

We require V, M and P to be disjoint sets. The encoding length of the instance is assumed to be $|V| + |M| + |P|$.

Fuel is associated with locations rather than mobiles because this is the way it is handled in the MYSTERY-like domains. Both definitions would make sense.

The encoding length as specified by us does not need to take into account E, as its size is polynomial in $|V|$, similarly for P_G with respect to $|P|$, and for loc_0 and loc_G with respect to $(|M| + |P|) \cdot |V|$, which is polynomial in $|V| + |M| + |P|$. The encoding of *road* will be polynomial in $|M| \cdot |E|$ and thus polynomial in $|V| + |M|$. Although *cap* could theoretically be unbounded, we can safely assume it to be bounded by $|P|$, which means it can be encoded in space polynomial in $|M| \cdot |P|$ or $|M| + |P|$.

While there *are* sequences of actions that would require an arbitrary amount of fuel, any reasonable plan only includes a number of movements that is bounded by a polynomial in $|V| + |M| + |P|$, and so we can safely assume the fuel function to be bounded by this polynomial, too, without loss of generality. The proof of this claim will follow shortly, with Theorem 3.3.

Thus, $|V|+|M|+|P|$ is polynomially equivalent to the length of a "reasonable" PDDL encoding of the planning instance. Note that we are taking into account that numbers are represented in PDDL using a *unary* encoding.

Definition 3.2 Transport **domain**
Let L be the set of all Transport *instances. The* Transport **domain** *is the function $\mathcal{D} : L \to \mathcal{M}$ mapping instances to state models as follows:*

$$
\begin{aligned}
&(V, E, M, P, P_G, loc_0, loc_G, cap, fuel, road) \mapsto (S, s_0, S_G, A, \delta), \text{ where}\\
&S \overset{\text{def}}{=} (M \to V) \times (P \to V \cup M) \times (V \to \{0, \ldots, \max\{\ fuel(m) \mid m \in M\ \}\}),\\
&s_0 \overset{\text{def}}{=} (loc_0 \cap (M \times V), loc_0 \cap (P \times V), fuel),\\
&S_G \overset{\text{def}}{=} \{\ (l_M, l_P, f) \in S \mid loc_G \subseteq l_P\ \},\\
&A \overset{\text{def}}{=} \{\ move_{m,v} \mid m \in M, v \in V\ \}\\
&\quad \cup \{\ pick_{m,p} \mid m \in M, p \in P\ \}\\
&\quad \cup \{\ drop_{m,p} \mid m \in M, p \in P\ \},\\
&\forall m \in M, v, v' \in V, s = (l_M, l_P, f) \in S :\\
&\qquad l_M(m) = v \wedge \{v, v'\} \in road(m) \wedge f(v) > 0\\
&\quad \Rightarrow \delta(s, move_{m,v'}) \overset{\text{def}}{=} (l_M \oplus \{(m, v')\}, l_P, f \oplus \{(v, f(v) - 1)\}),\\
&\forall m \in M, p \in P, s = (l_M, l_P, f) \in S :\\
&\qquad l_M(m) = l_P(p) \wedge |\{\ p \in P \mid l_P(p) = m\ \}| < cap(m)\\
&\quad \Rightarrow \delta(s, pick_{m,p}) \overset{\text{def}}{=} (l_M, l_P \oplus \{(p, m)\}, f),\\
&\forall m \in M, p \in P, s = (l_M, l_P, f) \in S :\\
&\qquad l_P(p) = m\\
&\quad \Rightarrow \delta(s, drop_{m,p}) \overset{\text{def}}{=} (l_M, l_P \oplus \{(p, l_M(m))\}, f) \text{ and finally}\\
&\delta \text{ is undefined otherwise.}
\end{aligned}
$$

This definition captures the intuition of a Transport instance. Observe that states are three-tuples characterizing the current location of the mobiles (which traverse the locations), the location of the portables (being at locations or within mobiles) and the amount of fuel available at the various locations.

The *move* operator is parameterized by the moving mobile and its destination location, the *pick* and *drop* operators by the mobile and portable involved.

As captured by the definition of δ, a mobile may only move to a location if this is allowed by its roadmap and if there is some fuel available at its current destination. Movement changes the location of the mobile and deducts fuel. Portables may only be picked up by mobiles which share their location and have not reached their full carrying capacity. A mobile may only drop a portable it is currently carrying. Pickup and drop actions change the location of the portable involved accordingly.

Now that we have defined the semantics of the domain and hopefully provided the reader with an intuition about the operators, we can present the first result.

Theorem 3.3 PlanLen-Transport $\in$ **NP**
PlanEx-Transport *and* PlanLen-Transport *are in* **NP***.*
Proof: We will prove that Transport is a domain with polynomial length plans and apply Lemma 2.10.

Let V be the set of locations, M be the set of mobiles and P be the set of portables of a given Transport instance.

For each portable, we can safely bound the number of *move* actions performed by any mobile while carrying that portable by $|V|-1$, as in any plan where the portable is moved to the same location twice, it could have been dropped and left there at the first visit. Thus, the total number of movements performed by non-empty mobiles is bounded by $|P| \cdot (|V|-1)$. The same bound applies to the number of *pick* and *drop* actions, because in between two movements of a portable, it is dropped and picked up at most once in any reasonable plan. Additionally, it is picked up once before being moved for the first time, and it is dropped once after being moved for the last time. We call the actions we have considered so far "portable-related" actions, and as has been shown their number can be bounded by $3|P| \cdot (|V|-1)$.

The only actions which we still need to take into account are movements by empty mobiles. To bound these, note that between any two portable-related actions and before the first portable-related action, a given mobile should not commit more than $|V|-1$ movements, as it does not make sense for a mobile to visit a given location twice if no portable-related actions have been applied in between. No further actions are necessary after the last portable-related action (which drops the last portable at its final destination). Thus, the number of movements by empty mobiles can be bounded by $|M| \cdot (|V|-1) \cdot (3|P| \cdot (|V|-1))$.

Summing up the two terms, we get an upper bound of $3|P| \cdot (|V|-1)(1+M \cdot (|V|-1))$, which is polynomial in the encoding length $|V|+|M|+|P|$. ■

In addition to its other implications, the proof of Theorem 3.3 shows that the amount of fuel at any location can safely be bounded by a polynomial in $|V|+|M|+|P|$, because this is true for the number of movement actions in any reasonable plan. If there is "enough" fuel at a given location v, we will in the following write this as $f(v) = \infty$.

3.2 Special Cases of Transport

Our next objective is to define some special cases of the Transport domain that are of particular interest. We will introduce a total of 27 variants, with the original domain being a generalization of all the others. They will be assigned the name Transport-$C_iF_jM_k$ (capacity/fuel/mobiles), where $i, j \in \{1, \infty, *\}$ and $k \in \{1, +, *\}$.

Definition 3.4 Special cases of Transport
For $i, j \in \{1, \infty, *\}$ *and* $k \in \{1, +, *\}$*, the* Transport-$C_iF_jM_k$ *domain is the restriction of* Transport *to those instances* $I = (V, E, M, P, P_G, loc_0, loc_G, cap, fuel, road)$ *that adhere to the following constraints:*

- *For $i = 1$, $cap \equiv 1$. Each mobile can only carry one portable at a given time in these instances.*
- *For $i = \infty$, $cap \equiv |P|$. Each mobile can carry any number of portables in these instances.*
- *For $i = *$, there are no restrictions on the carrying capacity function.*
- *For $j = 1$, $fuel \equiv 1$. There is exactly one unit of fuel at each location in these instances.*
- *For $j = \infty$, $fuel \equiv \infty$. There are no fuel constraints in these instances.*
- *For $j = *$, there are no restrictions on the fuel function.*
- *For $k = 1$, $|M| = 1$ and $road \equiv E$. In these instances there is only one mobile, which can traverse the entire roadmap.*
- *For $k = +$, $road \equiv E$. In these instance all mobiles traverse the whole map, there is only one type of mobiles.*
- *For $k = *$, there are no restrictions on the number or type of mobiles.*

According to that definition, Transport-$C_*F_*M_*$ is just another name for Transport.

capacity	**1**: one portable	∞: unbounded	*: varies
fuel	**1**: one unit per location	∞: unbounded	*: varies
mobiles	**1**: one mobile	$+$: one mobile type	*: many mobile types

Figure 3.1: The Transport domain family.

Figure 3.1 summarizes these definitions. Note that C_* domains generalize the corresponding C_1 and C_∞ domains, F_* domains generalize the corresponding F_1 and F_∞ domains, M_* domains generalize the corresponding M_+ and M_1 domains and finally M_+ domains generalize the corresponding M_1 domains.

Apart from these, there are no other subsumption relationships in this framework. Specifically, there are *four* most basic domains, ones that do not have any specialization, namely Transport-$C_1F_1M_1$, Transport-$C_\infty F_1M_1$, Transport-$C_1F_\infty M_1$ and Transport-$C_\infty F_\infty M_1$.

We will now prove a number of results for some of these domains. Exploiting their specialization/generalization relationship, we will then be able to say for each of those domains and both decision problems we are interested in whether they are polynomial or **NP**-complete. From Theorem 3.3 we already know that they are all members of **NP**.

3.3 PlanEx for the Transport Family

We will start by investigating the PlanEx problem.

Theorem 3.5 PlanEx-Transport-$C_* F_\infty M_* \in$ **P**
PlanEx-Transport-$C_* F_\infty M_*$ *can be solved in polynomial time.*
Proof: For each mobile m with $cap(m) > 0$, we calculate the set of vertices this mobile can move to by doing a breadth-first search on $(V, road(m))$, starting at $l_0(m)$. We then remove all edges from $road(m)$ that can never be reached by m.

These reduced roadmaps of the individual mobiles are then united to form a graph we call the *reachability graph RG*. Edges of the reachability graph are labeled with the mobiles which contribute to them.

Clearly, for any two locations $v \neq v'$, it is possible to deliver a portable from v to v' if and only if there is a path between those two locations in the reachability graph. Thus, a plan exists if and only if for all portables $p \in P_G$ with $loc_0(p) \neq loc_G(p)$, these two locations belong to the same connected component in RG.

We now give an algorithm that generates a plan. The "current location" of a mobile m is initialized to $loc_0(m)$.

We then perform the following steps for each portable p to be moved:

1. Calculate a shortest path from $loc_0(p)$ to $loc_G(p)$ in RG, consisting of vertices $[v_0, v_1, \ldots, v_k]$.
2. For $i \in \{0, \ldots, k-1\}$, do the following:
   (a) Look at the label of the edge $\{v_i, v_{i+1}\}$ to find a mobile m that can traverse it.
   (b) Calculate a shortest path in $road(m)$ from the current location of m to v_i.
   (c) Add the corresponding *move* actions to the plan.
   (d) Add the actions $pick_{m,p}$, $move_{m,v_{i+1}}$ and $drop_{m,p}$.
   (e) Update the current location of m to be v_{i+1}.

It is evident that this action sequence constitutes a plan. We conclude the proof by mentioning that all the calculations mentioned can clearly be done in polynomial time using standard algorithms. ■

So we have shown that we can handle two of the three constraints we introduced, capacity and movement, even in their most general case, as long as fuel is unlimited. Solving these instances is easy because each portable can be looked at separately.

However, this is not true if fuel is limited, and indeed, limited fuel domains are much harder to solve. We will prove this by showing that Transport-$C_\infty F_1 M_1$ and Transport-$C_1 F_1 M_1$ are both **NP**-complete. As all domains involving limited fuel are generalizations of one of these, this completes the picture for PlanEx-Transport.

We will perform this proof in three steps. First, we will use a polynomial transformation of a known **NP**-complete problem called Hamiltonian Circuit - Planar - Cubic to show **NP**-completeness of a problem called Hamiltonian Path - Fixed Start - Planar. Then, we will polynomially transform this problem to Transport-$C_\infty F_1 M_1$. Because we will need this property later, and because it is interesting in its own right, we

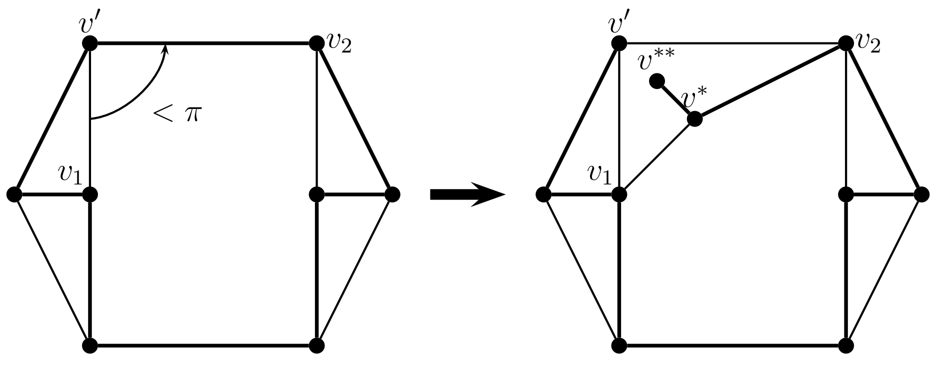


Figure 3.2: Adding v^* and v^{**}. The highlighted edges mark corresponding solutions to the Hamiltonian Circuit/Path problems.

will actually prove that TRANSPORT-$C_\infty F_1 M_1$ is already **NP**-complete if the roadmap is restricted to be a planar graph.

Unfortunately, a similar transformation to TRANSPORT-$C_1 F_1 M_1$ destroys planarity. Thus, we will in the third and last step of our proof use the more general HAMILTONIAN PATH - FIXED START problem as the basis of our transformation.

Definition 3.6 HAMILTONIAN CIRCUIT - PLANAR - CUBIC
Given a planar, cubic[1] *graph $G = (V, E)$, does G contain a Hamiltonian circuit, i. e. a sequence $[v_1, \ldots, v_{|V|}] \in \text{order}(V)$ such that for all $i \in \{1, \ldots, |V|-1\}$, $\{v_i, v_{i+1}\} \in E$ and $(v_{|V|}, v_1) \in E$?*

This problem is cited as being **NP**-complete in [Garey and Johnson, 1979].

Definition 3.7 HAMILTONIAN PATH - FIXED START - PLANAR
Given a planar graph $G = (V, E)$ and vertex $v_1 \in V$, does there exist a Hamiltonian path in V that starts at v_1, i. e. a sequence $[v_1, \ldots, v_{|V|}] \in \text{order}(V)$ such that for all $i \in \{1, \ldots, |V|-1\}$, $\{v_i, v_{i+1}\} \in E$?

Lemma 3.8 NP-completeness of HAMILTONIAN PATH - FIXED START - PLANAR
HAMILTONIAN PATH - FIXED START - PLANAR *is* **NP**-*complete.*
Proof: Membership in **NP** is obvious.

For **NP**-hardness, we provide a polynomial transformation to prove HAMILTONIAN CIRCUIT - PLANAR - CUBIC $\leq_p$ HAMILTONIAN PATH - FIXED START - PLANAR.

Let $G = (V, E)$ be the original instance and $n = |V|$. We omit the trivial case $n = 0$ and assume that $|V| \geq 4$ (a non-empty cubic graph can obviously never have less than four vertices). First, we calculate a planar embedding of G.

Let v' be any vertex from V. Because G is cubic, v' has exactly three neighbours. Let v_1, v_2 be two of those neighbours satisfying $\angle v_1 v' v_2 < \pi$. There must be a pair of neighbouring vertices satisfying this condition since the three angles corresponding to the three possible choices of v_1, v_2 add up to 2π.

It is then possible to add vertices v^*, v^{**} and edges $\{v_1, v^*\}$, $\{v_2, v^*\}$, $\{v^*, v^{**}\}$ to the graph without destroying its planarity (as illustrated in Figure 3.2). The graph that G is mapped to is then defined as:

[1] That is, every vertex has degree 3.

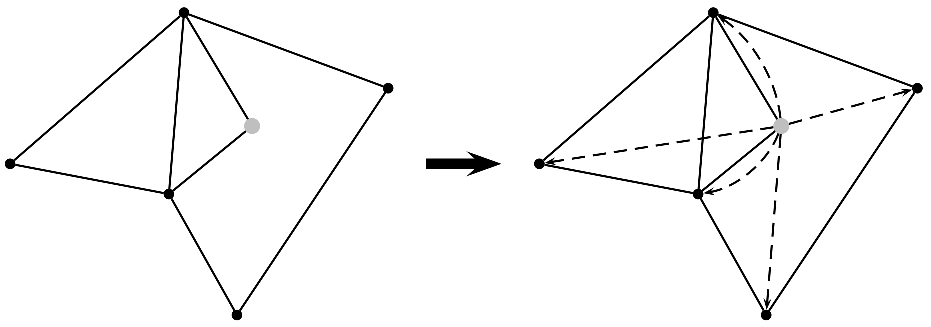


Figure 3.3: Graph and corresponding TRANSPORT-$C_\infty F_1 M_1$ instance. Portables are displayed as dotted lines pointing from their initial to goal locations. The designated start vertex and the initial location of the mobile are drawn as bigger shaded circles.

$G' \overset{\text{def}}{=} (V \cup \{v^*, v^{**}\}, E \cup \{\{v_1, v^*\}, \{v_2, v^*\}, \{v^*, v^{**}\}\})$.

Clearly all of this can be done in polynomial time.

Assume that G contains a Hamiltonian circuit $[u_1, \dots, u_n]$. As it must contain two of the three edges adjacent to v', it must contain at least one of $\{v', v_1\}$ or $\{v', v_2\}$, say the former. Because sequences denoting Hamiltonian circuits can be inverted and rotated, we can assume that $u_1 = v'$ and $u_n = v_1$. Then $[v', u_2, u_3, \dots, u_{n-1}, v_1, v^*, v^{**}]$ is a Hamiltonian path in G' starting at v'.

Now assume that G' contains a Hamiltonian path starting at v'. As v^{**} has only one neighbour, it must be the last node on the path, preceded by v^* in turn preceded by either v_1 or v_2, say the former. Thus, the Hamiltonian path can be written as $[v', u_2, \dots, u_{n-1}, v_1, v^*, v^{**}]$ and $[v', u_2, \dots, u_{n-1}, v_1]$ is a Hamiltonian circuit in G.

This shows that the mapping described above is indeed a polynomial transformation and concludes the proof. ∎

Theorem 3.9 NP-completeness of PLANEX-TRANSPORT-$C_\infty F_1 M_1$
PLANEX-TRANSPORT-$C_\infty F_1 M_1$ *is* **NP**-*complete, even if the roadmap is restricted to be a planar graph and all portables start at the same location as the mobile.*

Proof: Membership in **NP** has already been proved (special case of TRANSPORT).

In order to prove **NP**-hardness, we describe a polynomial transformation that shows HAMILTONIAN PATH - FIXED START - PLANAR $\leq_p$ PLANEX-TRANSPORT-$C_\infty F_1 M_1$. The mapping is defined as follows:

$((V, E), v_1) \mapsto I = (V', E', M, P, P_G, loc_0, loc_G, cap, fuel, road)$ where:

$V' \overset{\text{def}}{=} V$	$loc_0 \equiv v_1$
$E' \overset{\text{def}}{=} E$	$loc_G(p_v) \overset{\text{def}}{=} v$ for $v \in V$
$M \overset{\text{def}}{=} \{m\}$	$cap \equiv \lvert P \rvert$
$P \overset{\text{def}}{=} \{\, p_v \mid v \in V \,\}$	$fuel \equiv 1$
$P_G \overset{\text{def}}{=} P$	$road \equiv E'$

This mapping satisfies all the requirements for a TRANSPORT-$C_\infty F_1 M_1$ instance and can clearly be calculated in polynomial time. Figure 3.3 gives an example of this mapping.

Let $n = |V|$. Assume that (V, E) contains a Hamiltonian path $[v_1, \dots, v_n]$ (starting at v_1). Then $[pick_{m,p_{v_2}}, \dots, pick_{m,p_{v_n}}, move_{m,v_2}, drop_{m,v_2}, \dots, move_{m,v_n}, drop_{m,v_n}]$ is a plan for I.

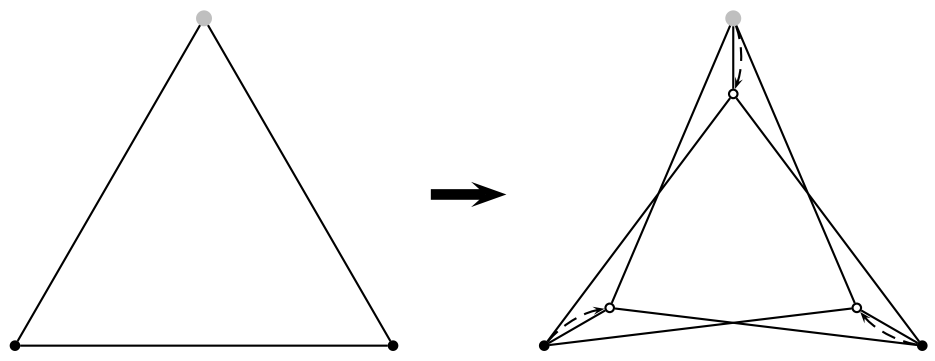


Figure 3.4: Graph and corresponding Transport-$C_1F_1M_1$ instance. Portables are displayed as dotted lines pointing from their initial to goal locations. The designated start vertex and the initial location of the mobile are drawn as bigger shaded circles. Entrances are solid circles, exits hollow circles.

Now assume that there is a plan for I. Due to the fuel constraints, each location can be left by the only mobile at most once. This means that the plan must start with a series of pickup actions — it is not possible to return to the initial location to pick up leftover packages and deliver them later.

We can assume that the portable to be carried to a given location is dropped when that location is visited for the first time, and that there are no further actions after all goals have been satisfied. This means that the location where the last portable is dropped is never left, and thus there can be no more than $n-1$ *move* actions in the plan. As every location must be visited at least once, there cannot be *less* than $n-1$ *move* actions.

So there are exactly $n-1$ *move* actions in that plan, visiting every location *exactly* once. This means that the movement path of the mobile is a Hamiltonian path for (V, E), starting at the initial location of the mobile, v_1. ■

Theorem 3.10 NP-completeness of PlanEx-Transport-$C_1F_1M_1$
PlanEx-Transport-$C_1F_1M_1$ *is* **NP**-*complete, even if all portables only need to be moved to adjacent locations.*
Proof: Membership in **NP** has already been proved (special case of Transport).

We describe an algorithm that maps Hamiltonian Path - Fixed Start instances to membership-equivalent Transport-$C_1F_1M_1$ instances in polynomial time, thus proving Hamiltonian Path - Fixed Start $\leq_p$ PlanEx-Transport-$C_1F_1M_1$. The mapping is defined as follows:

$((V,E),v_1) \mapsto I = (V', E', M, P, P_G, loc_0, loc_G, cap, fuel, road)$ where:

$$
\begin{array}{ll}
V' \overset{\text{def}}{=} \bigcup_{v\in V}\{v, v^*\} & loc_0(p_v) \overset{\text{def}}{=} v \text{ for } v \in V \\
E' \overset{\text{def}}{=} \{\ \{u, v^*\} \mid \{u,v\} \in E\ \} \cup \{\ \{v, v^*\} \mid v \in V\ \} & loc_G(p_v) \overset{\text{def}}{=} v^* \text{ for } v \in V \\
M \overset{\text{def}}{=} \{m\} & cap \equiv 1 \\
P \overset{\text{def}}{=} \{\ p_v \mid v \in V\ \} & fuel \equiv 1 \\
P_G \overset{\text{def}}{=} P & road \equiv E'
\end{array}
$$

Figure 3.4 gives an example of this mapping. It satisfies all the requirements for a Transport-$C_1F_1M_1$ instance and can clearly be calculated in polynomial time. To provide some intuition for this transformation, we will call $v \in V$ an *entrance* and v^* an *exit*. Thus, portables need to be delivered from each entrance to the corresponding exit, which is connected to the entrance by an edge. An edge in the original graph translates to crosswise connections of exits and entrances of the two adjacent nodes.

Let $n = |V|$. Assume that (V, E) contains a Hamiltonian path $[v_1, \dots, v_n]$ (starting at v_1). Then the following action sequence is a plan for I:
$[pick_{m,p_{v_1}}, move_{m,v_1^*}, drop_{m,p_{v_1}}, move_{m,v_2}, \dots, move_{m,v_n}, pick_{m,p_{v_n}}, move_{m,v_n^*}, drop_{m,p_{v_n}}]$.

Now assume that there is a plan for I. Again, each location can be left at most once, and every location must be visited at least once (either to pick up or drop a portable there). By the same reasoning as in the last proof, we can assume that every location is visited exactly once, and at the only visit of each location, either a *pick* or *drop* must be performed. Due to the carrying capacity constraint, there can never be two *pick* actions without a *drop* in between. It follows that after picking up a portable at an entrance, the mobile must next move to the corresponding exit and drop that portable.

Again, by the same reasoning as in the last proof, tracing the order in which the entrances are visited gives a Hamiltonian path in the original graph, starting at v_1. ■

We are now done with our analysis of the PLANEX problem for the TRANSPORT family. The results we have proved and the specialization ordering on the domains are all we need to be able to conclude that all the fuel-constrained variants are **NP**-complete with regard to this problem, and all the other ones are polynomial.

3.4 PLANLEN for the TRANSPORT Family

Because plan lengths in the TRANSPORT family can be bounded by a polynomial, we already know that the respective PLANLEN problems are in **NP** (Theorem 3.3).

Because these problems cannot be easier than their PLANEX counterparts (Lemma 2.9), the results just presented also imply that PLANLEN is **NP**-hard as soon as fuel is restricted.

We will now investigate domains without fuel restrictions. By showing that PLANLEN is **NP**-complete for the two most specific of these domains, TRANSPORT-$C_\infty F_\infty M_1$ and TRANSPORT-$C_1 F_\infty M_1$, we will prove **NP**-completeness of PLANLEN in the whole TRANSPORT family. Again, we will first consider the case without capacity restrictions, giving three hardness proofs for this problem, each applying to a different restriction of TRANSPORT-$C_\infty F_\infty M_1$. Our first proof will be very similar to the one in Theorem 3.9 on page 24.

Theorem 3.11 NP-completeness of PLANLEN-TRANSPORT-$C_\infty F_\infty M_1$
PLANLEN-TRANSPORT-$C_\infty F_\infty M_1$ is **NP***-complete, even if the roadmap is restricted to be a planar graph and all portables start at the same location as the mobile.*
Proof: Membership in **NP** has already been proved (special case of TRANSPORT).

Given a HAMILTONIAN PATH - FIXED START - PLANAR instance $((V, E), v_1)$, we map it to the same instance as in Theorem 3.9 and set $L = 3(|V| - 1)$. Again, Figure 3.3 on page 24 gives an example of this mapping. If a Hamiltonian path exists, then the corresponding plan from Theorem 3.9 has length $3(|V| - 1)$.

If on the other hand such a plan exists, it must contain at least $|V| - 1$ *move*, *pick* and *drop* actions each in order to pick up and drop every portable and visit every location. This implies that *exactly* $|V| - 1$ *move* actions are part of the plan, describing a Hamiltonian path in (V, E). ■

The next proof we want to give applies to roadmaps which are complete graphs, a case that is particularly interesting for analyzing the LOGISTICS domain. We will first define the problem the transformation is based on.

Definition 3.12 FEEDBACK VERTEX SET
Given a directed graph $G = (V, A)$ and natural number $K \leq |V|$, is there a subset $V' \subseteq V$ with $|V'| \leq K$ such that V' contains at least one vertex from every directed cycle in G?

Another way of formulating this is asking for V' such that the graph induced by $V \setminus V'$ does not contain any directed cycles. That is, V' contains all the "feedback vertices" of the graph, hence the name of the problem. This problem is cited as being **NP**-complete in [Garey and Johnson, 1979].

Theorem 3.13 NP-completeness of PLANLEN-TRANSPORT-$C_\infty F_\infty M_1$
PLANLEN-TRANSPORT-$C_\infty F_\infty M_1$ *is* **NP***-complete, even if the roadmap is restricted to be a complete graph.*
Proof: Membership in **NP** has already been proved (special case of TRANSPORT).
Hardness is shown by a reduction from FEEDBACK VERTEX SET, using the following mapping (which can easily be calculated in polynomial time):

$$((V, A), K) \mapsto (I, L) = ((V', E', M, P, P_G, loc_0, loc_G, cap, fuel, road), L) \text{ where:}$$

$$\begin{aligned} V' &\overset{\text{def}}{=} V \cup \{s\} & loc_0(p_{u,v}) &\overset{\text{def}}{=} u \text{ for } p_{u,v} \in P \\ E' &\overset{\text{def}}{=} \{\ \{u, v\} \mid u, v \in V', u \neq v\ \} & loc_G(p_{u,v}) &\overset{\text{def}}{=} v \text{ for } p_{u,v} \in P \\ M &\overset{\text{def}}{=} \{m\} & cap &\equiv |P| \\ P &\overset{\text{def}}{=} \{\ p_{s,v} \mid v \in V\ \} \cup \{\ p_{u,v} \mid (u, v) \in A\ \} & fuel &\equiv \infty \\ P_G &\overset{\text{def}}{=} P & road &\equiv E' \\ loc_0(m) &\overset{\text{def}}{=} s & L &\overset{\text{def}}{=} 2|A| + 3|V| + K \end{aligned}$$

This mapping satisfies all the requirements for a TRANSPORT-$C_\infty F_\infty M_1$ instance with a complete graph as a roadmap. Figure 3.5 gives an example.

We call s the *start location* (the mobile starts here). For every other vertex in the graph, the start location contains a portable $p_{s,v}$ to be moved to that location. This is to ensure that all the locations have to be visited by the mobile. For every arc from u to v in the graph, there is a portable to be moved from u to v.

Assume that (V, A) contains a feedback vertex set V' of size at most K. We then define $V_0 = V \setminus V'$. Let $l \in ord(V_0)$ be a topological sort of the subgraph induced by V_0. Such a sequence exists because that graph no longer contains any directed cycles by definition of a feedback vertex set.

This leads to the following action sequence: Pick up all the portables at the start location, visit all the vertices from V', in any order, to pick up all portables lying there, then move to the vertices from V_0 in the order imposed by l, picking up all the portables to be moved and dropping them when their goal location is reached, then finally visit the vertices from V' again, dropping all the portables that need to be put there.

This constitutes a plan: Every portable to be moved to some vertex from V' will be delivered, as those vertices are visited after every vertex in the graph has been visited at

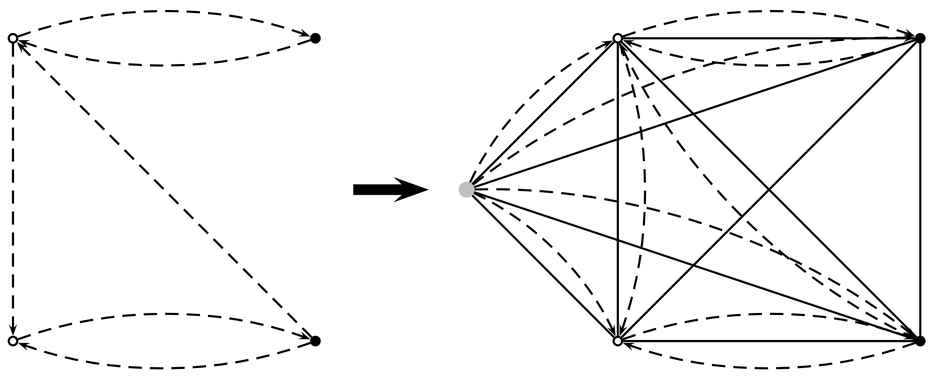


Figure 3.5: Graph and corresponding Transport-$C_\infty F_\infty M_1$ instance. Portables are displayed as dotted lines pointing from their initial to goal locations. The initial location of the mobile is drawn as a bigger shaded circle. Hollow circles depict vertices that form a feedback set or locations that should be visited twice, respectively.

least once, hence there was an opportunity to pick up that portable. Every portable to be moved to some vertex $v \in V_0$ either originates from s or V' or some vertex $u \in V_0$ that appears before v in the sequence l (as it is topologically sorted). The only other vertex is s, and no portable needs to be moved there.

This plan contains $2(|A| + |V|)$ pickup and drop actions, $|V_0|$ move actions for visiting the nodes in V_0 and $2|V'|$ move actions for visiting the nodes from V' (twice). Hence, its length is $2(|A| + |V|) + |V_0| + 2|V'| = 2|A| + 2|V| + \underbrace{|V_0| + |V'|}_{|V|} + \underbrace{|V'|}_{\leq K} \leq 2|A| + 3|V| + K$.

Now assume that there is a plan of length at most L for I. To deliver all portables, every location needs to be visited at least once, leading to at least $|V|$ movement actions. Every portable needs to be picked up and dropped at least once, leading to another $2(|V| + |A|)$ actions. Thus, there are only up to K actions remaining, which means that no more than K locations can be visited more than once.

The set of locations from V that are visited twice form a feedback vertex set, because the assumption that there still is a cycle in the graph where those vertices with their adjacent arcs have been removed leads to a contradiction: There is no possible way in which the portables corresponding to that cycle could have been delivered without visiting at least one of the locations involved twice. ■

This result shows that it is not (only) the route planning aspect of the transportation problems that makes them hard, as there is no real route-planning involved when the roadmap is a complete graph. The source of difficulty rather seems to be the interaction between the individual subgoals, i. e. goal portables. This is an observation that applies to many of the proofs shown in this chapter.

The last proof of this section addresses the limited capacity problem we have not discussed yet as well as the unrestricted capacity case. While it would not be difficult to adapt Theorem 3.10 to get a hardness proof for PlanLen-Transport-$C_1 F_\infty M_1$, we prefer to investigate a more interestingly restricted special case of that problem to get the hardness result. Specifically, we are considering instances that feature *grid graphs*, a term to be defined now.

Definition 3.14 Grid graph
For $w, h \in \mathbb{N}_0$, the **standard grid** Grid$[w, h]$ *with* **width** $w + 1$ *and* **height** $h + 1$ *is defined as the graph with the vertex set $V = \{0, \dots, w\} \times \{0, \dots, h\}$ and the edge set $E = \{ \{(a, b), (a', b')\} \subseteq V \mid |a - a'| + |b - b'| = 1 \}$.*

A graph is called a **grid** *if it is isomorphic to a standard grid.*

Note that grid graphs are always planar.

Our **NP**-completeness proof is based on a variant of the well-known traveling salesman problem.

Definition 3.15 Traveling Salesman $\mathcal{L}_1$ Metric
Given a finite set $S \subseteq \mathbb{N}_0 \times \mathbb{N}_0$ of points (sites) in the plane and a natural number B, is there an ordering $l \in$ order(S) such that $\sum_{i=1}^{|S|-1} d(l_i, l_{i+1}) + d(l_{|S|}, l_1) \leq B$, where the $\mathcal{L}_1$ or Manhattan metric d is defined as $d((x_1, y_1), (x_2, y_2)) = |x_2 - x_1| + |y_2 - y_1|$?

[Garey and Johnson, 1979] cite this problem as being **NP**-complete in the strong sense. We will assume that the encoding size of a number is linear (rather than logarithmic) in its magnitude; otherwise our transformation would not be polynomial. For problems that are **NP**-complete in the strong sense, this assumption is valid.

Theorem 3.16 NP-completeness of PlanLen-Transport-$C_{\infty/1}F_\infty M_1$
The PlanLen *problems for* Transport-$C_\infty F_\infty M_1$ *and* Transport-$C_1 F_\infty M_1$ *are* **NP**-*complete, even if the roadmap is restricted to be a grid and all portables only need to be moved to adjacent locations.*

Proof: Membership in **NP** has already been proved (special case of Transport).

Let (S, B) be a Traveling Salesman $\mathcal{L}_1$ Metric instance. We omit the trivial case of $S = \emptyset$ and assume that $n = |S| > 0$. We first calculate the maximum x and y coordinates of all sites in S, calling them $x_{\max}$ and $y_{\max}$, respectively. Let $s^* = (s^*_x, y_{\max})$ be any site in S that maximizes the y coordinate. Let $S' = S \setminus \{s^*\}$. Furthermore, we set $K = 2n$ and $D = Kn \cdot (x_{\max} + y_{\max}) + 4n - 1$.

We then map (S, B) to the following PlanLen-Transport-$C_\infty F_\infty M_1$ instance that can easily be calculated in polynomial time:

$$(S, B) \mapsto (I, L) = ((V, E, M, P, P_G, loc_0, loc_G, cap, fuel, road), L) \text{ where:}$$

$$\begin{aligned}
(V, E) &\overset{\text{def}}{=} \text{Grid}[Kx_{\max} + 1, Ky_{\max} + D] & loc_0(p_{(x,y)}) &\overset{\text{def}}{=} (Kx, Ky) \text{ for } (x, y) \in S' \\
M &\overset{\text{def}}{=} \{m\} & loc_G(p) &\overset{\text{def}}{=} loc_0(p) + (1, 0) \text{ for } p \in P \\
P &\overset{\text{def}}{=} \{ p_s \mid s \in S \} & cap &\equiv \infty \\
P_G &\overset{\text{def}}{=} P & fuel &\equiv \infty \\
loc_0(m) &\overset{\text{def}}{=} (Ks^*_x, Ky_{\max}) & road &\equiv E \\
loc_0(p_{s^*}) &\overset{\text{def}}{=} (Ks^*_x, Ky_{\max} + D) & L &\overset{\text{def}}{=} KB + D + 4n - 1
\end{aligned}$$

This mapping satisfies all the requirements for a Transport-$C_\infty F_\infty M_1$ instance with a grid graph as a roadmap and all portables starting out next to their goal location. By setting $cap \equiv 1$, the very same reduction can be used for Transport-$C_1 F_\infty M_1$ instead – we will only have to consider plans where the mobile never carries more than one portable at once, so this will not cause any problems.

We will restate this definition in words to make it more understandable. First, we scale all coordinates by a factor K. Then, we place a portable at each site from the

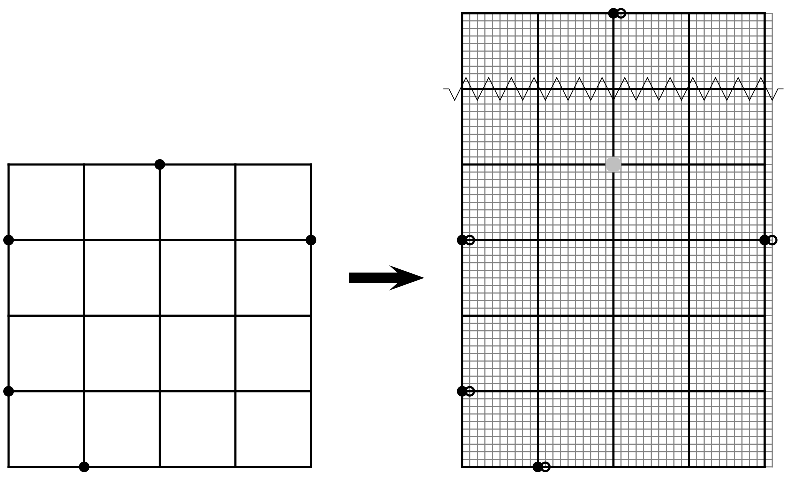


Figure 3.6: Traveling Salesman $\mathcal{L}_1$ Metric sites (as points on the $\mathbb{N}_0 \times \mathbb{N}_0$ grid) and the corresponding Transport-$C_\infty F_\infty M_1$ instance, solid circles indicating initial positions and adjacent hollow circles indicating goal positions for portables. The bigger shaded circle indicates the initial position of the mobile. The zigzag line indicates that the distance between the initial position and the topmost portable is not to the scale — the portable should be further up.

traveling salesman problem excluding the northmost one, where we place the mobile. We place an additional portable far up north (D units after scaling), on the same column as the mobile. We call this the *final portable*. The goal is to move all the portables one unit to the right. Figure 3.6 gives an example of this mapping.

Assume that there is a traveling salesman tour of length at most B for the sites S. It is easy to see that there is a strong connection between distances in the $\mathcal{L}_1$ metric and movements in the planning instance: If $d(p, q) = k$, then the shortest action sequence to move from p to q on a standard grid consists of k actions.

Thus, taking the scaling constant K into account, there is a sequence of *move* actions of length at most KB that passes through all the portables' initial locations (except for the final portable) and then returns to the location of origin. With another D movements, the mobile can then reach the final portable. All that remains to be done now is to insert four actions whenever a location containing a portable is encountered: pick up that portable, move east, drop it, move west again, where the last movement is not needed for the final portable. This leads to a plan of length at most $KB + D + 4n - 1 = L$.

Now assume that there is *no* traveling salesman tour of length B or less, i. e. the shortest tour has length $B + 1$ or more. We will show that all plans consist of more than L steps. First note that the distance between any two sites is always strictly less than $x_{\max} + y_{\max}$, because it is $x_{\max} + y_{\max} - 2$ in the most extreme case of the sites being located in opposite corners of the grid. Because a tour is the sum of n distances, there is *always* a traveling salesman tour that is shorter than $n(x_{\max} + y_{\max})$, so B must be less than this value. This implies that $D > KB + 4n - 1$.

First consider any plan where the final portable is picked up for the first time at some point before all the other portables have been moved to their goal locations. In this case the mobile will at some point have to move from row $Ky_{\max}$ to row $Ky_{\max} + D$ to get to the initial location of that portable and later get back to row $Ky_{\max}$ or below to drop other portables. This will involve at least $2D$ movements, thus plan length will be at least $2D > KB + D + 4n - 1 = L$, exceeding the boundary.

Therefore we only need to consider the case where the final portable is picked up for the first time after all the other portables have been dropped at their goal location. We can safely assume that the movement between the last but one portable to be dropped and the final portable passes through the initial location of the mobile. If it does not, the movement path can be changed to achieve this without increasing plan length by first moving eastwards or westwards until the column of the final portable has been reached, then going northwards.

For each portable, the mobile must at some point move to the initial location of that portable, and it will return to its own initial location at some point. The number of *move* actions to achieve this cannot be less than the length of the shortest traveling salesman tour for the set of sites that is given by scaling each site in S by a factor of K. Thus, at least $K(B+1)$ move actions will be needed for this, plus D move actions for getting from the initial location to the location of the final portable.

In addition, at least $2n$ *pick* and *drop* actions are needed, totalling a lower bound of $K(B+1) + D + 2n = KB + K + D + 2n = KB + 2n + D + 2n = KB + D + 4n > L$ actions. Thus, no plan of length at most L exists. ■

3.5 Parallel Plans in the Transport Family

In AI planning, sequential length is not the only accepted quality criterion for a plan. People are also interested in short *parallel plans*, where several actions can be applied at the same time, provided they do not interfere, where the precise meaning of *interference* needs to be specified.

In the Transport framework, it would make sense to allow different mobiles to act at the same time, provided they do not operate on the same portables, and perhaps allow individual mobiles to pick up or drop multiple portables at the same time.[2]

How would the complexity results we proved change if parallel activity was taken into account and the number of *time steps* (with potentially many actions happening at the same time step) was investigated in place of sequential plan length?

In fact, Theorems 3.11, 3.13 and 3.16 still apply in the parallel case. This is not hard to see, as all proofs involve only one mobile and the only possible parallelism to be exploited is the possibility of dropping and picking up multiple portables at the same time.

For Theorem 3.11, $L = 3(|V| - 1)$ would need to be be changed to $2(|V| - 1) + 1 = 2|V| - 1$, as all the *pick* actions can now be performed in one step.

For Theorem 3.13, $L = 2|A| + 3|V| + K$ would become $2(|V| + K) + 1$, counting $|V| + K$ movements and one step each in between every two movements, before the first movement and after the last movement to do all the necessary picking up and dropping.

For Theorem 3.16, nothing needs to be changed.

Thus, all the complexity results we have proved for the Transport family still apply in the parallel case.

[2]If the popular notion of **Graphplan** ([Blum and Furst, 1997]) parallelism is used, these two kinds of parallel activity are allowed in the usual PDDL encodings of the Logistics and Gripper domains. In Mystery and Mystery', a mobile may only pick up or drop one portable per time step, and different mobiles may only move simultaneously if they are at different locations. Parallel movement or pickup cannot occur in Grid because there is only one mobile, and it can only carry one portable at a time.

3.6 Summary of the Transport Family

With these results, we conclude our analysis of the Transport family. The most important results are summarized in Figure 3.7.

- PlanEx is polynomial if there is unlimited fuel.
- PlanEx is **NP**-complete otherwise.
- PlanLen is always **NP**-complete.

Special cases for which both problems are still **NP**-complete:

- One mobile of unlimited capacity, one unit of fuel at every location, all portables start at same location as mobile, roadmap is a planar graph.
- One mobile of capacity one, one unit of fuel at every location, all portables only need to be moved to adjacent locations.

Other special cases for which PlanLen is still **NP**-complete:

- One mobile of unlimited capacity, unlimited fuel, all portables start at same location as mobile, roadmap is a planar graph.
- One mobile of unlimited capacity, unlimited fuel, roadmap is a complete graph.
- One mobile of capacity one, unlimited fuel, all portables only need to be moved to adjacent locations, roadmap is a grid.

Figure 3.7: Complexity results for the Transport domain family.

3.7 Gripper, Logistics, Mystery and Mystery'

So far, no results have been presented for the transportation domains used in the AIPS competitions. However, we will see that these fit nicely into the framework introduced in the preceding sections.

Definition 3.17 Gripper **domain**
The Gripper *domain is the special case of* Transport-$C_*F_\infty M_1$ *where for all instances* $I = (V, E, M, P, P_G, loc_0, loc_G, cap, fuel, road)$ *the following holds:*

$$\begin{aligned} V &\overset{\text{def}}{=} \{room_1, room_2\} & l_0 &\equiv room_1 \\ E &\overset{\text{def}}{=} \{\{room_1, room_2\}\} & l_G &\equiv room_2 \\ P_G &\overset{\text{def}}{=} P & cap &\equiv 2 \end{aligned}$$

The result from Transport-$C_*F_\infty M_*$ implies that PlanEx-Gripper $\in$ **P**. But it is not hard to see that Gripper instances are easy to solve even if optimal plans are required, as this is a very restricted domain: An instance is characterized by a single parameter, the number of portables b (for *balls*).

Theorem 3.18 PLANLEN-GRIPPER $\in$ **P**
PLANLEN-GRIPPER *can be solved in polynomial time.*
Proof: The following strategy leads to an optimal plan: If there are at least two portables at location $room_1$, choose any two of them and pick them up. If there is only one portable, pick that one up. Otherwise nothing needs to be done. Then move to the other room and drop them. If the goal is not yet satisfied, move back and iterate. It is obvious that this is an optimal sequential plan.

The plan can be written down in polynomial time in the input size (which we defined to be $|V| + |M| + |P| = |P| + 3$). ■

It is easy to adapt this algorithm to the parallel case where several portables can be picked up or dropped at the same time.

Note that the PDDL encoding of GRIPPER instances is not *reasonably concise* from a complexity theory point of view. Under a concise encoding of the instances, plan lengths are exponential in the length of the input.[3]

We will now move to more interesting domains.

Definition 3.19 LOGISTICS **domain**
The LOGISTICS *domain is the restriction of* TRANSPORT-$C_\infty F_\infty M_*$ *to those instances* $I = (V, E, M, P, P_G, loc_0, loc_G, cap, fuel, road)$*, where there exists a set of locations* $A \subseteq V$ *(called* airports*) such that for all* $m \in M$*, the movement graph* $(\bigcup_{e \in road(m)} e, road(m))$ *of that mobile is a complete graph and either has vertex set* A *(in which case* m *is called an* airplane*), or shares exactly one vertex with* A *(in which case* m *is called a* truck*). Moreover, for any two trucks* m*,* m'*, it must be the case that either* $road(m) = road(m')$ *or the two roadmaps are disjoint.*

As a special case of TRANSPORT-$C_* F_\infty M_*$, PLANEX-LOGISTICS $\in$ **P**.

The PLANLEN problem in this domain is **NP**-complete, even in the case where there is only one truck and no airplane or equivalently only one airplane and no truck. This was proved in Theorem 3.13 and also applies to the parallel case.

Definition 3.20 MYSTERY **and** MYSTERY' **domains**
The MYSTERY *domain is the restriction of* TRANSPORT-$C_* F_* M_+$ *to instances for which the roadmap is a planar graph.*

The MYSTERY' *domain is an extended version of* MYSTERY *where an additional type of actions allows transferring one unit of fuel from any location that has at least two units two any other location.*

Theorem 3.21 NP-completeness of PLANEX-MYSTERY **and** PLANEX-MYSTERY'
The PLANEX *problem for* MYSTERY *and* MYSTERY' *is* **NP***-complete.*
Proof: As a special case of TRANSPORT, PLANEX-MYSTERY $\in$ **NP**. As a generalization of TRANSPORT-$C_1 F_1 M_1$ with planar roadmaps, it is **NP**-hard and thus **NP**-complete.

[3]However, the decision problems are still solvable in polynomial time, since plans *always* exist, so PLANEX is trivial, and the length of a shortest sequential or parallel plan for b balls can easily be calculated.

For MYSTERY', we can restrict our attention to plans where no fuel is moved after the first *move*, *pick* or *drop* action. Thus, the situation after the last fuel movement corresponds to a MYSTERY instance and thus if there is any plan, there is one for which the number of *move*, *pick* and *drop* actions is bounded by a polynomial in the input size. It does not make sense to both move fuel to and from a given location in the same plan, as it could have been moved directly to its final destination. No location needs more than a polynomial amount of fuel and thus each location need only be the target of a move fuel action for a polynomial number of times. Thus, the total number of fuel movements necessary can also be bounded by a polynomial. Therefore Lemma 2.10 can be applied, and PLANEX-MYSTERY' $\in$ NP.

If there is no more than one unit of fuel at every location, MYSTERY' instances and corresponding MYSTERY instances are identical. Thus, MYSTERY' is also a generalization of TRANSPORT-$C_1F_1M_1$ with planar roadmaps and thus PLANEX-MYSTERY' is **NP**-hard and thus **NP**-complete. ■

This result of course implies that PLANLEN is **NP**-complete for the MYSTERY and MYSTERY' domains (membership in **NP** again following from a polynomial plan length argument).

3.8 MICONIC-10

The two remaining transportation domains to be examined are sufficiently different from the TRANSPORT family to warrant their own sections. The Miconic-10 elevator domain from the AIPS 2000 competition is actually a family of three domains, one of them encoded in the STRIPS subset of PDDL, the other ones making use of some ADL extensions.

The MICONIC-10 STRIPS domain is very similar to TRANSPORT-$C_\infty F_\infty M_1$ with a complete graph, the mobile being called the *elevator*, the portables going by the name of *passengers*. Differently to TRANSPORT-$C_\infty F_\infty M_1$ however, it is not possible to have a passenger leave the elevator at any location other than his goal location, and once a passenger has left the elevator, he can never board it again. Additionally, in all variants of the MICONIC-10 domain, initial and goal locations of a passenger must always be different and a goal location is specified for each passenger (rather than just a subset of passengers).

Despite these differences, Theorem 3.13 still applies, so PLANEX is polynomial and PLANLEN is **NP**-complete in this variant of MICONIC-10.

The first, simplified ADL domain is quite similar, with just one difference: Rather than individual *pick* and *drop* operators, there is just one *stop* action that makes all passengers who have reached their final destination leave the elevator, while the ones waiting outside board it. The same results apply, although the definition of L in Theorem 3.13 must be changed to $2(|V| + K) + 1$, just like in the case where portables can be picked up and dropped in parallel.

The other ADL domain exhibits some additional features that require closer investigation. As a generalization of its simplified brother, the PLANLEN problem is still **NP**-hard.

For membership in **NP**, it suffices to observe that plan lengths can safely be bounded by a polynomial. This is indeed the case: Passengers can only ever board and leave the elevator once, it does not make sense to perform a *stop* action which does not cause at least one passenger to board or leave the elevator, and there should always be exactly one *move* action in between to stops (and no more than one before the first stop). Therefore, plan lengths in solvable instances can always be bounded by four times the number of passengers.

But how about PLANEX? Although no fuel constraints are present in this transportation domain, we will see that PLANEX is actually **NP**-hard (and thus **NP**-complete) due to the various restrictions imposed on elevator movement for special passengers. To prove this, we will have to introduce the domain in a more formal way than the previous ones. Because this is the "real" Miconic-10 elevator domain, we will just call it MICONIC-10.

Definition 3.22 MICONIC-10 **instance**
A MICONIC-10 **instance** *is a 13-tuple*
$I = (F, f_0, P, loc_0, loc_G, P_V, P_N, P_D, P_A, P_S, P_1, P_2, Acc)$*, where*

- $F \subseteq \mathbb{N}_0$ *is the finite set of* **floors***,*
- $f_0 \in F$ *is the* **initial floor***,*
- P *is the finite set of* **passengers***,*
- $loc_0 : P \to F$ *is the* **initial location** *function,*
- $loc_G : P \to F$ *is the* **goal location** *function, satisfying* $loc_G(p) \neq loc_0(p)$ *for* $p \in P$*,*
- $P_V \subseteq P$ *is the set of* **VIP passengers***,*
- $P_N \subseteq P$ *is the set of* **non-stop passengers***,*
- $P_D \subseteq P$ *is the set of* **direct travel passengers***,*
- $P_A \subseteq P$ *is the set of* **attendants***,*
- $P_S \subseteq P$ *is the set of* **attended passengers** *(S for supervised),*
- $P_1 \subseteq P$ *is the set of* **group one passengers***,*
- $P_2 \subseteq P$ *is the set of* **group two passengers** *and finally*
- $Acc \subseteq P \times F$ *is the* **access** *relation.*

The encoding length of the instance is assumed to be $|F| + |P|$*.*

MICONIC-10 instances are very similar to TRANSPORT instances as defined in Definition 3.1. Apart from the various passenger subsets that will be explained together with the operators of the domain, two things should be commented on.

Firstly, the requirement for the floors to be natural numbers is just an easy way to get a total (vertical) order on floors, which is needed for modelling direct travel passengers.

Secondly, the access relation models restricted access to certain floors in the building: An elevator may only open its doors at a given floor if all passengers inside have access to that floor as specified by this relation. This is an example of a *dynamic roadmap* as mentioned in the introduction to this chapter.

Definition 3.23 Miconic-10 **domain**
Let L be the set of all Miconic-10 *instances. The* Miconic-10 **domain** *is the function $\mathcal{D} : L \to \mathcal{M}$ mapping instances to state models as follows:*

$$
\begin{aligned}
&(F, f_0, P, loc_0, loc_G, P_V, P_N, P_D, P_A, P_S, P_1, P_2, Acc) \mapsto (S, s_0, S_G, A, \delta), \textit{ where}\\
&S \overset{\text{def}}{=} F \times \mathbb{P}(P) \times \mathbb{P}(P),\\
&s_0 \overset{\text{def}}{=} (f_0, \emptyset, \emptyset),\\
&S_G \overset{\text{def}}{=} F \times \{\emptyset\} \times \{P\},\\
&A \overset{\text{def}}{=} \{\ move_f \mid f \in F\ \} \cup \{stop\},\\
&\forall f' \in F, s = (f, P_E, P_G) \in S :\\
&\qquad f \neq f'\\
&\quad \wedge\ \forall p \in P_D \cap P_E : (\operatorname{sgn}(loc_G(p) - f) = \operatorname{sgn}(f' - f) \vee loc_G(p) = f')\\
&\quad \Rightarrow \delta(s, move_{f'}) \overset{\text{def}}{=} (f', P_E, P_G),\\
&\forall s = (f, P_E, P_G) \in S, P'_E \subseteq P, F_V \subseteq F :\\
&\qquad P'_E = (P_E \setminus loc_G^{-1}(\{f\})) \cup (loc_0^{-1}(\{f\}) \setminus P_G)\\
&\quad \wedge\ F_V = loc_0(P_V \setminus (P_E \cup P_G)) \cup loc_G(P_V \cap P_E)\\
&\quad \wedge\ (f \in F_V \vee F_V = \emptyset)\\
&\quad \wedge\ (P_N \cap P_E = \emptyset \vee f \in loc_G(P_N \cap P_E))\\
&\quad \wedge\ (P_S \cap P'_E \neq \emptyset \Rightarrow P_A \cap P'_E \neq \emptyset)\\
&\quad \wedge\ (P_1 \cap P'_E = \emptyset \vee P_2 \cap P'_E = \emptyset)\\
&\quad \wedge\ \forall p \in P_E : (p, f) \in Acc\\
&\quad \Rightarrow \delta(s, stop) \overset{\text{def}}{=} (f, P'_E, P_G \cup (loc_G^{-1}(\{f\}) \cap P_E)) \textit{ and finally}\\
&\delta \textit{ is undefined otherwise.}
\end{aligned}
$$

In this state model, a state consists of the current floor f, the set of passengers P_E in the elevator and the set of passengers P_G that have been served already. In goal states, there are no passengers in the elevator and all passengers have been served. The current floor does not matter. P_E and P_G are disjoint in any reachable state.

The *move* operator is parameterized by the destination floor and has the obvious effect. If there is a direct travel passenger in the cabin, then the elevator is required to move upwards if the goal location of that passenger is above the current floor, and it must not move past it; the converse is true for passengers going down. A case distinction is avoided in the formalization by making use of the sgn function.

A single *stop* action models the elevator opening its doors, allowing passengers to board or leave. As a result of this action, all passengers inside the elevator that have reached their goal destination leave the cabin, and all passengers waiting outside enter. The new set of passengers inside the elevator is referred to as P'_E in the definition of the semantics of *stop*.

The elevator may only stop if the following requirements are met:

- The destination floor currently is a *VIP floor*, i. e. a VIP passenger is waiting there, or a VIP passenger inside the elevator wants to get there, or there are no VIP floors, implying that all VIPs have been served already.[4] Note that the set of VIP floors

[4]This definition follows the textual description in [Koehler and Schuster, 2000], although the actual PDDL domain always allows for movement to the initial or goal location of a VIP, even if this VIP has been served already, which seems to be a bug.

is referred to as F_V in the definition.

- There are no non-stop passengers in the elevator or the elevator is traveling to the goal location of one of the non-stop passengers currently in it.[5]
- If the stop would result in an attended passenger being in the elevator, then it must also result in an attendant being in it.
- After the stop there must not be passengers from both group one and group two in the cabin.
- All the passengers in the elevator must have access to the destination floor.

For a motivation of the various features of the MICONIC-10 domain, we refer to [Koehler and Schuster, 2000], where it was first introduced.

A proof of **NP**-completeness for PLANEX-MICONIC-10 concludes this section.

Theorem 3.24 NP-completeness of PLANEX-MICONIC-10
PLANEX-MICONIC-10 *is* **NP***-complete, even without direct travel or non-stop passengers.*
Proof: Membership in **NP** has already been shown. We will prove **NP**-hardness by a reduction from DIRECTED HAMILTONIAN PATH - FIXED START, the directed graph variant of HAMILTONIAN PATH - FIXED START. This problem is **NP**-hard because with regard to Hamiltonian paths, directed graphs are a generalization of undirected graphs.

Given a DIRECTED HAMILTONIAN PATH - FIXED START instance $((V, A), v_1)$, we describe a polynomial algorithm for constructing a membership-equivalent MICONIC-10 instance $(F, f_0, P, loc_0, loc_G, P_V, P_N, P_D, P_A, P_S, P_1, P_2, Acc)$, thus proving DIRECTED HAMILTONIAN PATH - FIXED START $\leq_p$ PLANEX-MICONIC-10.

Because the actual natural numbers for floors are only important for direct travel passengers who will not be used in the reduction, we will refer to floors with symbols like f_v rather than numbers. Different symbols mean different floors.

The set of floors consists of the following:

- Init and final floors f_0 and f_∞.
- For all $v \in V$, a vertex start floor f_v and vertex end floor f_v^*.
- For all arcs $(u, v) \in A$ an arc floor $f_{u,v}$.

Thus, $|F| = 2 + 2|V| + |A|$.

The other features of the instance are defined as follows:

- p_0, the **init passenger**, satisfies $loc_0(p_0) = f_0$ and $loc_G(p_0) = f_{v_1}$. He only has access to the init floor and the vertex start floor of v_1. He is a VIP and an attendant (member of P_V and P_A).
- For all $v \in V$, p_v is called a **vertex start passenger**, with $loc_0(p_v) = f_0$ and $loc_G(p_v) = f_v$. He has access to all floors, is an attended passenger and belongs to group one (member of P_S and P_1).

[5]Again, this follows the textual description in [Koehler and Schuster, 2000], although there seems to be another bug in the PDDL domain which leads to the elevator never re-opening its doors if several non-stop passengers with different goal locations enter it.

- For all $v \in V$, p_v^* is called a **vertex end passenger**, with $loc_0(p_v^*) = f_v$ and $loc_G(p_v^*) = f_v^*$. He only has access to the vertex start and end floors of v, the arc floors for outgoing arcs of v and the final floor. He is an attendant (member of P_A).
- For all $(u, v) \in A$, $p_{u,v}$ is called an **arc passenger**, with $loc_0(p_{u,v}) = f_{u,v}$ and $loc_G(p_{u,v}) = f_v$. He only has access to the vertex end floor of u and the vertex start floor of v. He is an attendant (member of P_A).
- For all $v \in V$, p_v^∞ is called a **no vertex return passenger**, with $loc_0(p_v^\infty) = f_v^*$ and $loc_G(p_v^\infty) = f^\infty$. He has access to all floors except the vertex start floor of v.
- p_∞, the **final passenger**, satisfies $loc_0(p_\infty) = f_\infty$ and $loc_G(p_\infty) = f_0$. He belongs to group two (member of P_2).

Apart from these, there are no other passengers or members of P_V, P_N, P_D, P_A, P_S, P_1 or P_2 and no further elements of Acc.

The main difficulty in finding a plan for this instance lies in always having attendants in the elevator for the vertex start passengers. Most of the other passengers serve the purpose of restricting the possible next floors to stop at.

The first floor to stop at must be the init floor, because it is the initial location of the init passenger, who is the only VIP passenger. There, in addition to the init passenger, all vertex start passengers will board, which means that an attendant will have to be present in the elevator from now on until all vertex start passengers have left, i. e. until all vertex start floors have been visited. The next floor to stop at must be the vertex start floor of v_1 because of the init passenger's access restrictions.

As long as not all vertex start floors have been visited, the following is true:

1. Whenever a vertex start floor is stopped at for the first time, a vertex end passenger will board, restricting the journey to continue either to an outgoing arc floor, the vertex end floor or the final floor. As long as not all vertex start floors have been visited, the only option is to go to an arc floor, since going to the vertex end floor would result in a lack of an attendant, and going to the final floor would result in group one and group two passengers both being inside the elevator.
2. When an arc floor for (u, v) is visited for the first time, the arc passenger will board. The journey can then only continue to the vertex end floor of u.
3. If a vertex end floor of u is visited for the first time, the no vertex return passenger will board, keeping the elevator from visiting the vertex start floor of u again. The journey will then continue to the vertex start floor of a vertex which has an ingoing arc from u, because there must be an arc passenger in the elevator (otherwise there would be no attendant in there). As long as there are still vertex start passengers on board, 1. applies again.

This means that as long as there are still vertex start passengers in the elevator, the movement of the elevator must be consistent with the arcs of the graph. During this time, no vertex can be visited twice because of the access restriction for no vertex return passengers. Thus, if a plan exists, there must be a Hamiltonian path in G, because if this were not the case, there would be no possibility of visiting every vertex start floor without visiting any of them twice.

On the other hand, if a Hamiltonian path exists, there is a sequence of actions that will lead to all vertex start passengers having arrived at their final destination and the elevator being at one of the vertex start floors. It can then immediately proceed to the vertex end floor, pick up the last no vertex return passenger and drop off the last vertex end passenger, move to the final floor as there are no more group one passengers in the elevator, drop off all the no vertex return passengers and move to the init floor to drop off the final passenger.

All that remains to be done now is serving the arc passengers of the arcs which are not part of the Hamiltonian path. This can easily be achieved by moving one arc passenger at a time, going to his initial floor and then to his goal floor.

Thus, if a Hamiltonian path exists, there is a plan for the MICONIC-10 instance, which concludes the proof. ■

3.9 GRID

The GRID domain is basically TRANSPORT-$C_1F_\infty M_1$ for grid graphs with one added feature: Rather than just being carried around, portables (called *keys* in this context) can also be employed to open up initially unaccessible (*locked*) locations. Thus, this is another example of a domain featuring dynamic roadmaps. If there are no initially locked locations, the domains are identical.[6]

Thus, applying Theorem 3.16, we already know that PLANLEN-GRID is **NP**-hard, even if all locations are initially accessible.

However, as the GRID domain is not a special case of TRANSPORT because of its dynamic roadmap, we do not have any results for PLANEX and cannot claim membership in **NP** for PLANLEN. In order to do so, we must first introduce the domain formally.

Definition 3.25 GRID **instance**
A GRID **instance** *is a 10-tuple*
$I = (V, E, l_0, P, P_G, loc_0, loc_G, S, door, type)$, *where*

- (V, E) *is a grid graph called the* **grid**,
- $l_0 \in V$ *is the* **initial robot location**,
- P *is the finite set of* **keys**,
- $P_G \subseteq P$ *is the set of* **goal keys**,
- $loc_0 : P \to V$ *is the* **initial location** *function,*
- $loc_G : P_G \to V$ *is the* **goal location** *function,*
- S *is the set of* **lock shapes**,
- $door : V \setminus \{l_0\} \nrightarrow S$ *is the* **door type** *function and finally*

[6]This is not entirely true because of one slight difference between corresponding TRANSPORT-$C_1F_\infty M_1$ and GRID instances: In GRID, it is possible to drop a key and pick up another one using only one action. This does not affect the applicability of Theorem 3.16, however.

- *type* : $P \to S$ *is the* **key type** *function.*

The encoding length of the instance is assumed to be $|V| + |P|$.

Comparing this to Definition 3.1, most parts of a Grid instance can go without explanation. In the Grid domain, each location can feature at most one door, where doors can be distinguished by the shape of their lock. If there is a door at a given location, the *door* function specifies the shape of its lock. Otherwise, that location will not be in the domain of *door*. Likewise, *type* specifies the shape of a key. A key can be used to open a lock if and only if their shapes match.

It is required that the robot does not start out at a locked location (which would be strange). The number of distinct shapes can be bounded by $|V|$ without loss of generality, because there is no need to have more shapes than locked locations (of which there cannot be more than $|V| - 1$) plus one for keys that cannot open any door. This justifies the definition of encoding length.

Definition 3.26 Grid **domain**
Let L be the set of all Grid *instances. The* Grid **domain** *is the function* $\mathcal{D} : L \to \mathcal{M}$ *mapping instances to state models as follows:*

$$
\begin{aligned}
&(V, E, l_0, P, P_G, loc_0, loc_G, S, door, type) \mapsto (S, s_0, S_G, A, \delta), \text{ where} \\
&\quad S \overset{\text{def}}{=} V \times (P \cup \{\bot\}) \times (P \to V \cup \{\bot\}) \times \mathbb{P}(V), \\
&\quad s_0 \overset{\text{def}}{=} (l_0, \bot, loc_0, \mathrm{dom}(door)), \\
&\quad S_G \overset{\text{def}}{=} \{\ (v, k, loc, lock) \in S \mid loc_G \subseteq loc\ \}, \\
&\quad A \overset{\text{def}}{=} \{\ move_v \mid v \in V\ \} \cup \{\ pick_p \mid p \in P\ \} \cup \{drop\} \\
&\qquad \cup\ \{\ swap_p \mid p \in P\ \} \cup \{\ unlock_v \mid v \in V\ \}, \\
&\forall v' \in V, s = (v, k, loc, lock) \in S : \\
&\qquad \{v, v'\} \in E \wedge v' \notin lock \\
&\quad \Rightarrow \delta(s, move_{v'}) = (v', k, loc, lock), \\
&\forall p \in P, s = (v, k, loc, lock) \in S : \\
&\qquad loc(p) = v \wedge k = \bot \\
&\quad \Rightarrow \delta(s, pick_p) = (v, p, loc \oplus \{(p, \bot)\}, lock), \\
&\forall s = (v, k, loc, lock) \in S : \\
&\qquad k \neq \bot \\
&\quad \Rightarrow \delta(s, drop) = (v, \bot, loc \oplus \{(k, v)\}, lock), \\
&\forall p \in P, s = (v, k, loc, lock) \in S : \\
&\qquad loc(p) = v \wedge k \neq \bot \\
&\quad \Rightarrow \delta(s, swap_p) = (v, p, loc \oplus \{(k, v), (p, \bot)\}, lock), \\
&\forall v' \in V, s = (v, k, loc, lock) \in S : \\
&\qquad \{v, v'\} \in E \wedge v' \in lock \wedge k \neq \bot \wedge door(v') = type(k) \\
&\quad \Rightarrow \delta(s, unlock_{v'}) = (v, k, loc, lock \setminus \{v'\}) \text{ and finally} \\
&\delta \text{ is undefined otherwise.}
\end{aligned}
$$

In this model, states consist of the current robot location, the key currently being carried (which may be none, represented as $\bot$), the current key locations (where $\bot$ is the location of a key currently being carried by the robot) and the set of locked doors. There is some redundancy in this state space, but being explicit about the key being carried facilitates the definition of δ.

The *move* operator is parameterized by the destination vertex, which must be an adjacent unlocked location.

The *pick* and *drop* actions do the obvious things. It should be noted that their definition formalizes the carrying capacity constraint of one. *swap* is a combination of *drop* and then *pick*.

The *unlock* operator is parameterized by the location of the door to be opened. This location must contain a locked door with a lock shape that fits to the key currently being carried.[7]

We will now show that PLANEX-GRID $\in$ **P** (and indeed, plans can be generated in polynomial time).

Theorem 3.27 PLANEX-GRID $\in$ **P**
PLANEX-GRID *can be solved in polynomial time.*
Proof: We devise a polynomial algorithm that generates a plan if one exists or reports that none exists. Start by calculating the set of vertices that can be accessed from the initial robot location without opening doors (the reachable vertices) and the set of keys located at these nodes (the reachable keys). This can be done in polynomial time using a breadth-first search.

Then, as long as there are still doors neighbouring a reachable vertex which have a lock that can be opened with a reachable key, move to that key, pick it up, move next to the door, unlock it and drop the key. The vertex of that door must then be added to the reachable vertex set, and all keys lying there must be added to the reachable key set. Clearly, this can be done in polynomial time, and it cannot be done more than $|V| - 1$ times, because there cannot be more than $|V| - 1$ locked doors.

A plan exists if and only if, once as many doors as possible have been opened, for all goal keys that are not at their goal position, their goal position and the keys themselves are reachable. Moving to one goal key, picking it up, moving to its goal location, dropping it and continuing in this fashion with the next key, a valid plan is constructed in polynomial time. ■

This could conclude our discussion of GRID, having already shown **NP**-completeness of PLANLEN-GRID. However, we will give another proof of **NP**-completeness for this problem that focuses on the dynamic roadmap aspect of that domain rather than the difficulty of goal ordering.

We will prove that GRID is **NP**-complete even if there is only one goal key. The hardness of the generated instances lies in deciding which doors to open in order to get access to other parts of the map, and the decision problem used in the transformation is not related to "route planning" in any obvious way. In fact, the route planning problems that form part of the generated planning instances are trivial.

[7]Note that **Graphplan** parallelism would allow for different doors next to the current robot location to be opened at the same time (as the only exploitable parallelism in this domain). This does not seem realistic, and it is only a minor issue to us, because whether or not this parallel activity is allowed does not affect the complexity of the domain — all proofs still apply in this "parallel" case, which is not hard to verify.

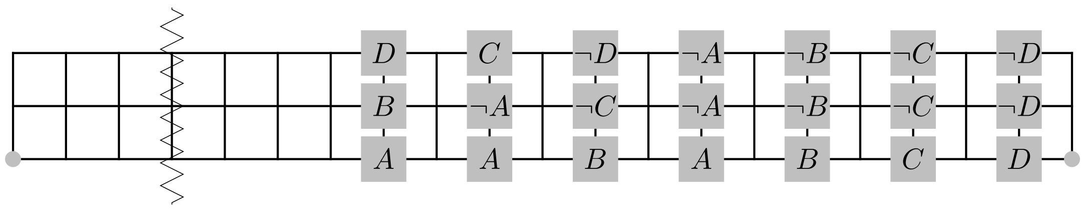


Figure 3.8: GRID instance corresponding to $(A \vee B \vee D) \wedge (A \vee \neg A \vee C) \wedge (B \vee \neg C \vee \neg D)$. Locations with doors are marked with squares, the initial and goal locations are shown as shaded circles. The zigzag indicates that the void is not to the scale (it should be wider).

Definition 3.28 3SAT
Given a set U of variables and a collection C of clauses over U such that each clause $c \in C$ has $|c| = 3$, is there a satisfying truth assignment for C?

This is perhaps the best-known **NP**-complete problem. Proofs of **NP**-completeness can be found in [Garey and Johnson, 1979] and almost any introductory text on complexity theory.

Theorem 3.29 NP-completeness of PLANLEN-GRID
PLANLEN-GRID is **NP***-complete, even in the special case where there is only one goal key, all keys and the robot start at the same location, and the height of the grid graph is restricted to three.*
Proof: Membership in **NP** follows from the existence of polynomial length plans.

Given the 3SAT instance $F = (\{v_1, \dots, v_n\}, \{\{l_{1,1}, l_{1,2}, l_{1,3}\}, \dots, \{l_{m,1}, l_{m,2}, l_{m,3}\}\})$, we first calculate $W = 3(n+m) + 2 + (2n+1)(2m+2n) + 4(n+m)^2 + 8(n+m) + 4$ and $w = W + 2m + 2n$.

The 3SAT instance is then mapped to the following PLANLEN-GRID instance:

$F \mapsto (I, L) = ((V, E, l_0, P, P_G, loc_0, loc_G, S, door, type), L)$ where:

$$\begin{aligned}
(V, E) &\overset{\text{def}}{=} \text{Grid}[w, 2] \\
l_0 &\overset{\text{def}}{=} (0, 0) \\
P &\overset{\text{def}}{=} \{goal, v_1, \dots, v_n, \neg v_1, \dots, \neg v_n\} \\
P_G &\overset{\text{def}}{=} \{goal\} \\
loc_0 &\equiv (0, 0) \\
loc_G &\equiv (w, 0) \\
S &\overset{\text{def}}{=} P \\
door((W + 2i - 1, j - 1)) &\overset{\text{def}}{=} l_{i,j} \text{ for } i \in \{1, \dots, m\},\ j \in \{1, 2, 3\} \\
door((W + 2(m+i) - 1, 0)) &\overset{\text{def}}{=} v_i \text{ for } i \in \{1, \dots, n\} \\
door((W + 2(m+i) - 1, j)) &\overset{\text{def}}{=} \neg v_i \text{ for } i \in \{1, \dots, n\},\ j \in \{1, 2\} \\
v &\notin \text{dom}(door) \text{ otherwise} \\
type(p) &\overset{\text{def}}{=} p \text{ for } p \in P \\
L &\overset{\text{def}}{=} (2n+2)W
\end{aligned}$$

This mapping satisfies all requirements including polynomial time computability. For an example, see Figure 3.8.

The mapping defines a Grid instance which features a very long corridor. On its far left, there is the initial location of the robot and all keys, with a wide open space of width W extending to the right, called the *void*. At the right end of that space, there is an area with many doors organized into groups of three that block off access to the parts of the corridor further to the right.

For each clause, there is one such group of doors, called a *clausal barrier*, their locks corresponding to the literals of the clause. To the right of these, there are similar structures for each variable, called *variable barriers*. The goal location is to their right, on the rightmost column.

We first show that if a plan of length at most $(2n + 2)W$ exists, there is a satisfying truth assignment for the 3SAT instance.

In order to gain access to the goal location, at least one of each doors of the clausal and variable barriers must be opened. For the variable barriers, this implies that for each variable v, at least one of the keys v and $\neg v$ must have been picked up and moved past the void. Additionally, the goal key must have been moved past the void (even further, but we do not care about that). As only one key can be moved at once, the W steps between the initial location and the clausal barrier must have been passed at least $2k - 1$ times if k keys were carried past the void, going from left to right k times and back $k - 1$ times.

As plan length is bounded by $(2n + 2)W$ and the movements considered so far take at least $(2k - 1)W$ actions, k cannot be greater than $n + 1$. As one of the keys must be the goal key, at most n other keys were used to open doors. We said that at least one key for each variable must be among them, which means that there is *exactly* one key for each variable.

We choose the truth assignment according to the keys selected, i. e. we set $v = \top$ if v was selected and $v = \bot$ if $\neg v$ was selected. We now check that each clause has at least one literal that is true under this assignment. To see this, consider a literal corresponding to an opened door on the barrier of that clause. Since the door was opened, the corresponding key was picked up, which by definition of the truth assignment implies that the corresponding literal is true. Thus, the instance is satisfiable.

Now assume there is a satisfying truth assignment for that clause. Let $P_S \subseteq P$ be the set of keys that correspond to the literals being true under this truth assignment. This means that $|P_S| = n$, and P_S contains at least one key for each clausal or variable barrier. We now have to check that, using no more than $(2n + 2)W$ actions, we can move the goal key to its goal location.

We will adopt the following strategy:

1. As long as there are barriers to be broken, do the following:
   (a) Randomly choose a key $p \in P_S$ that will open one of the doors of the leftmost barrier that has not been broken yet.
   (b) Move to the current location of p on a shortest path.
   (c) Pick up p.
   (d) Move to the location left to the door to be opened.
   (e) Unlock the door and drop the key.
2. Move to location $(0, 0)$.

3. Pick up the goal key.

4. Move to location $(w, 0)$.

5. Drop the goal key.

This will lead to exactly $n + m + 1$ *pick* actions, $n + m + 1$ *drop* actions and $n + m$ *unlock* actions, totaling $3(n + m) + 2$ non-movement actions.

To bound the number of *move* actions, we observe that the robot never moves to the right without carrying a key and never moves to the left while carrying a key, which means that it cannot move to the right more than $(n + 1)w$ times, as only $n + 1$ keys are being carried and none of them can be moved further to the right than w steps. As the robots final location is w units right of its initial location, it cannot move to the left more than $(n + 1)w - w = nw$ times, totalling $(2n + 1)w = (2n + 1)(W + 2m + 2n) = (2n + 1)W + (2n + 1)(2m + 2n)$ moves to the left or right.

Vertical moves are only necessary before passing barriers, for accessing the correct door to be opened and immediately before picking up a key or dropping the goal key. In each of these cases, no more than two moves upwards or downwards are needed.

There are $n+m$ barriers, and each of them must be passed no more than $2(n+m)+2$ times (at most twice on every iteration of the first step of the strategy above and at most once during each the second and fourth step). Doors are opened $n + m$ times, there are $n + m + 1$ occasions on which a key is picked up and one occasion on which the goal key is dropped.

This leads to an upper bound on the total number of vertical movements equal to $2((n + m)(2(n + m) + 2) + (n + m) + (n + m + 1) + 1) = 4(n + m)^2 + 8(n + m) + 4$

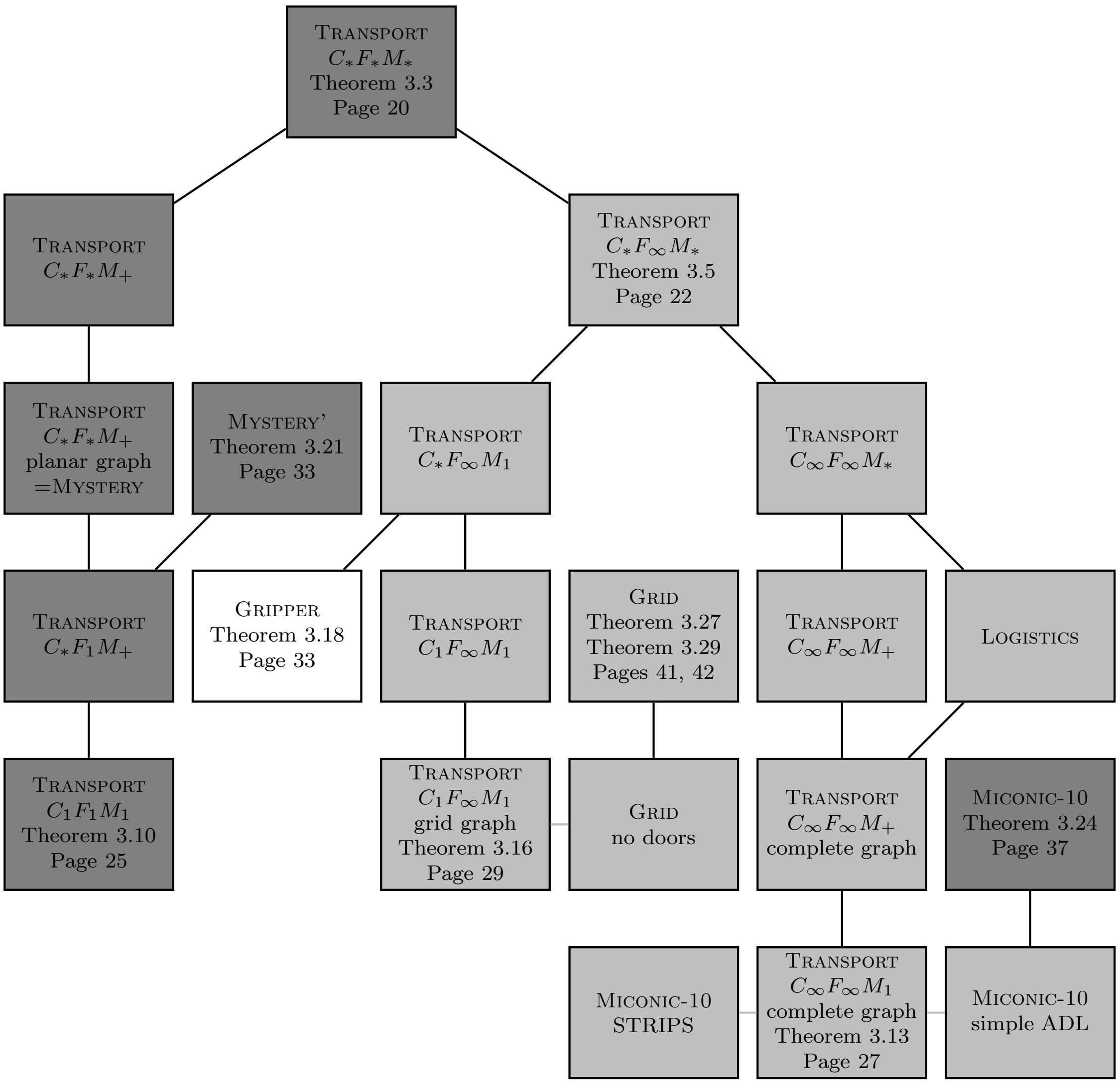
The total number of actions is thus no bigger than the sum of these three values, which is $3(n + m) + 2 + (2n + 1)W + (2n + 1)(2m + 2n) + 4(n + m)^2 + 8(n + m) + 4 = W + (2n + 1)W = (2n + 2)W$, as required. ■

3.10 Summary

We have discussed a variety of transportation domains, including the ones from the AIPS competitions. Figure 3.9 summarizes their generalization/specialization relationship and the most important results from this chapter.

We have seen that just finding some plan is not hard for most transportation domains. We think that this is mainly due to the fact that the "obvious" things to do, like moving portables nearer to their goal location or opening doors when this is possible, will always facilitate achieving the goal for most transportation domains. This means that forward planning and local search strategies can easily cope with those domains, as can also be seen from the good results a number of participating planning systems achieved in the AIPS 2000 competition on the LOGISTICS domain, employing such techniques.

The appropriateness of this greedy strategy of "moving portables towards the goal" changes, however, if additional constraints can lead to dead ends in the search space. In this case, finding a plan seems to be harder, and the theoretical results we have proved show that it really is. We have faced this problem when dealing with fuel constraints and in the full MICONIC-10 domain, where it may be unwise to have people board the elevator that restrict its movement too much. These domains, where goals can no longer



TRANSPORT $C_*F_*M_*$ Theorem 3.3 Page 20
TRANSPORT $C_*F_*M_+$
TRANSPORT $C_*F_\infty M_*$ Theorem 3.5 Page 22
TRANSPORT $C_*F_*M_+$ planar graph =MYSTERY
MYSTERY' Theorem 3.21 Page 33
TRANSPORT $C_*F_\infty M_1$
TRANSPORT $C_\infty F_\infty M_*$
TRANSPORT $C_*F_1M_+$
GRIPPER Theorem 3.18 Page 33
TRANSPORT $C_1F_\infty M_1$
GRID Theorem 3.27 Theorem 3.29 Pages 41, 42
TRANSPORT $C_\infty F_\infty M_+$
LOGISTICS
TRANSPORT $C_1F_1M_1$ Theorem 3.10 Page 25
TRANSPORT $C_1F_\infty M_1$ grid graph Theorem 3.16 Page 29
GRID no doors
TRANSPORT $C_\infty F_\infty M_+$ complete graph
MICONIC-10 Theorem 3.24 Page 37
MICONIC-10 STRIPS
TRANSPORT $C_\infty F_\infty M_1$ complete graph Theorem 3.13 Page 27
MICONIC-10 simple ADL

Figure 3.9: The transportation domains hierarchy. Domains further up in the graph are more general than their descendants. Additionally, TRANSPORT-$C_*F_\infty M_1$ is more general than TRANSPORT-$C_\infty F_\infty M_1$. A white box is used for domains for which the decision (and search) problems are polynomial. A light grey box indicates that plans can be generated in polynomial time, but the PLANLEN problem is **NP**-complete. For domains in dark grey boxes, both decision problems are **NP**-complete. Domains linked with light grey lines are almost identical (cf. Sections 3.8 and 3.9).

be achieved in a piecemeal fashion, are the only transportation domains we have analyzed for which plan existence is **NP**-complete.

Another important observation is that for all but the most trivial domains (i. e. all but GRIPPER), finding optimal plans is hard, the corresponding decision problems all being **NP**-complete. In most cases, our proofs exploited sub-goal interactions. It is difficult to decide in which order the portables need to be moved to come up with a short plan. This is still true under some very restrictive assumptions, like only having one mobile and moving on a complete graph.

Dynamic roadmaps as seen in the GRID domain seem to be another source of complexity, as outlined in the last proof. There is no subgoal interaction here since there is only one goal to be satisfied, and while one might argue that there are indeed interacting "intermediary goals", the difficulty lies in deciding *which* intermediary goals to pursue rather than choosing between different ways of scheduling them.

For convenience, Figure 3.10 repeats the complexity results for the transportation domains from the planning competitions.

Domain name	PLANEX	PLANLEN
GRID	polynomial	**NP**-complete
GRIPPER	polynomial	polynomial
LOGISTICS	polynomial	**NP**-complete
MICONIC-10 (simple)	polynomial	**NP**-complete
MICONIC-10 (full)	**NP**-complete	**NP**-complete
MYSTERY	**NP**-complete	**NP**-complete
MYSTERY'	**NP**-complete	**NP**-complete

Figure 3.10: Complexity results for the transportation domains from the AIPS 1998 and AIPS 2000 planning competitions.

Chapter 4

Manipulation Domains

In this chapter, we analyze planning domains that focus on the manipulation of physical objects[1], namely ASSEMBLY, BLOCKSWORLD, FREECELL and SCHEDULE. What do these have in common? In all of them except SCHEDULE, atomic parts (atomic assemblies, individual blocks, individual cards) are combined to form bigger things (assemblies, towers of blocks, piles of cards), which can then in turn be used as parts for further composite objects. The goal in these domains is to build up or construct one or several complex composite objects, using operators which attach parts to or detach parts from composites built already.

SCHEDULE, on the other hand, does not feature construction, but is mainly concerned with the transformation of physical objects by operations such as painting them or punching holes into them. A natural combination of these two different aspects of manipulation domains would be a variant of the BLOCKSWORLD domain where blocks or towers could be painted as well as moved around.

In addition to the actual composition, decomposition and transformation, different kinds of resource allocation problems typically arise in manipulation domains, one of them being restricted space (as in FREECELL), meaning that only so many objects may coexist at the same time, the other one being allocation of tools or machines (such as a screwdriver or spray painter) to objects, as in ASSEMBLY and SCHEDULE.

So there are three key features to a manipulation domain:

- First, *construction* and *destruction* can take place, combining smaller objects to build bigger objects or decomposing composite structures into smaller parts. The space available for doing the composition and decomposition might be limited.
- Secondly, *transformations* such as painting change physical properties of individual objects without affecting their subpart structure.
- Third, the allocation of *equipment* to perform the various construction, destruction and transformation tasks is an important aspect.

The difficulty of solving a manipulation-type problem can consist of finding the right order in which different goal composites should be built up (as in BLOCKSWORLD), in constraints for the build-up operators which might require intermediary objects to be

[1]We admit that this is a fairly general description, but then the domains to be presented cover fairly general problems, as the results will show.

built up and demolished again (as in Assembly and FreeCell), in restricted space (as in FreeCell) or in the scheduling of available equipment (as in Schedule).

Despite some obvious and some less obvious similarities in these domains, it is not apparent how they might be incorporated into a structured domain hierarchy like it has been done for the Transport family. The group of domains discussed in this chapter is far less homogeneous than the transportation family, and it is not easy to come up with "the" general manipulation domain.

Manipulation domains can differ in at least the following features:

- They can involve actions for attaching parts to other parts, actions for detaching parts from other parts, both or neither. On a more fine-grained level, this distinction can be drawn for individual composites. As an example, consider the foundation (home) piles in FreeCell, which can only be composed, but not decomposed.

- If both attaching and detaching are possible, they can be separate actions or part of a composite action. Especially in the first case, where detaching things and "putting them in the hand" is an action in its own right, it makes a difference whether there are one or several hands, and if the latter, if all of them can hold all types of objects or if there are certain restrictions.

- The composite objects in the domain can be made up from an arbitrary number of immediate subparts (as in Assembly) or a restricted number of them. An interesting special case are *stack* objects, consisting of two immediate subparts, a substack and a topmost object, which must be atomic (does not have any subparts), as in Blocksworld and FreeCell. In the most restricted case, all objects are atomic.

- Many possibilities exist for specifying the way things are constructed. It might be necessary to attach subparts in a given linear order, the sequence of attach operators can be required to adhere to some partial order (as in Assembly), or all sequences can be allowed. In some domains, composites can be built up or destroyed while they are part of other composites, in other domains this is disallowed. There might be one or multiple ways of building the same composite, possibly involving very different subparts, as in a chemistry domain where the same molecule can be created by various chemical reactions.

- The domain may or may not include *transformation* or *mutation* actions which do not rearrange the parts of an object but change some of its properties. As has been pointed out, these are the predominant actions in the Schedule domain.

- The domain may or may not make use of *equipment*, resource objects which need to be allocated to an object before parts can be attached to it or detached from it. If equipment is present, it can be be a separate class of objects (like the resources in Assembly and the machines in Schedule), or arbitrary objects can be used as equipment, which implies that the equipment itself might need to be built from other parts first.

- Last not least, the type of goals specified makes a difference. In the easiest case, there is just one goal state, completely specifying the object(s) to be built (as in

Assembly or FreeCell). In other domains, the goal objects might not comprise all atomic parts, so that there are leftover parts that are not cared about (as is the case for Blocksworld with "irrelevant" blocks). In the most general case, even the objects that need to be created are not completely specified, with goals of the type "build some tower of height eight". This kind of goal occurs with Schedule.

Rather than investigating all these possibilities in their full variety, we focus on a subhierarchy of manipulation domains that is reasonably general and covers all the benchmark domains. We want to emphasize that what is to come is not to be seen as "the" manipulation domain – variants of the general theme that are not closely related to the domain presented in the following section are easily imaginable.

Some of the more important restricting assumptions we make are summarized here:

- Attaching and detaching things are separate actions. It is not possible to move parts from one composite to another one using only one action, this kind of movement always passes through a *hand*. There is only one hand, although domains involving multiple hands can easily be reformulated in this way. An example of this is the FreeCell domain, featuring multiple hand-like free cells. The reformulation can (and will) lead to changes to plan length for domains that allow for direct movement between composites, and this would need to be taken into account when discussing PlanLen.

- Composites can only be decomposed into the same parts they were constructed from. It is still possible to allow for chemistry-like domains by making use of transformation operations, however. The subparts of a composite must be attached and can only be detached obeying a strict linear order, which is a bit restrictive, but sufficient for all domains that we want to cover. Given that subparts need to be attached in a linear fashion, no additional generality is lost by restricting composites to consist of two immediate subparts (the *base* part and the *added* part)[2], and we will do this.

- Transformations must always maintain the subpart hierarchy of an object, i. e. they cannot add or remove subparts or reorder them. However, they can recursively transform their subparts (not only immediate ones) to perform an operation like "paint a whole stack of blocks". This is not too restrictive, as even this restricted kind of transformation has a strong impact on the complexity of the domain, as we shall see.

- Equipment is only needed for two purposes: Attaching and detaching operations might require some specific piece of equipment (like a hammer) for incorporating or removing given parts, and transformations might require equipment to be applicable. Attaching and detaching will always be inverse to each other and require the same equipment for the same subpart. There is no way of specifying that equipment needs to be allocated to an object throughout the whole construction process of a complex structure. It will always be possible to deallocate and reallocate it in between.

[2]A linearly ordered composite consisting of n immediate subparts can easily be remodelled as a subcomposite with $n-1$ immediate subparts and an added part. This transformation can be applied recursively until no composites consist of more than two immediate subparts, introducing only polynomially many intermediary structures.

- The goal objects will always be completely specified objects. This is a severe restriction and requires a little work for integrating domains like SCHEDULE, where this is not the case, into the family. If this restriction were dropped, one could easily imagine problems for which checking if a given state is a goal state is an **NP**-hard problem in itself, involving SUBFOREST ISOMORPHISM-type decision problems.

4.1 MANIPULATE

Like we did for the transportation domains, we will first introduce a fairly general manipulation domain and then talk about special cases of it. Because of the greater differences between domains, this will involve more work than for TRANSPORT. To start with, we introduce the notion of an *instance signature.* The signature describes the physics of the instance, e. g. what types of objects can be attached to what types of objects, without specifying the actual initial state and goal. For many manipulation domains (like BLOCKSWORLD), this part does not vary or varies only slightly between instances.

Definition 4.1 MANIPULATE **instance signature**
A MANIPULATE **instance signature** *is a seven-tuple* $\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}})$, *where*

- B *is the finite set of* **blueprints**,
- $B_C \subseteq B$ *is the set of* **composable** *blueprints,*
- $B_D \subseteq B$ *is the set of* **decomposable** *blueprints,*
- $B^{\mathcal{I}} : B \nrightarrow \mathbb{P}(B) \times \mathbb{P}(B) \times \mathbb{P}(B)$ *is the* **blueprint interpretation** *function, satisfying* $B_C \cup B_D \subseteq \operatorname{dom}(B^{\mathcal{I}})$. *The domain of this function is referred to as the set of* **composite blueprints** B_{Comp}, $B \setminus B_{Comp}$ *as the set of* **atomic blueprints** B_{Atom}.

  We write $b^{\mathcal{I}}$ *as a shorthand for* $B^{\mathcal{I}}(b)$. *For* $b^{\mathcal{I}} = (B_1, B_2, B_E)$, *the blueprints from* B_1 *are called* **allowed base parts** *of* b, *the ones from* B_2 *are called* **allowed added parts** *of* b *and the ones from* B_E *form the* **required equipment** *of* b.
- T *is the finite set of* **transformations**,
- $T_A \subseteq T$ *is the set of* **applicable transformations** *and finally*
- $T^{\mathcal{I}} : T \to B \times B \times \mathbb{P}(B) \times (B \nrightarrow T) \times (B \nrightarrow T)$ *is the* **transformation interpretation** *function. We write* $t^{\mathcal{I}}$ *as a shorthand for* $T^{\mathcal{I}}(t)$. *For* $t^{\mathcal{I}} = (b, b', B_E, t_1, t_2)$, b *is called the* **previous blueprint**, b' *is called the* **effect blueprint**, B_E *is called the* **transformation equipment** *and* t_1 *and* t_2 *are called the implicit* **subtransformation** *specifications of* t. *It must be true that* b *is an atomic blueprint if and only if* b' *is an atomic blueprint. In this case,* t_1 *and* t_2 *must be empty sets.*

  Otherwise, let $b^{\mathcal{I}} = (B_1, B_2, B_E)$ *and* $b'^{\mathcal{I}} = (B'_1, B'_2, B'_E)$.
  Then for all $i \in \{1, 2\}$, *it must be true that* $\operatorname{dom}(t_i) = B_i$, *and for all* $b_i \in B_i$,
  $T^{\mathcal{I}}(t_i(b_i)) \in \{b_i\} \times B'_i \times \{ \ B^*_E \mid B^*_E \subseteq B_E \ \} \times (B \nrightarrow T) \times (B \nrightarrow T)$.

The encoding length of the signature is assumed to be $|B| + |T|$.

It is not hard to verify that the specified encoding length is indeed polynomially equivalent to the length of any reasonable encoding of the information conveyed by a Manipulate instance signature.

A *blueprint* should be seen as a recipe for building things: In the FreeCell domain, a typical composite blueprint would be a *black 7 tableau pile*, with a *red 8 tableau pile* as its only allowed base part and two allowed added parts, atomic blueprints of type ♠7 *card* or ♣7 *card*. Decomposition is inverse to composition.

No equipment is required for that blueprint, but the required equipment set can be used in general for specifying that objects of certain types (such as a hammer) must be allocated to the base part as equipment before the added part can be attached to it. For each blueprint in the equipment set, one corresponding object must be committed to the task. The equipment requirements apply to composition as well as decomposition.

Transformations transform an object of one type specified by the previous blueprint, to an object of another type, specified by the effect blueprint, requiring that objects of certain types are committed to the transformed object, specified by the transformation equipment set.

What happens to the subparts of the object when a given transformation is applied is defined by its implicit subtransformation specifications t_1 and t_2: If the base part of the target object is of type b_1, then that object is recursively transformed by the transformation $t_1(b_1)$. The constraints on t_1 ensure that this subtransformation is defined to work on that blueprint and creates something of an appropriate type. Subtransformations may only have equipment requirements that are already part of the original transformation, so they do not add equipment requirements. Similarly, t_2 describes what happens to the added part.

Some transformations might only be used as subtransformations and cannot be explicitly invoked as an action in a plan. These transformations will then not be included in the set of applicable transformations.

We have informally mentioned the term *objects* a number of times already. Having defined instance signatures, we can now formally define what we understand by it. Because objects can contain other objects as subparts, this is a recursive definition.

Definition 4.2 Object
Given a Manipulate *instance signature* $\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}})$,
the set of **objects** O_Σ,
the set of **atomic objects** *or* **atoms** $Atom_\Sigma$,
the set of **composite objects** $Comp_\Sigma$,
the **object type function** $\tau_\Sigma : O_\Sigma \to B$ *and*
the **object size function** $size_\Sigma : O_\Sigma \to \mathbb{N}_0$ *are defined as follows:*

1. *For atomic blueprints* $b \in B_{Atom}$*, we define:*

$$\begin{aligned} b &\in O_\Sigma \\ b &\in Atom_\Sigma \\ \tau_\Sigma(b) &\overset{\text{def}}{=} b \\ size_\Sigma(b) &\overset{\text{def}}{=} 1 \end{aligned}$$

2. *For composite blueprints* $b \in B_{Comp}$ *with* $b^{\mathcal{I}} = (B_1, B_2, B_E)$ *and* $p_1, p_2 \in O_\Sigma$ *satisfying* $\tau_\Sigma(p_1) \in B_1$ *and* $\tau_\Sigma(p_2) \in B_2$*, we define:*

$$\begin{aligned} o \overset{\text{def}}{=} (b, p_1, p_2) &\in O_\Sigma \\ o &\in \mathit{Comp}_\Sigma \\ \tau_\Sigma(o) &\overset{\text{def}}{=} b \\ \mathit{size}_\Sigma(o) &\overset{\text{def}}{=} \mathit{size}_\Sigma(p_1) + \mathit{size}_\Sigma(p_2) \end{aligned}$$

3. *O_Σ, Atom_Σ and Comp_Σ do not contain any other elements except those whose membership follows from the above rules.*

Objects can be manipulated in three basic ways: Parts can be attached, parts can be detached, and the whole object can be transformed.

Definition 4.3 Operations on Objects
Given a Manipulate *instance signature* $\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}})$,

$$\begin{aligned} \mathit{attach}_\Sigma : &\; B_C \times O_\Sigma \times O_\Sigma \times \mathbb{P}(B) \nrightarrow O_\Sigma, \\ \mathit{detach}_\Sigma : &\; O_\Sigma \times \mathbb{P}(B) \nrightarrow O_\Sigma \times O_\Sigma \text{ and} \\ \mathit{transform}_\Sigma : &\; O_\Sigma \times T \times \mathbb{P}(B) \nrightarrow O_\Sigma \end{aligned}$$

are defined as follows:

$\forall b \in B_C, p_1, p_2 \in O_\Sigma, \mathit{Equip}, B_1, B_2, B_E \subseteq B :$
$\quad b^{\mathcal{I}} = (B_1, B_2, B_E)$
$\wedge\ \tau_\Sigma(p_1) \in B_1$
$\wedge\ \tau_\Sigma(p_2) \in B_2$
$\wedge\ B_E \subseteq \mathit{Equip}$
$\Rightarrow \mathit{attach}_\Sigma(b, p_1, p_2, \mathit{Equip}) = (b, p_1, p_2)$

$\forall o = (b, p_1, p_2) \in \mathit{Comp}_\Sigma, \mathit{Equip}, B_1, B_2, B_E \subseteq B$
$\quad b \in B_D$
$\wedge\ b^{\mathcal{I}} = (B_1, B_2, B_E)$
$\wedge\ B_E \subseteq \mathit{Equip}$
$\Rightarrow \mathit{detach}_\Sigma(o, \mathit{Equip}) = (p_1, p_2)$

$\forall o, o' \in \mathit{Atom}_\Sigma, t \in T, \mathit{Equip}, B_E \subseteq B :$
$\quad t^{\mathcal{I}} = (o, o', B_E, \emptyset, \emptyset)$
$\wedge\ B_E \subseteq \mathit{Equip}$
$\Rightarrow \mathit{transform}_\Sigma(o, t, \mathit{Equip}) = o'$

$\forall o = (b, p_1, p_2), o' = (b', p_1', p_2') \in \mathit{Comp}_\Sigma, t \in T, \mathit{Equip}, B_E \subseteq B, t_1, t_2 \in B \nrightarrow T :$
$\quad t^{\mathcal{I}} = (b, b', B_E, t_1, t_2)$
$\wedge\ B_E \subseteq \mathit{Equip}$
$\wedge\ p_1' = \mathit{transform}_\Sigma(p_1, t_1(\tau_\Sigma(p_1)), \mathit{Equip})$
$\wedge\ p_2' = \mathit{transform}_\Sigma(p_2, t_2(\tau_\Sigma(p_2)), \mathit{Equip})$
$\Rightarrow \mathit{transform}_\Sigma(o, t, \mathit{Equip}) = o'$

attach_Σ, detach_Σ and $\mathit{transform}_\Sigma$ are undefined otherwise

This definition formalizes the operations and constraints that have been mentioned informally before. The arguments to attach_Σ are, in sequence: the blueprint to be created, the object that is being worked on, the object that is being attached to it and the types of all equipment that has been committed to the task. For detach_Σ, they are the target object and the types of available equipment. For trans_Σ, they are the object to be transformed, the transformation and the types of available equipment.

Now that objects have been defined, we can define instances of the MANIPULATE domain.

Definition 4.4 MANIPULATE **instance**
A MANIPULATE **instance** *is a four-tuple* $I = (\Sigma, Pos, init, goal)$*, where*

- Σ *is a* MANIPULATE *instance signature of encoding length* M*,*
- Pos *is a finite set of* **table positions***,*
- $init : Pos \nrightarrow O_\Sigma$ *is the* **initial state specification** *and finally*
- $goal : Pos \nrightarrow O_\Sigma$ *is the* **goal specification***, satisfying* $N = \sum_{o \in \mathrm{ran}(init)} size_\Sigma(o) = \sum_{o \in \mathrm{ran}(goal)} size_\Sigma(o)$*.*

The encoding length of the instance is assumed to be $M + N$*.*

It is assumed that the initial state specification maps table positions to the objects that are initially located there. It is a partial function to take empty table positions into account. Although the goal state specification is also defined as a function of table positions rather than just a set of objects, this is only to allow for multiple identical objects in the goal state. For reaching a goal state, it does not matter on *which* table position an object is located as long as it is present. The number of table positions does not need to be taken into account for the encoding length because it can be bounded by N without loss of generality.

We can now define the MANIPULATE domain.

Definition 4.5 MANIPULATE **domain**
Let L *be the set of all* MANIPULATE *instances. The* MANIPULATE *domain is the function* $\mathcal{D} : L \to \mathcal{M}$ *mapping instances to state models as follows:*

For $\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}})$*,*
$(\Sigma, Pos, init, goal) \mapsto (S, s_0, S_G, A, \delta)$*, where*

$$\begin{aligned}
S &\overset{\text{def}}{=} (O_\Sigma \cup \{\bot\}) \times (Pos \nrightarrow O_\Sigma) \times (Pos \nrightarrow Pos),\\
s_0 &\overset{\text{def}}{=} (\bot, init, \emptyset),\\
S_G &\overset{\text{def}}{=} \{(\bot, table, alloc) \in S \mid \exists \sigma \in \mathrm{Sym}(Pos) : table = goal \circ \sigma\ \},\\
A &\overset{\text{def}}{=} \{\ pick_{pos} \mid pos \in Pos\ \} \cup \{\ drop_{pos} \mid pos \in Pos\ \}\\
&\quad \cup \{\ decompose_{pos} \mid pos \in Pos\ \} \cup \{\ compose_{pos,b} \mid pos \in Pos, b \in B_C\ \}\\
&\quad \cup \{\ commit_{pos,pos'} \mid pos, pos' \in Pos\ \} \cup \{\ release_{pos} \mid pos \in Pos\ \}\\
&\quad \cup \{\ transform_{pos,t} \mid pos \in Pos, t \in T\ \},
\end{aligned}$$

$$\begin{aligned}
&\forall pos \in Pos, s = (hand, table, alloc) \in S :\\
&\qquad hand = \bot\\
&\quad \wedge\ pos \in \mathrm{dom}(table) \setminus (\mathrm{dom}(alloc) \cup \mathrm{ran}(alloc))\\
&\quad \Rightarrow \delta(s, pick_{pos}) \overset{\text{def}}{=} (table(pos), table \setminus \{(pos, table(pos))\}, alloc),\\
&\forall pos \in Pos, s = (hand, table, alloc) \in S :\\
&\qquad hand \neq \bot\\
&\quad \wedge\ pos \notin \mathrm{dom}(table)\\
&\quad \Rightarrow \delta(s, drop_{pos}) \overset{\text{def}}{=} (\bot, table \cup \{(pos, hand)\}, alloc),
\end{aligned}$$

$$
\begin{aligned}
&\forall pos \in Pos, s = (hand, table, alloc) \in S, Equip \subseteq B, o', hand' \in O_\Sigma :\\
&\qquad hand = \bot\\
&\quad \wedge\ pos \in \mathrm{dom}(table) \setminus \mathrm{dom}(alloc)\\
&\quad \wedge\ Equip = \{\ \tau_\Sigma(table(p)) \mid p \in alloc^{-1}(\{pos\})\ \}\\
&\quad \wedge\ detach_\Sigma(table(pos), Equip) = (o', hand')\\
&\quad \Rightarrow \delta(s, decompose_{pos}) \overset{\text{def}}{=} (hand', table \oplus \{(pos, o')\}, alloc),\\
&\forall pos \in Pos, b \in B_C, s = (hand, table, alloc) \in S, Equip \subseteq B, o' \in O_\Sigma :\\
&\qquad hand \neq \bot\\
&\quad \wedge\ pos \in \mathrm{dom}(table) \setminus \mathrm{dom}(alloc)\\
&\quad \wedge\ Equip = \{\ \tau_\Sigma(table(p)) \mid p \in alloc^{-1}(\{pos\})\ \}\\
&\quad \wedge\ attach_\Sigma(b, table(pos), hand, Equip) = o'\\
&\quad \Rightarrow \delta(s, compose_{pos,b}) \overset{\text{def}}{=} (\bot, table \oplus \{(pos, o')\}, alloc),\\
&\forall pos, pos' \in Pos, s = (hand, table, alloc) \in S :\\
&\qquad pos \neq pos'\\
&\quad \wedge\ pos \in \mathrm{dom}(table) \setminus \mathrm{dom}(alloc)\\
&\quad \wedge\ pos' \in \mathrm{dom}(table)\\
&\quad \Rightarrow \delta(s, commit_{pos,pos'}) \overset{\text{def}}{=} (hand, table, alloc \cup \{(pos, pos')\}),\\
&\forall pos \in Pos, s = (hand, table, alloc) \in S :\\
&\qquad pos \in \mathrm{dom}(alloc)\\
&\quad \Rightarrow \delta(s, release_{pos}) \overset{\text{def}}{=} (hand, table, alloc \setminus \{(pos, alloc(pos))\}),\\
&\forall pos \in Pos, t \in T_A, s = (hand, table, alloc) \in S, Equip \subseteq B, o' \in O_\Sigma :\\
&\qquad pos \in \mathrm{dom}(table) \setminus \mathrm{dom}(alloc)\\
&\quad \wedge\ Equip = \{\ \tau_\Sigma(table(p)) \mid p \in alloc^{-1}(\{pos\})\ \}\\
&\quad \wedge\ transform_\Sigma(table(pos), t, Equip) = o'\\
&\quad \Rightarrow \delta(s, transform_{pos,t}) \overset{\text{def}}{=} (hand, table \oplus \{(pos, o')\}, alloc)\ \textit{and finally}\\
&\delta \textit{ is undefined otherwise.}
\end{aligned}
$$

A state in the MANIPULATE domain consists of three parts. The first part specifies the object in hand (or $\bot$, if no object is in hand). The second part, usually called *table*, maps table positions to the objects located there. This is a partial function to allow for empty table positions. The third part, usually called *alloc* for *equipment allocation*, maps table positions to other table positions, where $alloc(pos_1) = pos_2$ means that the object at the first position can be used as equipment at the second table position.

In the initial state, the hand is empty, the state of the table follows the initial state specification, and there are no equipment allocations.

In a goal state, the only requirement is that there is some permutation of the objects on the table that matches the goal specification. This can only be possible if the hand is empty, since *init* and *goal* contain the same number of atomic objects as subparts by definition.

The *pick* and *drop* operators move objects between the table and the hand. An object cannot be picked up if it is currently being used as equipment or has equipment allocated to it.

The *decompose* and *compose* operators detach a subpart from an object at a given table position and put in in hand or incorporate the object in hand into the object at a given table position. Objects that are currently used as equipment cannot be worked on, and in order to be able to work on an object, type and equipment constraints must be met that are checked by $detach_\Sigma$ and $attach_\Sigma$.

To allocate or deallocate equipment, the *commit* and *release* operators can be used. Objects can only be committed to one table position at the same time.

Finally, $transform$ transforms an object at a given table position using a given transformation, provided the necessary equipment has been made available and the target object is not currently being used as equipment.

4.2 Special Cases of MANIPULATE

We will now define special cases of MANIPULATE that we consider particularly interesting. We will distinguish four orthogonal features, totalling 36 variants of the original domain. They are named MANIPULATE-$S_iT_jE_kP_l$, where $i, k \in \{0, 1, 2\}$ and $j, l \in \{0, 1\}$ are the parameters for the different features of the domain. For all four parameters it is true that higher values define more general domains. MANIPULATE-$S_2T_1E_2P_1$ is just another name for MANIPULATE.

Definition 4.6 Special cases of MANIPULATE
For $i, k \in \{0, 1, 2\}$ and $j, l \in \{0, 1\}$, the MANIPULATE-$S_iT_jE_kP_l$ *domain (subparts – transformations – equipment – (table) positions) is defined as the restriction of the* MANIPULATE *domain to those instances $I = (\Sigma, Pos, init, goal)$, using an instance signature $\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}})$, that adhere to the following constraints:*

- *For $i = 0$, all blueprints are atomic, i. e. $B = B_{Atom}$. There is no construction or destruction going on in these domains.*
- *For $i = 1$, additionally composite blueprints that have sets of atomic blueprints as their allowed added parts are possible. This allows building "stacks" of atomic objects.*
- *For $i = 2$, there is no restriction on the subparts of a blueprint.*
- *For $j = 0$, $T = \emptyset$. There are no transformations in these domains.*
- *For $j = 1$, there are no restrictions on T.*
- *The set of* **equipment blueprints** *of Σ is defined as:*
  $B_{Equip} \overset{\text{def}}{=} \bigcup_{(B_1, B_2, B_E) \in \operatorname{ran}(B^{\mathcal{I}})} B_E \cup \bigcup_{(b, b', B_E, t_1, t_2) \in \operatorname{ran}(T^{\mathcal{I}})} B_E$.
  For $k = 0$, $B_{Equip} = \emptyset$. No equipment is required in these domains.
- *For $k = 1$, B_{Equip} only consists of atomic blueprints, and no blueprint in B_{Equip} appears as an allowed subpart in $B^{\mathcal{I}}$ or as the previous or effect blueprint of any transformation. Objects used as equipment in these domains are "separate" from other kinds of objects: they are always atomic, can never be incorporated into composites, and cannot be created or changed by a transformation.*
- *For $k = 2$, there is no restriction on B_{Equip}.*

- *For $l = 0$, $|Pos| = \sum_{o \in \mathrm{ran}(init)} size_\Sigma(o)$. There are enough table positions for holding all atomic objects in these domains, and hence the number of table positions can be regarded as unlimited.*

- *For $l = 1$, $|Pos|$ can be any natural number.*

As is easy to check from these definitions, the generalization-specialization relationship between these domains is straight-forward: MANIPULATE-$S_iT_jE_kP_l$ is a specialization of MANIPULATE-$S_{i'}T_{j'}E_{k'}P_{l'}$ if and only if $i \leq i'$, $j \leq j'$, $k \leq k'$, and $l \leq l'$.

Figure 4.1 summarizes these definitions.

subparts	**0**: no subparts **1**: one composite, one atom **2**: no restrictions
transformations	**0**: no **1**: yes
equipment	**0**: none **1**: separate blueprints **2**: no restrictions
table positions	**0**: unlimited **1**: any number

Figure 4.1: The MANIPULATE domain family.

There are two subfamilies in this hierarchy that are sufficiently important to warrant their own name, namely the ones without transformations and the ones without composite blueprints.

Domains without transformations are called *construction* domains, and for $i \in \{1, 2\}$[3], $k \in \{0, 1, 2\}$ and $l \in \{0, 1\}$, CONSTRUCT-$S_iE_kP_l$ is defined to be another name for MANIPULATE-$S_iT_0E_kP_l$. Domains without composite blueprints but with transformations are called *transformation* domains, and for $k \in \{0, 1, 2\}$, TRANSFORM-E_k is defined to be another name for MANIPULATE-$S_0T_1E_kP_0$.[4]

The CONSTRUCT and TRANSFORM domains contain disjoint sets of domains, and all non-trivial domains that fall into neither family must feature composite blueprints *and* transformations and thus be generalizations of MANIPULATE-$S_1T_1E_0P_0$. We will first show that PLANEX (and thus also PLANLEN) for this domain is **PSPACE**-hard, even if all composites consist of two *atomic* parts only.

After that, we will prove **PSPACE**-completeness for all domains outside the construction and transformation subfamilies.

For the first proof, we must first introduce deterministic Turing Machines.

Definition 4.7 Deterministic Turing Machine
A **deterministic Turing Machine (DTM)** *is an eight-tuple*
$M = (Q, q_0, q_Y, q_N, \Sigma, \Gamma, \#, \delta)$*, where*

[3]There is no point in allowing $i = 0$, because no interesting activity can take place in a domain without transformations if there are no composite blueprints.

[4]There is no point in allowing to limit the number of table position as in these domains there is no reason to use the hand at all. There is no difference between P_0 and P_1 here.

- *Q is the finite set of* **states***, containing (along with other states) a* **start state** *q_0, an* **accepting halt state** *q_Y and a* **rejecting halt state** *q_N,*
- *Σ is the finite* **input alphabet***,*
- *$\Gamma \supseteq \Sigma$ is the finite* **tape alphabet***, containing a* **blank symbol** *$\#$ and finally*
- *$\delta : (Q \setminus \{q_Y, q_N\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ is the* **transition function**.

Our definition is the same as in [Garey and Johnson, 1979], except that we write the blank symbol differently to avoid confusion with blueprints. Also see this reference for a definition of the LINEAR SPACE ACCEPTANCE problem used in the following proof and pointers to the well-known proof of **PSPACE**-completeness for this problem.

Theorem 4.8 PSPACE-hardness of PLANEX-MANIPULATE-$S_1T_1E_0P_0$
PLANEX-MANIPULATE-$S_1T_1E_0P_0$ *is* **PSPACE***-hard, even if composite blueprints solely consist of atomic parts.*
Proof: We reduce the LINEAR SPACE ACCEPTANCE problem to the problem at hand.

Let $M = (Q, q_0, q_Y, q_N, \Sigma_T, \Gamma, \#, \delta)$ be a DTM and $w = w_1 \dots w_n$ for $n \in \mathbb{N}_0$ be the input to M. Without loss of generality, we can assume that the machine will only halt when all tape cells contain the blank symbol and the current tape position is the first position. It is not hard to modify M so that these requirements are met.

We define $\delta_{-1} = \delta \cap ((Q \times \Gamma) \times (Q \times \Gamma \times \{-1\}))$ and $\delta_{+1} = \delta \cap ((Q \times \Gamma) \times (Q \times \Gamma \times \{+1\}))$.

The corresponding PLANEX-MANIPULATE-$S_1T_1E_0P_0$ instance I is defined as
$\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}}), I = (\Sigma, \mathit{Pos}, \mathit{init}, \mathit{goal})$, where

$$
\begin{aligned}
B_{\mathit{Cell}} &\overset{\text{def}}{=} \{1, \dots, n+1\} \times \Gamma \\
B_{\mathit{State}} &\overset{\text{def}}{=} \{1, \dots, n+1\} \times \Gamma \times Q \\
B_{\mathit{Comp}} &\overset{\text{def}}{=} B_{\mathit{State}} \times B_{\mathit{Cell}} \\
B &\overset{\text{def}}{=} B_{\mathit{Cell}} \cup B_{\mathit{State}} \cup B_{\mathit{Comp}} \\
B_C \overset{\text{def}}{=} B_D &\overset{\text{def}}{=} B_{\mathit{Comp}} \\
(A, B)^{\mathcal{I}} &\overset{\text{def}}{=} (\{A\}, \{B\}, \emptyset) \text{ for } (A, B) \in B_{\mathit{Comp}} \\
T_A &\overset{\text{def}}{=} (\{2, \dots, n+1\} \times \delta_{-1} \times \Gamma) \cup (\{1, \dots, n\} \times \delta_{+1} \times \Gamma) \\
T &\overset{\text{def}}{=} T_A \cup (\{P_1, P_2\} \times T_A) \\
\forall \mathit{trans} &= ((q, a), (q', a', d)) \in \delta, t = (i, \mathit{trans}, b) \in T_A : \\
t^{\mathcal{I}} &\overset{\text{def}}{=} ((\mathit{old}_1, \mathit{old}_2), (\mathit{new}_1, \mathit{new}_2), \emptyset, \{(\mathit{old}_1, (P_1, t))\}, \{(\mathit{old}_2, (P_2, t))\}) \\
(P_1, t)^{\mathcal{I}} &\overset{\text{def}}{=} (\mathit{old}_1, \mathit{new}_1, \emptyset, \emptyset, \emptyset) \\
(P_2, t)^{\mathcal{I}} &\overset{\text{def}}{=} (\mathit{old}_2, \mathit{new}_2, \emptyset, \emptyset, \emptyset) \\
&\text{where } \mathit{old}_1 \overset{\text{def}}{=} (i, a, q), \mathit{old}_2 \overset{\text{def}}{=} (i+d, b), \mathit{new}_1 \overset{\text{def}}{=} (i+d, b, q'), \mathit{new}_2 \overset{\text{def}}{=} (i, a') \\
\mathit{Pos} &\overset{\text{def}}{=} \{1, \dots, n+1\} \\
\mathit{init} &\overset{\text{def}}{=} \{(1, (1, w_1, q_0))\} \cup \{\ (i, (i, w_i)) \mid i \in \{2, \dots, n\}\ \} \cup \{(n+1, (n+1, \#))\} \\
\mathit{goal} &\overset{\text{def}}{=} \{(1, (1, \#, q_Y))\} \cup \{\ (i, (i, \#)) \mid i \in \{2, \dots, n+1\}\ \}
\end{aligned}
$$

First note that the size of the resulting instance is polynomial in $|Q| + |\Gamma| + n$ and thus polynomial in the size of the input, and that it can be written down in polynomial time. It remains to prove that the resulting MANIPULATE-$S_1T_1E_0P_0$ instance is solvable if and only if there is an accepting computation for the original Turing Machine that only visits tape cells between 1 and $n + 1$. We will show this by pointing out a close relationship between transformations in the planning instance and state transitions in the Turing Machine that do not leave these tape cells.

There are $n + 1$ table positions in the generated instance. Initially, table position i corresponds to tape cell i. Although this strict correspondence may change after some actions are performed, it helps to identify table positions with tape cells.

There are two sets of atomic blueprints, B_{Cell}, called *cell blueprints*, containing elements like $(3, x)$ to signify that tape cell 3 contains character x, and B_{State}, called *state blueprints*, containing elements like $(3, x, q_7)$ to signify that tape cell 3 contains character x, is the current position of the read/write head, and the machine is in state q_7.

Any state blueprint can be combined with any cell blueprint to form a composite blueprint from B_{Comp}. These are composable and decomposable, and no equipment is required. The definitions of *Pos*, *init* and *goal* are straight-forward.

The definition of T and its interpretation function is most interesting. We will only explain transformations that correspond to transitions that move the read/write head to the left. The other transformations are modelled correspondingly.

Such an element $t \in T_A$ is characterized by three parts: A number reflecting the current tape cell, the transition from δ_{-1} it corresponds to and a character from the tape alphabet, which reflects the current contents of the tape cell that the read/write head moves to as an effect of the transition. For each such t, there are transformations (P_1, t) and (P_2, t) that cannot be invoked explicitly and are used as implicit subtransfomations of t (for the base and added part, respectively).

In order to apply a transformation relating to tape cell i, a Turing Machine transition *trans* and a neighbouring character b, a composite must be formed out of a state blueprint for tape cell i, matching state and tape character of the transition, and a cell blueprint for tape cell $i - 1$, holding character b. After applying the transformation (including the subtransformations), the state blueprint now relates to tape cell $i - 1$, b and the new state (because the read/write head has moved and the state has changed), and the cell blueprint relates to tape cell i, now holding the character written by the transition. The resulting composite can now be taken apart again, and the next transition can be simulated.

Thus, if there is an accepting computation for the Turing Machine that only visits tape cells between 1 and $n + 1$, then there is a plan for the MANIPULATE-$S_1T_1E_0P_0$ instance. To see that the converse is also true, note that the only way to make progress towards the goal is by applying transformations, which always correspond to a transition of the Turing Machine. ■

Although the preceding theorem did not discuss plan lengths, it is not hard to come up with finite tape Turing Machines that have accepting paths, but require exponentially many steps to halt. For these, plans for the corresponding MANIPULATE-$S_1T_1E_0P_0$ instance must also be exponentially long

In the sections to come, when we will encounter domains that exhibit exponential length shortest plans, we will pay less attention to the complexity of PLANEX and PLANLEN because of the obvious implications of the exponential length result for PLANGEN and PLANOPT.

Theorem 4.9 PLANEX-MANIPULATE **and** PLANLEN-MANIPULATE $\in$ **PSPACE**
PLANEX-MANIPULATE *and* PLANLEN-MANIPULATE *can be solved with polynomial space.*
Proof: We show membership in **NPSPACE**, which proves the result because **PSPACE** = **NPSPACE**. States in MANIPULATE can be encoded using space that is polynomial in the description length because objects can easily be encoded using space that is polynomial in their size as can be verified inductively.

Given a state and action, applicability of that action can be checked in polynomial time, and the resulting state of the application can be computed in polynomial time. The same is true for membership in the goal set if a non-deterministic algorithm is used for guessing the correct permutation of table positions to match a state with the goal specification. So all these operations require no more than polynomial space.

Thus, a guess and check algorithm can solve PLANEX-MANIPULATE using polynomial space. For PLANLEN-MANIPULATE, a counter is needed to keep track of the number of actions in the plan, but this, again, only needs polynomial space in the size of the input (no more space than the part of the input that specifies the allowed plan length). ∎

Note that the latter result is not implied by the domain-dependent planning result in [Erol et al., 1995], because it is not obvious how the general MANIPULATE domain with its recursive transformations could be encoded in the language used there.

Putting the last two theorems together, we can conclude that any generalization of MANIPULATE-$S_1T_1E_0P_0$ in the MANIPULATE hierarchy, and thus all non-trivial domains in it that are neither construction nor transformation domains, are **PSPACE**-complete regarding PLANEX and PLANLEN. So we can limit our further analysis to these two subhierarchies, starting with transformation domains.

4.3 TRANSFORM

To simplify notation and terminology, we first observe that in TRANSFORM, as all objects are atomic, there is no need to distinguish between blueprints and objects. We will use these terms interchangeably.

For an instance signature $\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}})$ in the TRANSFORM family, B_C, B_D and $B^{\mathcal{I}}$ are required to be empty sets, and we can safely assume that $T_A = T$. Furthermore, all subtransformation specifications must be empty. We will thus omit these parts when specifying Σ and its components.

Of the three domains in this hierarchy, we will first investigate the easiest one, the no-equipment domain TRANSFORM-E_0.

Theorem 4.10 PLANLEN-TRANSFORM-E_0 $\in$ **P**
PLANLEN-TRANSFORM-E_0 *can be solved in polynomial time.*
Proof: As there is no equipment in this domain, reasonable plans solely consist of transformations.

Given an instance $I = ((B, T, T^{\mathcal{I}}), Pos, init, goal)$, we describe an algorithm for generating an optimal plan or detecting that no plan exists.

The *transformation digraph* of the instance is the digraph with vertex set B and arc set $\{(b, b') \mid \exists t \in T : t^{\mathcal{I}} = (b, b', \emptyset)\}$. If there is a directed path between two objects o and o', then the former object can be transformed into the latter by applying the transformation operations corresponding to the arcs of that path, in sequence. Shortest paths in this digraph correspond to shortest action sequences for a given multi-step transformation.

Without loss of generality, we can assume that *init* and *goal* are total functions. For $pos, pos' \in Pos$, let $dist(pos, pos')$ be the length of the shortest path between $init(pos)$ and $goal(pos')$ in the transformation digraph or ∞ if no such path exists.

Each initial object must ultimately be transformed into one of the goal objects, so for generating a shortest plan, we have to find a permutation $\sigma \in \mathrm{Sym}(Pos)$ that minimizes $\sum_{pos \in Pos} dist(pos, \sigma(pos))$. Assuming that the best σ has been found, a shortest plan consists of $|Pos|$ independent sequences of transformations, transforming the object at table position *pos* into the object $goal(\sigma(pos))$.

For calculating the *dist* values and coming up with the corresponding transformation sequences, a polynomial time graph search algorithm such as breadth-first search can be employed. Finding the optimal σ is a minimum weighted matching problem, for which polynomial algorithms are known (cf. [Mehlhorn and Näher, 1999]). Thus, the problem can be solved in polynomial time. ■

In the easier of the two remaining cases, the objects that are going to be used as equipment are not involved in transformations.

Theorem 4.11 PLANEX-TRANSFORM-E_1 $\in$ **P**
PLANEX-TRANSFORM-E_1 *can be solved in polynomial time.*
Proof: In this domain, transformations might require equipment. To reflect this, we redefine the transformation digraph as a *labeled* digraph, where the arc of a transformation $t \in T$ is labeled with the required equipment. Multiple arcs with different labels may exist between the same two vertices. In constructing the digraph, we check that for each blueprint in a label, an object of that type is part of the initial state. If this is not the case, this arc can never be used, and it is not created.

Once this has been done, the same two-dimensional matching algorithm as in Theorem 4.10 can be used to check if a plan exists. The actual plan is again made up of a number of individual transformation sequences corresponding to shortest paths in the transformation digraph, and each arc on such a path corresponds to a sequence of *commit* actions to allocate the equipment, a $transform$ action and a series of *release* actions to deallocate the equipment. ■

However, the plan generated by that algorithm is no longer a shortest plan, as some arcs are more costly in terms of the number of actions generated than others, and even if shortest paths in a weighted graph were used, not all *release* actions would be necessary. Indeed, the related optimization problem is harder, as will be proved now. But first we have to define the problem that is used as a basis for reduction.

Definition 4.12 3DM
Given three disjoint sets W, X, Y of the same cardinality n and $M \subseteq W \times X \times Y$, does M contain a three-dimensional matching, i. e. a subset M' of cardinality n such that $\bigcup_{(w,x,y) \in M'} \{w, x, y\} = W \cup X \cup Y$?

This problem is proved **NP**-complete in [Garey and Johnson, 1979].

Theorem 4.13 NP-completeness of PLANLEN-TRANSFORM-E_1
PLANLEN-TRANSFORM-E_1 *is* **NP**-*complete.*
Proof: Membership in **NP** follows from the preceding theorem.

For **NP**-hardness, we prove 3DM $\leq_p$ PLANLEN-TRANSFORM-E_1 by giving a polynomial transformation. The mapping between instances is defined as follows:

$$(W = \{w_1, \dots, w_n\}, X = \{x_1, \dots, x_n\}, Y = \{y_1, \dots, y_n\}, M) \mapsto (((B, T, T^{\mathcal{I}}), \mathit{Pos}, \mathit{init}, \mathit{goal}), L), \text{ where:}$$

$$\begin{aligned} B &\overset{\text{def}}{=} W \cup X \cup Y & \mathit{init}(i) &\overset{\text{def}}{=} w_i \text{ for } i \in \{1, \dots, n\} \\ T &\overset{\text{def}}{=} M & \mathit{init}(i) &\overset{\text{def}}{=} y_{i-n} \text{ for } i \in \{n+1, \dots, 2n\} \\ (w, x, y)^{\mathcal{I}} &\overset{\text{def}}{=} (w, x, \{y\}) \text{ for } (w, x, y) \in T & \mathit{goal}(i) &\overset{\text{def}}{=} x_i \text{ for } i \in \{1, \dots, n\} \\ \mathit{Pos} &\overset{\text{def}}{=} \{1, \dots, 2n\} & \mathit{goal}(i) &\overset{\text{def}}{=} y_{i-n} \text{ for } i \in \{n+1, \dots, 2n\} \\ L &\overset{\text{def}}{=} 2n \end{aligned}$$

This mapping satisfies all the requirements for a TRANSFORM-E_1 instance and can clearly be calculated in polynomial time.

Assume that M contains a three-dimensional matching $\{m_1, \dots, m_n\}$, where m_i is the triple $(w_{j_i}, x_{k_i}, y_{l_i})$. Then it is not hard to verify that the generated TRANSFORM-E_1 instance is solved by $[\mathit{commit}_{n+l_1,j_1}, \mathit{transform}_{j_1,m_1}, \dots, \mathit{commit}_{n+l_n,j_n}, \mathit{transform}_{j_n,m_n}]$, a plan of length $2n$.

Now assume that there is a plan of length at most $2n$ for the TRANSFORM-E_1 instance. None of the n objects on the first n table positions are part of the goal, so they all need to be transformed, requiring at least n transformation actions. As every transformation needs some equipment to be committed, there are at least n commitment actions, implying that there are exactly n actions of type *transform* and *commit* each and none of type *release*.

Let M' be the set containing the set of transformations that are used in the plan. M' has cardinality n. Because each of the first n objects from the initial state is transformed, M' covers all elements of W. Because each of the first n objects from the goal specification is created by one of these transformations, M' covers all the elements of X. Finally, because all equipment requirements are satisfied without any object being used as equipment twice (which would require a *release* action), and only the objects at initial table positions $n + 1, \dots, 2n$ can be used as equipment, M' covers all the elements of Y.

Thus, M' is a three-dimensional matching. This concludes the proof. ■

For the most general case of TRANSFORM, where equipment can be transformed and be created by a transformation, we will show that the length of shortest plans can grow exponentially with encoding length.

Theorem 4.14 Exponential plans for TRANSFORM-E_2
Shortest plan lengths in TRANSFORM-E_2 *can grow exponentially in the encoding length of the instance.*
Proof: We devise a sequence $(I_n)_{n \in \mathbb{N}_0}$ of instances for which encoding lengths grow polynomially but shortest plan lengths grow exponentially.

For $n \in \mathbb{N}_0$, $I_n = ((B, T, T^{\mathcal{I}}), \mathit{Pos}, \mathit{init}, \mathit{goal})$ is defined as follows:

$$\begin{aligned} B &\overset{\text{def}}{=} \textstyle\bigcup_{i \in \{1, \dots, n\}} \{A_i, B_i, C_i\} \\ T &\overset{\text{def}}{=} \textstyle\bigcup_{i \in \{1, \dots, n\}} \{AB_i, BC_i, CA_i\} \end{aligned}$$

$$\begin{aligned}
AB_i^{\mathcal{I}} &\overset{\text{def}}{=} (A_i, B_i, \{C_1, \dots, C_{i-1}\}) \text{ for } i \in \{1, \dots, n\} \\
BC_i^{\mathcal{I}} &\overset{\text{def}}{=} (B_i, C_i, \{A_1, \dots, A_{i-1}\}) \text{ for } i \in \{1, \dots, n\} \\
CA_i^{\mathcal{I}} &\overset{\text{def}}{=} (C_i, A_i, \emptyset) \text{ for } i \in \{1, \dots, n\} \\
Pos &\overset{\text{def}}{=} \{1, \dots, n\} \\
init(i) &\overset{\text{def}}{=} A_i \text{ for } i \in \{1, \dots, n\} \\
goal(i) &\overset{\text{def}}{=} C_i \text{ for } i \in \{1, \dots, n\}
\end{aligned}$$

Clearly, encoding length of I_n is polynomial in n.

Let $(l_n)_{n \in N_0}$ denote the length of the shortest plan for solving I_n. First notice that the shortest plan for solving I_1 consists of two steps, $transform_{1,AB_1}$ and $transform_{1,BC_1}$, thus $l_1 = 2$.

Now let $n \geq 1$ be a natural number. For solving I_{n+1}, an object of type C_{n+1} must be created at some point, which is only possible using transformation BC_{n+1}, which in turn requires an object of type B_{n+1}, only creatable by AB_{n+1}.

For AB_{n+1} to be applicable, equipment of types $C_1, \dots, C_n$ must be allocated to the object to be transformed. Towards this end, objects of this type must be created in the first place. Achieving this subgoal requires solving a subproblem identical to I_n, thus requiring at least l_n steps. Thus, more that l_n steps are needed before an object of type B_{n+1} can be created for the first time.

Now consider the state after an object of type C_{n+1} has been created for the first time. Because of the equipment requirements for BC_{n+1}, the other table positions must contain objects $A_1, \dots, A_n$ at this point, and transforming these to $C_1, \dots, C_n$ requires at least another $l(n)$ actions, thus $l(n+1) \geq 2l(n)$, and, since $l(1) = 2$, $l(n) \geq 2^n$ as required.

This does not conclude the proof, however. We have yet to show that for all $n \in \mathbb{N}_0$, I_n is actually solvable and thus the definition of $(l_n)_{n \in \mathbb{N}_0}$ makes sense. This is obvious for $n = 0$, as a zero-step plan solves this instance.

Now let n be a natural number and P_n be a plan that solves I_n. Then it can easily be verified that the following action sequence solves I_{n+1} :

$$\begin{aligned}
P_n \;&{+}{+}\; [commit_{1,n+1}, \dots, commit_{n,n+1}, transform_{n+1,AB_{n+1}}, release_1, \dots, release_n] \\
&{+}{+}\; [transform_{1,CA_1}, \dots, transform_{n,CA_n}] \\
&{+}{+}\; [commit_{1,n+1}, \dots, commit_{n,n+1}, transform_{n+1,BC_{n+1}}, release_1, \dots, release_n] \\
&{+}{+}\; P_n
\end{aligned}$$

■

4.4 Schedule

The Schedule domain from the AIPS 2000 competition can be seen as a special case of Transform-E_1: In this domain, the goal is to change certain properties of a set of objects, such as colour, shape, or surface condition, to match the goal requirements. This is achieved by processing (transforming) them with certain equipment, namely a polisher, a roller, a lathe, a grinder, a punch, a drill press, a spray painter and an immersion painter.

There are two aspects however, which make it hard to fit the Schedule domain into the framework we introduced: Firstly, different to Transform-E_1, the goal specification does not completely define the state of objects. Rather than requiring object A_1 to be cold, red, oblong and of a smooth surface, it may just be required that A_1 is red, no matter what its other properties are. It is not hard to achieve a similar effect in the

TRANSFORM framework, however, by adding extra transformations that can turn cold, red, oblong, smooth objects into red objects. This would affect plan length of course, but not in a way such that PLANLEN-SCHEDULE and its TRANSFORM equivalent could fall into different complexity classes.

Unfortunately, the second aspect cannot be accounted for that easily. The mechanisms for committing and releasing equipment in SCHEDULE are different to the ones used in TRANSFORM: The transformation actions (such as *do-lathe*) implicitly commit the required equipment (and cannot be invoked if it is already being used), and all equipment is released simultaneously, with just one *do-time-step* action. No more than one piece of equipment can be committed to an object at the same time. This is a difference in modelling that can cause different complexity results for PLANLEN-SCHEDULE and its counterpart in the TRANSFORM framework, so we can not avoid discussing the "real" SCHEDULE domain to come up with meaningful results.

Note, however, that the domains are sufficiently similar to allow for the conclusion that PLANEX-SCHEDULE $\in$ **P**, and indeed plans can be generated by an algorithm that is almost identical to the one from Theorem 4.11.

Definition 4.15 SCHEDULE **object states and machines**
The sets of SCHEDULE **temperatures**, **surface conditions**, **shapes**, **colours** *and* **holes** *are defined as follows:*

$$\begin{aligned} Temp &\overset{\text{def}}{=} \{cold, hot\} \\ Surface &\overset{\text{def}}{=} \{rough, smooth, polished, none\} \\ Shape &\overset{\text{def}}{=} \{cylindrical, circular, oblong\} \\ Colour &\overset{\text{def}}{=} \{blue, yellow, red, black, none\} \\ Hole &\overset{\text{def}}{=} \{Front1, Front2, Front3, Back1, Back2, Back3\} \end{aligned}$$

The set of SCHEDULE **object states** *is defined as*
$States \overset{\text{def}}{=} Temp \times Surface \times Shape \times Colour \times \mathbb{P}(Hole)$.

The set of SCHEDULE **goal state descriptions** *is defined as*
$GoalDesc \overset{\text{def}}{=} \mathbb{P}(States)$.

The set of SCHEDULE **equipment** *is defined as*
$Equip \overset{\text{def}}{=} \{roller, lathe, grinder, polisher, punch, drill\text{-}press, spray\text{-}painter, i.\text{-}painter\}$.

For each $e \in Equip$, $apply_e$ *is a fixed partial function between object states, the exact definition of which is not important to our analysis.*

Definition 4.16 SCHEDULE **instance**
The set of SCHEDULE **instances** *is defined to be* $(States \times GoalDesc)^*$. *The encoding length of an instance is assumed to be its (sequence) length.*

SCHEDULE instances specify a sequence of initial states for objects along with their goal description. Note that in the real PDDL domain, not all sets of states are candidates for a goal description, so that instances as defined here are in fact harder to solve than the "real" AIPS 2000 instances. However, as we will see by the results, this will not make a difference. It is of critical importance that *States* and *GoalDesc* are *not* part of the instance but fixed for all instances.

While it is in fact possible to define SCHEDULE instances with a richer state space based on the given PDDL domain file (by adding additional holes), this possibility is not made use of in the AIPS 2000 problem suite, and it is our goal to state results on the domain as it is used in the benchmarks.

Definition 4.17 Schedule **domain**
Let L be the set of all Schedule *instances. The* Schedule *domain is the function $\mathcal{D} : L \to \mathcal{M}$ mapping instances to state models as follows:*

$$((s_1, g_1), \dots, (s_n, g_n)) \mapsto (S, s_0, S_G, A, \delta), \text{ where}$$
$$S \overset{\text{def}}{=} \mathit{States}^n \times (\mathit{Equip} \nrightarrow \{1, \dots, n\}),$$
$$s_0 \overset{\text{def}}{=} ((s_1, \dots, s_n), \emptyset),$$
$$S_G \overset{\text{def}}{=} (g_1 \times \dots \times g_n) \times (\mathit{Equip} \nrightarrow \{1, \dots, n\}),$$
$$A \overset{\text{def}}{=} \{\ \mathit{use}_{\mathit{pos},e} \mid \mathit{pos} \in \{1, \dots, n\}, e \in \mathit{Equip}\ \} \cup \{\mathit{time\text{-}step}\},$$
$$\forall \mathit{pos} \in \{1, \dots, n\}, e \in \mathit{Equip}, s = ((s_1, \dots, s_n), \mathit{equip}) \in S :$$
$$s_{\mathit{pos}} \in \operatorname{dom}(\mathit{apply}_e) \wedge e \notin \operatorname{dom}(\mathit{equip}) \wedge \mathit{pos} \notin \operatorname{ran}(\mathit{equip})$$
$$\Rightarrow \delta(s, \mathit{use}_{\mathit{pos},e}) = ((s_1, \dots, s_{\mathit{pos}-1}, \mathit{apply}_e(s_{\mathit{pos}}), s_{\mathit{pos}+1}, \dots, s_n), \mathit{equip} \cup \{(e, \mathit{pos})\}),$$
$$\forall s = (\mathit{objs}, \mathit{equip}) \in S : \delta(s, \mathit{time\text{-}step}) \overset{\text{def}}{=} (\mathit{objs}, \emptyset) \text{ and finally}$$
δ is undefined otherwise.

States in this state model consist of a description of the object states of the various objects and the current equipment allocation function. Equipment allocation is reset by the *time-step* action. In the original Schedule domain, this action can only be applied if some *use* action was applied previously, but we do not have to model this explicitly because if this were not the case, there would be no point in applying *time-step* anyway.

We will now prove that PlanLen for this definition of the Schedule domain, which is a generalization of the Schedule domain as used in the AIPS 2000 competition, is already polynomial.

Theorem 4.18 PlanLen-Schedule $\in$ **P**
PlanLen-Schedule *can be solved in polynomial time.*
Proof: First, observe that objects with the same current state and goal specification need not be distinguished. As there is only a fixed number M of possible combinations of current state and goal specification of an object, a state of a Schedule instance (apart from the current allocation of equipment) can be described in an abstract way by an M-tuple of natural numbers, specifying *how many* objects of each kind are present. This will be called an *abstract state* of the instance. Abstract states where each entry greater than zero relates to a state/goal pair where the state matches the goal description are called *abstract goal states.*

The same applies to transformations: Rather than specifying that object k is being transformed using $e \in \mathit{Equip}$, it suffices to say that *some object* which matches the current state and goal description of k is being transformed using e. Thus, the *apply* operators can be reformulated to work on abstract states. Converting an abstract plan of that kind into an actual plan is sufficiently easy to not require further discussion.

To avoid having to worry about allocated equipment, the abstract actions can be replaced by *abstract macro actions*, sequences of abstract actions containing exactly one *time-step* action, which is the last one in the sequence. Each plan can be partitioned into macro actions of that kind, assuming that it ends in a *time-step* action (which does not make sense for shortest plans, but we can require this property and then remove the last action after the plan has been generated). The advantage of this view is that before and after each abstract macro action, no equipment is allocated to any object.

Because there are only eight items of equipment, each abstract macro action can consist of no more than nine actions, one for each item plus the concluding *time-step*. Thus, the number of possible macro moves is constant.

What is gained by recasting the problem in that way? As was said before, abstract states can be represented by an M-tuple of natural numbers, where $M = |States| \cdot |GoalDesc|$ is fixed and each natural number in this tuple can be bounded by the total number of objects in the instance, which is its encoding length. Thus, the number of abstract states is *polynomial* in the encoding length.

This means that explicit graph-search techniques can be used to find a shortest plan consisting of abstract macro moves. Because different macro moves can comprise a different number of actions (between 1 and 9), they should be weighted by that number. Still, the one-to-all shortest path problem in a weighted digraph can be solved in polynomial time, and it is then easy to pick the abstract goal state with shortest weighted distance from the abstract initial state (if any such state exists), extract the abstract plan, expand the macro actions and assign actual objects to the abstract actions to compute an optimal sequential plan in polynomial time. ■

Note that in reality, people are usually not really interested in finding shortest plans in the SCHEDULE domain, but finding plans that involve a minimal number of *time-step* actions. This can be achieved with the algorithm described in the last proof by using an *unweighted* graph, i. e. weighting all macro moves with the same value of 1.

Also note that the execution time of this algorithm in its basic form grows at least as quickly as N^M, where N is the input size and $M = |States| \cdot |GoalDesc| = 7680 \cdot 2^{7680}$. Although the "real" *GoalDesc* is far smaller, as the kind of disjunctive goal conditions used in the AIPS competition reduces its cardinality from 2^{7680} to 80, an $N^{7680 \cdot 80}$ algorithm still is not tractable in practice.

Although further optimization techniques can be used to decrease the complexity significantly, it is not obvious what a really tractable algorithm could look like.

This concludes our discussion of transformation domains and we move on to the last family of domains to be discussed, ones focusing on construction.

4.5 CONSTRUCT and BLOCKSWORLD

In this subfamily of MANIPULATE, no transformations take place, which again allows to simplify notation.

For a CONSTRUCT instance signature $\Sigma = (B, B_C, B_D, B^{\mathcal{I}}, T, T_A, T^{\mathcal{I}})$, the last three components will always be empty sets, and we will omit them.

We have already seen that shortest plans can get exponentially long in the MANIPULATE domain when transformations and composites are allowed (Theorem 4.8) or when transformations and arbitrary equipment requirements are allowed (Theorem 4.14). We will now investigate two cases of CONSTRUCT that exhibit this behaviour, starting with construction of stacks with limited table positions.

Theorem 4.19 Exponential plans for CONSTRUCT-$S_1E_0P_1$
Shortest plan lengths in CONSTRUCT-$S_1E_0P_1$ *can grow exponentially in the encoding length of the instance.*
Proof: We devise a sequence $(I_n)_{n\geq 2}$ of instances for which encoding lengths grow polynomially but shortest plan lengths grow exponentially.

For $n \geq 2$, , $I_n = ((B, B_C, B_D, B^{\mathcal{I}}), \mathit{Pos}, \mathit{init}, \mathit{goal})$ is defined as follows:

$$\begin{aligned}
B &\overset{\text{def}}{=} \{\ A_i \mid i \in \{1,\dots,n\}\ \} \cup \{\ T_i \mid i \in \{1,\dots,n-1\}\ \}\\
B_C \overset{\text{def}}{=} B_D &\overset{\text{def}}{=} \{\ T_i \mid i \in \{1,\dots,n-1\}\ \}\\
T_i^{\mathcal{I}} &\overset{\text{def}}{=} (\{T_{i+1},\dots,T_n,A_{i+1},\dots,A_n\},\{A_i\},\emptyset) \text{ for } i \in \{1,\dots,n-1\}\\
\mathit{Pos} &\overset{\text{def}}{=} \{1,2,3\}\\
\mathit{init} &\overset{\text{def}}{=} \{(1,(T_1,(T_2,\dots,(T_{n-1},A_n,A_{n-1}),\dots,A_2),A_1))\}\\
\mathit{goal} &\overset{\text{def}}{=} \{(1,(T_1,(T_2,\dots,(T_{n-2},A_{n-1},A_{n-2}),\dots,A_2),A_1)),(2,A_n)\}
\end{aligned}$$

Clearly, encoding length of I_n is polynomial in n.

If A_i is interpreted as "a block of width i", and T_i is read as "a tower with a block of width i on top", then this formalizes the semantics of the Towers of Hanoi problem set, apart from the goal specification. An object of type T_i consists of a tower or block of greater width than i and a block of width i on top of it, capturing the constraints for a Towers of Hanoi problem.

In the initial state, all blocks are piled up in one big tower at the first table position, just like in a usual Towers of Hanoi problem. Note that making use of the normal Towers of Hanoi goal of moving the whole tower to the third table position would not make sense, because table positions of goal objects are not distinguished in the CONSTRUCT domain. So the modified goal is to separate the top $n-1$ blocks of the initial tower from the bottom-most block, which is in a way "half" the Towers of Hanoi problem.[5]

Thus, the length of the shortest solution to that instance is half the length of the solution to the corresponding Towers of Hanoi problem minus one, which is exponential in n. More precisely, it is $2^{n-1}-1$. Because the Towers of Hanoi problems and their solutions are well-known, we will not go into more detail here. ■

The previous proof shows that one source of long plans is restricted space on the table: Intermediary objects that are not immediately useful towards satisfying the goal need to be built in order to make room for other movements. Another source of long plans are complex equipment requirements, as seen in Theorem 4.14. The same is true if transformations are not allowed, but composites are, even if only composites consisting of two atomic parts are used. This is the message of the following theorem.

Theorem 4.20 Exponential plans for CONSTRUCT-$S_1E_2P_0$
Shortest plan lengths in CONSTRUCT-$S_1E_2P_0$ *can grow exponentially in the encoding length of the instance, even if composite blueprints solely consist of atomic parts.*
Proof: We devise a sequence $(I_n)_{n\in\mathbb{N}_0}$ of instances for which encoding lengths grow polynomially but shortest plan lengths grow exponentially.

[5]Remember that in Towers of Hanoi, the optimal plan first moves the top $n-1$ blocks to the second position, then moves the bottommost block to the third position, then moves the top $n-1$ blocks to the third position.

For $n \in \mathbb{N}_0$, $I_n = ((B, B_C, B_D, B^{\mathcal{I}}), Pos, init, goal)$ is defined as follows:

$$
\begin{aligned}
B &\overset{\text{def}}{=} \bigcup_{i\in\{1,\dots,n\}} \{A_i, B_i, AB_i, C_i, D_i, CD_i\} \\
B_C \overset{\text{def}}{=} B_D &\overset{\text{def}}{=} \bigcup_{i\in\{1,\dots,n\}} \{AB_i, CD_i\} \\
AB_i^{\mathcal{I}} &\overset{\text{def}}{=} (\{A_i\}, \{B_i\}, \{AB_1, \dots, AB_{i-1}, CD_1, \dots, CD_{i-1}\}) \text{ for } i \in \{1, \dots, n\} \\
CD_i^{\mathcal{I}} &\overset{\text{def}}{=} (\{C_i\}, \{D_i\}, \{AB_i, A_1, \dots, A_{i-1}, C_1, \dots, C_{i-1}\}) \text{ for } i \in \{1, \dots, n\} \\
Pos &\overset{\text{def}}{=} \{1, \dots, 4n\} \\
init &\overset{\text{def}}{=} \{(1, A_1), (2, B_1), (3, C_1), (4, D_1), \dots, \\
&\qquad (4n-3, A_n), (4n-2, B_n), (4n-1, C_n), (4n, D_n)\} \\
goal &\overset{\text{def}}{=} \{(1, (AB_1, A_1, B_1)), (3, (CD_1, C_1, D_1)), \dots \\
&\qquad (4n-3, (AB_n, A_n, B_n)), (4n-1, (CD_n, C_n, D_n))\}
\end{aligned}
$$

Clearly, encoding length of I_n is polynomial in n.

We first show that all these instances are solvable. For $n = 0$, the empty plan is a solution. Let P_n be a plan for I_n and Q_n be a plan for the inverse problem (with goal and init specification swapped). Then P_{n+1} and Q_{n+1} are defined as follows:

$$
\begin{aligned}
P_{n+1} &\overset{\text{def}}{=} P_n \mathbin{++} [commit_{1,4n+1}, commit_{3,4n+1}, \dots, commit_{4n-1,4n+1}] \\
&\mathbin{++} [pick_{4n+2}, compose_{4n+1,AB_{n+1}}, release_1, release_3, \dots, release_{4n-1}] \mathbin{++} Q_n \\
&\mathbin{++} [commit_{1,4n+3}, commit_{3,4n+3}, \dots, commit_{4n+1,4n+3}] \\
&\mathbin{++} [pick_{4n+4}, compose_{4n+3,CD_{n+1}}, release_1, release_3, \dots, release_{4n+1}] \mathbin{++} P_n \\
Q_{n+1} &\overset{\text{def}}{=} Q_n \mathbin{++} [commit_{1,4n+3}, commit_{3,4n+3}, \dots, commit_{4n+1,4n+3}] \\
&\mathbin{++} [decompose_{4n+3}, drop_{4n+4}, release_1, release_3, \dots, release_{4n+1}] \mathbin{++} P_n \\
&\mathbin{++} [commit_{1,4n+1}, commit_{3,4n+1}, \dots, commit_{4n-1,4n+1}] \\
&\mathbin{++} [decompose_{4n+1}, drop_{4n+2}, release_1, release_3, \dots, release_{4n-1}] \mathbin{++} Q_n
\end{aligned}
$$

It should not be hard to verify that this solves these instances. What about minimal plan length? Solving I_1 requires more than zero steps, because init and goal specification do not match.

For solving I_{n+1} for $n \geq 1$, an object of type AB_{n+1} must be created at some point. This requires solving I_n to get the necessary equipment. Before an object of type CD_{n+1} is created for the first time, an object of type AB_{n+1} must be present and apart from that, the initial state must have been restored, so after creating the object of type CD_{n+1}, objects of type AB_1, …, AB_n must be recreated from scratch, which again requires solving I_n. So the length of the shortest plan for I_{n+1} is at least twice the length of the shortest plan for I_n, proving the exponential length claim. ■

Summarizing the previous two results, Construct can require exponential length plans if arbitrary equipment requirements are allowed or space is restricted. The cases that still need to be discussed are thus all specializations of Construct-$S_2E_1P_0$, with separate (or no) equipment blueprints and unlimited space.

We start our discussion of these domains by showing that plans can be generated in polynomial time (and are of polynomial length).

Theorem 4.21 PlanEx-Construct-$S_2E_1P_0 \in \mathbf{P}$

PlanEx-Construct-$S_2E_1P_0$ can be solved in polynomial time.

Proof: For PlanEx, there is no significant difference between having separate equipment blueprints or no equipment blueprints. If the length of the plan is not critical, equipment can be allocated "on demand", i. e. whenever a given *compose* or *decompose* action requires certain equipment, objects of the required types are committed to the target location and immediately released again after the action. This can only increase plan length by a polynomial factor (as the number of *commit* and *release* actions per *compose* or *decompose* action are each limited by the number of blueprints in the instance).

Equipment requirements can only get in the way where it is not possible to satisfy them. However, it is easy to check that: Just record the set of equipment blueprints for which objects are provided in the initial state and then check for each composite blueprint that it does not require any equipment not in that set. If it does, it can be seen as if it were not composable or decomposable. Thus, we can and will ignore equipment for the rest of this proof. The main difficulty in these instances lies in handling blueprints that can be composed but not decomposed or vice versa, otherwise a simple "take everything apart, then stick it together in the right way" strategy would suffice, as there is unlimited table space.

The algorithm we are going to present has two phases, and so has the resulting plan. In the first phase, objects are being decomposed, so only *decompose* and *drop* actions are applied. Once this has been done, the second phase combines the generated objects with *pick* and *compose* actions. It is not hard to see that if any plan exists, then plans that can be separated into these two phases exist, given that table space is unlimited.

We will first talk about the second, easier to accomplish phase. To combine a set of objects to form the goal objects, build one goal object after the other. If the corresponding object already exists, mark it as used (remove it from further consideration) and proceed with the next goal object. If not, recursively build the immediate subparts that object is constructed from and combine them. If this is not possible because that blueprint is not composable or if recursive calls require an atom which is not available, then fail. Go on until all goal objects are built or failure is detected. Provided that the goal objects can be composed from the given set of objects, this will clearly generate a plan that achieves the goal.

So the critical problem lies in decomposing things properly. This means that we should not decompose an object further unless that is necessary, because it might not be possible to compose it again afterwards.

To facilitate understanding the decomposition phase, we observe a correspondence between objects and (rooted) binary trees where inner nodes are labeled with composite blueprints and leaves are labeled with atomic blueprints: The atom b corresponds to the one-node tree with b as its label, and the composite (b, p_1, p_2) corresponds to the tree with a root node labeled b and with trees for p_1 and p_2 as its left and right son, respectively. We say that a given object is a *subpart* of o if its tree is a subtree of the tree for o.

The decomposition phase then proceeds as follows:

1. *currState* is initialized as a sequence of trees corresponding to the initial objects. *goals* is initialized as a sequence of trees corresponding to the objects in the goal specification.

2. While $goals \neq \emptyset$, do the following:

   (a) If *goals* contains a tree from *currState*, then if any plan exists, there is some plan in which the object represented by that member of *currState* will be used to satisfy the goal represented by that element of *goal*. It thus does not need to be decomposed further. Thus, the tree is removed from both *currState* and *goals*.

(b) Otherwise, if there exists $T \in \mathit{currState}$ which is not a subtree of any element of *goal*, then the corresponding object needs to be decomposed. If this is not possible, fail. If it is, add a *decompose* and *drop* action to the plan to replace that object by its immediate subparts and update *currState* by removing T and adding the children of the root node.

(c) Otherwise, each element of *currState* is a proper subtree of some element of *goal*. Choose one such tree $T \in \mathit{currState}$ with a maximal number of leaves (implying that the corresponding object is not a proper subpart of any other object represented in *currState*) and a goal tree T_G which has T as a subtree.

At some point in the composition phase an object with tree T will be needed to form the goal object represented by T_G, as no bigger object that could contain T as part of its tree representation is available or could be made available by decomposition. So it is safe to allocate T to that task now, removing it from *currState* and cutting the (or rather *one*, if there is more than one) occurrence of T as a subtree of T_G from T_G, because this part of the goal object is being provided now and need not be considered again during decomposition.

If this leads to the father of the removed subtree becoming a leaf node, it must also be removed from T_G because all of its constituents have been allocated already. If this happens, its father is checked recursively, possibly propagating all the way to the root. If the root is cut, T_G is removed from *goals* entirely.

Soundness of the algorithm should be obvious from the descriptions motivating the various steps. It is polynomial because for case (a), only a linear number of goals can be matched, for (b), only a linear number of decomposition operators can be applied, and for (c), only a linear number of tree nodes can be cut.

This concludes the proof. ■

This result implies that PlanLen-Construct-$S_2E_1P_0 \in \mathbf{NP}$, by a polynomial plan length argument. To prove **NP**-hardness, we now introduce a special case of the most specific construction domain in our hierarchy, Construct-$S_1E_0P_0$.

Definition 4.22 Blocksworld **domain**
The Blocksworld *domain is the special case of* Construct-$S_1E_0P_0$ *where for all instances* $((B, B_C, B_D, B^{\mathcal{I}}), Pos, init, goal)$ *there exists some* $n \in \mathbb{N}_0$ *such that the following holds:*

$$\begin{aligned} B &= \{tower, block_1, \ldots, block_n\} \\ B_C = B_D &= \{tower\} \\ tower^{\mathcal{I}} &= (\{tower, block_1, \ldots, block_n\}, \{block_1, \ldots, block_n\}, \emptyset) \\ Pos &= \{1, \ldots, n\} \end{aligned}$$

All atomic objects appear as subparts of init and goal objects exactly once.

In Blocksworld, there are blueprints for individual blocks and one blueprint for a tower of blocks. By definition of *init* and *goal*, all block objects have different blueprints (i. e., they are distinguishable). Dropping the restriction on *init* and *goal* would lead to the interesting variant of Blocksworld where some blocks cannot be distinguished.

A tower can be formed by attaching any block to an existing tower or another block. The number of table positions is essentially unlimited, since there are as many table positions as blocks. As is always the case in the MANIPULATE framework, initial and goal states are completely specified.

Theorem 4.23 NP-completeness of PLANLEN-BLOCKSWORLD
PLANLEN-BLOCKSWORLD *is* **NP***-complete.*

This was proved in [Gupta and Nau, 1992]. The proof they give is based on a slightly different formulation of the problem, where moving a block between towers or between a tower and the table requires but one action, not two of them as is the case for our version, where these movements are accomplished by a *pickup* or *decompose* action followed by *drop* or *compose*. The proof in the reference still applies, however, if the maximal allowable plan length is doubled to take this difference in definition into account.

The two remaining sections on construction domains are concerned with two other domains from the AIPS competition, which both require some special treatment.

4.6 FREECELL

The FREECELL domain used in the AIPS 2000 competition is based on a popular solitaire game, the rules of which are as follows:

The game is played with a standard deck of 52 cards, initially arranged into eight *tableau piles* of six or seven cards each. Any such partitioning of the cards is a valid initial configuration. Cards can then be moved between these eight tableau piles, four *free cells* and four *foundation piles* according to the following rules:

- Cards may only be picked up if they occupy a free cell or if they are the top card of a tableau pile. No more than one card can be picked up at the same time.
- Cards may only be dropped in a free cell if it does not currently hold any other card.
- Cards may only be added to a tableau pile (as its new top card) if that pile is empty, or the value of the card is one less than the value of the top card of the pile and it is of a different colour (e. g., the four of spades can only be added to tableau piles with the five of diamonds or hearts as their top card).
- Aces may be added to an empty foundation pile. Other cards may only be added to a foundation pile if their value is one higher than the value of the top card of the pile and they are of the same suit.

The objective of the game is to move all cards to foundations. This can be achieved from most, but not all initial configurations.

Of course, the standard FREECELL game only allows for a constant number of different configurations because of the limited number of cards, and indeed none of the problems in the AIPS competition featured more than 52 cards. This is not very interesting from a complexity theory point of view, because it only makes sense to discuss the complexity of problems with an infinite number of instances. There are various ways of increasing the number of instances:

1. Adding additional suits, either by adding more suits per colour (red or black) or by adding colours.
2. Increasing the number of cards per suit.
3. Increasing the number of free cells.
4. Increasing the number of tableau piles.

The third and fourth options alone do not lead to an infinite number of instances, because as soon as there are as many free cells or tableau piles as cards, it does not make a difference how many of them there are exactly.

Of the other two options, the second seems to be the more natural one. However, increasing the number of cards per suit without arranging for extra space to move these cards around must inevitably lead to an ever decreasing fraction of solvable instances as the number of cards increases. To come up with more interesting instances, it makes sense to adjust the number of free cells and tableau piles, too. We will show that solvability of FreeCell instances is an **NP**-complete problem even if only the number of cards per suit and the number of tableau piles changes with instances, for any fixed number of free cells. This is the generalized FreeCell domain we will define now.

To facilitate understanding, we will use the symbols $\diamondsuit$, $\heartsuit$, $\spadesuit$ and $\clubsuit$, which can be regarded as synonyms of 1, 2, 3 and 4, respectively.

Definition 4.24 FreeCell **domain**
The FreeCell *domain is the special case of* Construct-$S_1E_0P_1$ *where for all instances* $((B, B_C, B_D, B^{\mathcal{I}}), \mathit{Pos}, \mathit{init}, \mathit{goal})$ *there exists a four-tuple* $(\mathit{val}, \mathit{col}, \mathit{cells}, \mathit{setup}) \in \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}_0 \nrightarrow \mathbb{N}_0 \times \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\})$ *such that the following holds:*

$$\begin{aligned}
\operatorname{dom}(\mathit{setup}) &= \{\ (c,r)\ \in \{1,\dots,\mathit{col}\} \times \mathbb{N}_0 \mid r \geq 1 \wedge (r-1)\cdot \mathit{col} + c \leq 4\ \mathit{val}\ \} \\
\operatorname{ran}(\mathit{setup}) &= \{1,\dots,\mathit{val}\} \times \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\} \\
B_{\mathit{Card}} &= \{\ c_{v,s} \mid v \in \{1,\dots,\mathit{val}\}, s \in \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\}\ \} \\
B_{\mathit{Tableau}} &= \{\ \mathit{red}_v \mid v \in \{1,\dots,\mathit{val}-1\}\ \} \cup \{\ \mathit{black}_v \mid v \in \{1,\dots,\mathit{val}-1\}\ \} \\
B_{\mathit{Cell}} &= \{\mathit{emptycell}, \mathit{filledcell}\} \\
B_{\mathit{Found}} &= \{\ \mathit{found}_{v,s} \mid v \in \{0,\dots,\mathit{val}\}, s \in \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\}\ \} \\
B_{\mathit{Init}} &= \{\ \mathit{init}_{c,r} \mid (c,r) \in \operatorname{dom}(\mathit{setup}), r > 1\ \} \\
B &= B_{\mathit{Card}} \cup B_{\mathit{Tableau}} \cup B_{\mathit{Cell}} \cup B_{\mathit{Found}} \cup B_{\mathit{Init}} \\
B_C &= B_{\mathit{Tableau}} \cup \{\mathit{filledcell}\} \cup B_{\mathit{Found}} \setminus \{\mathit{found}_{0,s} \mid s \in \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\}\ \} \\
B_D &= B_{\mathit{Tableau}} \cup \{\mathit{filledcell}\} \cup B_{\mathit{Init}} \\
\mathit{red}_v^{\mathcal{I}} &= (B \cap \{c_{v+1,\spadesuit}, c_{v+1,\clubsuit}, \mathit{black}_{v+1}, \mathit{init}_{\mathit{setup}^{-1}((v+1,\spadesuit))}, \mathit{init}_{\mathit{setup}^{-1}((v+1,\clubsuit))}\}, \\
&\qquad \{c_{v,\diamondsuit}, c_{v,\heartsuit}\}, \emptyset)\ \mathit{for}\ v \in \{1,\dots,\mathit{val}-1\} \\
\mathit{black}_v^{\mathcal{I}} &= (B \cap \{c_{v+1,\diamondsuit}, c_{v+1,\heartsuit}, \mathit{red}_{v+1}, \mathit{init}_{\mathit{setup}^{-1}((v+1,\diamondsuit))}, \mathit{init}_{\mathit{setup}^{-1}((v+1,\heartsuit))}\}, \\
&\qquad \{c_{v,\spadesuit}, c_{v,\clubsuit}\}, \emptyset)\ \mathit{for}\ v \in \{1,\dots,\mathit{val}-1\} \\
\mathit{filledcell}^{\mathcal{I}} &= (\{\mathit{emptycell}\}, B_{\mathit{Card}}, \emptyset) \\
\mathit{found}_{v,s}^{\mathcal{I}} &= (\{\mathit{found}_{v-1,s}\}, \{c_{v,s}\}, \emptyset)\ \mathit{for}\ v \in \{1,\dots,\mathit{val}\},\ s \in \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\} \\
\mathit{init}_{c,2}^{\mathcal{I}} &= (\{c_{\mathit{setup}((c,1))}\}, \{c_{\mathit{setup}((c,2))}\}, \emptyset)\ \mathit{for}\ c\ \mathit{with}\ (c,2) \in \operatorname{dom}(\mathit{setup}) \\
\mathit{init}_{c,r}^{\mathcal{I}} &= (\{\mathit{init}_{c,r-1}\}, \{c_{\mathit{setup}((c,3))}\}, \emptyset)\ \mathit{for}\ (c,r) \in \operatorname{dom}(\mathit{setup})\ \mathit{with}\ r \geq 3
\end{aligned}$$

$$\begin{aligned}
\mathit{Pos} &= \{\mathit{tableau}_1, \dots, \mathit{tableau}_{\mathit{col}}\} \cup \{\mathit{cell}_1, \dots, \mathit{cell}_{\mathit{cells}}\} \\
&\quad \cup \{\mathit{found}_\diamondsuit, \dots, \mathit{found}_\clubsuit\} \\
\mathit{init} &= \{\ (\mathit{tableau}_c, (\mathit{init}_{c,r(c)}, (\mathit{init}_{c,r(c)-1}, \dots, (\mathit{init}_{c,2}, c_{\mathit{setup}((c,1))}, c_{\mathit{setup}((c,2))}), \\
&\qquad \dots, c_{\mathit{setup}((c,r(c)-1))}), c_{\mathit{setup}((c,r(c)))}))\ | \\
&\qquad c \in \{1, \dots, \mathit{col}\}, r(c) = \max\{\mathit{row} \in \mathbb{N}_0 \mid (c, \mathit{row}) \in \mathrm{dom}(\mathit{setup})\ \}\ \} \\
&\quad \cup \{(\mathit{cell}_1, \mathit{emptycell}), \dots, (\mathit{cell}_{\mathit{cells}}, \mathit{emptycell})\} \\
&\quad \cup \{(\mathit{found}_\diamondsuit, \mathit{found}_{0,\diamondsuit}), \dots, (\mathit{found}_\clubsuit, \mathit{found}_{0,\clubsuit})\} \\
\mathit{goal} &= \{\ (\mathit{found}_s, (\mathit{found}_{\mathit{val},s}, (\mathit{found}_{\mathit{val}-1,s}, \dots, (\mathit{found}_{1,s}, \mathit{found}_{0,s}, c_{1,s}), \\
&\qquad \dots, c_{\mathit{val}-1,s}), c_{\mathit{val},s}))\ |\ s \in \{\diamondsuit, \heartsuit, \spadesuit, \clubsuit\}\ \} \\
&\quad \cup \{(\mathit{cell}_1, \mathit{emptycell}), \dots, (\mathit{cell}_{\mathit{cells}}, \mathit{emptycell})\}
\end{aligned}$$

This definition embeds the generalized FreeCell domain into the Construct hierarchy. It is a domain with stack-like composites, no equipment and a restricted number of table positions. An instance is characterized by the number of cards per suit (*val* for different card *values*), the number of tableau piles (*col* for *columns*), the number of free cells (*cells*) and the initial setup of cards.

Cards are referred to as (v, s), denoting the value and suit of the card. In standard FreeCell problems, the value ranges from 1 to 13, where 1 corresponds to an ace and 13 to a king. The suit can be diamonds, hearts, spades or clubs. Throughout the definition, v always denotes the value and s the suit of a card.

The initial setup function maps positions in the tableau to cards, where the mapping $\mathit{setup}((r, c)) = (v, s)$ means that the card (v, s) is initially located in the r-th *row* (i. e., is the r-th card from the bottom of the pile) of the c-th *column* (i. e. tableau pile) of the initial tableau. The constraints ensure that each card appears at exactly one position and that all tableau piles comprise an equal number of cards. If the number of cards is not a multiple of the number of tableau piles, the first piles will hold one more card.

The blueprints in FreeCell instances are individual cards ($c_{v,s}$), tableau piles characterized by the value and *colour* of their top card (red_v for diamonds and hearts, black_v for spades and clubs), free cells holding or not holding a card ($\mathit{filledcell}$ and $\mathit{emptycell}$), foundation piles characterized by their top card ($\mathit{found}_{v,s}$, where a value of 0 denotes a foundation pile without any card in it) and *initial* tableau piles, which differ from other tableau piles in that they do not have to adhere to the constraints for putting a card on top of another one ($\mathit{init}_{c,r}$, the blueprint for the r bottommost cards of the c-th tableau pile).

Cards can be added to tableau piles, free cells or foundations by creating new objects of types from B_{Tableau}, $\mathit{filledcell}$ or B_{Found}. Note that blueprints from B_{Init} are not composable because they do not necessarily adhere to the constraints on the suit and value of the top card.

However, initial piles can always be decomposed, as can "legal" tableau piles from B_{Tableau} and filled free cells. Cards cannot be moved from foundations, so B_{Found} does not contain decomposable blueprints.

The interpretation of the various blueprints follows the intuitive description of the rules of FreeCell provided earlier in this section. The most important part is the definition of the allowed base parts of red and black tableau piles. The definition lists all blueprints of cards with a matching colour and value and piles of cards that can appear on the tableau (that is, elements of B_{Tableau} and B_{Init}) which have a top card satisfying these requirements. The specified sets are intersected with B to avoid having to specify special cases (for example, red_{val} and $init_{1,1}$ do not exist in B, so they are ruled out by this operation).

There are three different kinds of table positions, related to the tableau, free cells or foundations. Although objects can of course be picked up and dropped on different table positions to destroy the intuitive relationship between the names of the table positions and their purpose, there will always be exactly *cells* table positions occupied by objects of type *emptycell* or $filledcell$ and exactly one table position occupied by an object of type $found_{v,s}$ for each of the four possible values of s. Thus, it is not possible to temporarily trade in an additional table position for the tableau by only using three foundation piles or violate the rules of the game in a similar way.

The specifications of the initial state and goal should be self-explanatory.

From Theorem 4.19, we know that there are sets of instances within the CONSTRUCT-$S_1E_0P_1$ domain that feature very long plans. The FREECELL domain, especially in the special case without any free cells, resembles the Towers of Hanoi problem used in that proof in some ways. However, as we will prove now, it does not have the same exponential plan length property.

Theorem 4.25 PLANLEN-FREECELL $\in$ **NP**
PLANEX-FREECELL *and* PLANLEN-FREECELL *are in* **NP**.
Proof: We will prove that FREECELL is a domain with polynomial length plans and apply Lemma 2.10.

To this end, we first observe that there are only two kinds of actions within FREECELL that cannot be undone immediately by applying an inverse action, namely decomposing a card from an initial pile and adding a card to a foundations pile. If m is the number of cards in the instance, then any plan will contain no more than m actions of the first and exactly m actions of the second kind. Thus, the number of non-undoable actions is polynomial in the number of cards and thus the size of the instance, and thus we only need to come up with a polynomial bound for the length of any sequence of *undoable* actions appearing within an optimal plan.

To provide this bound, we will give an algorithm that, given an initial and goal state such that the goal state can be reached from the initial state by only using undoable actions, calculates a polynomial length action sequence that accomplishes this state transition. We call this subproblem without "unsafe" moves the *safe* FREECELL *problem.*

In fact, we will not bound the actual number of actions but the number of *macro moves* that are part of the plan, where a macro move is a sequence of actions that moves an entire tableau pile or several cards of it onto another tableau pile or to an empty column of the tableau, possibly making use of free cells, empty positions and other piles. A macro move that moves one card simply consists of a single pickup or decompose and a single drop or compose action.

Macro moves involving m free cells as temporary storage positions only (*free cell moves*) can move no more than $m + 1$ cards, moving one card into each free cell, then moving the bottommost card, then moving the cards from the free cells to their destination.

Let $C(m, k)$ be the maximum number of cards that can be moved from one table position to another by using m free cells and k empty table positions as temporary locations, and let $T(m, k)$ be the number of free cell moves required for that operation. Clearly, $C(m, 0) = m + 1$ and $T(m, 0) = 1$.

Having $k > 0$ temporary locations for moving a pile of cards from A to C, the operation can be divided into three parts: First move some of the cards from A to an intermediary location B, then move the remaining cards from A to C, and finally move the cards from B to C. For the last two operations, $k-1$ temporary locations are available (the original ones except B), so the number of cards that can be moved in this fashion is $2C(m, k-1)$, and clearly $T(m,k) = 3T(m, k-1)$. This recursion solves to $C(m,k) = (m+1)2^{k-1}$ and $T(m,k) = 3^{k-1}$, and thus $T(m,k) = (C(m,k)/(m+1))^{log_2 3}$, showing that the number of moves required is polynomial in the number of cards being moved. The same is true for macro moves also involving non-empty table positions as temporary locations, as is not hard to see and will not be spelled out in detail. This property is the fundamental difference between the Towers of Hanoi problem and FreeCell.

A card (v, c) is called *red-even* if $c \in \{\diamondsuit, \heartsuit\}$ and v is even, or if $c \notin \{\diamondsuit, \heartsuit\}$ and v is odd. Otherwise, it is called *red-odd.* In tableau piles, only red-even cards are added to red-even cards, and only red-odd cards are added to red-odd cards. Consequently, the same adjectives can be used when referring to tableau piles. The card on top of another card in a tableau pile is called the *son* of that card.

The safe FreeCell problem can be solved in three stages: Firstly, *compact* the tableau by making all piles as big as possible, freeing up as many table positions and free cells as possible. This can clearly be done with a polynomial number of steps, because each card needs to be made the son of a given other card as the effect of a macro move at most once.

Secondly, identify *bad son cards*, i. e. cards which have a son in both the current and goal state, and the two son cards are of different suits (they must have the same value and colour). As long as there are any bad son cards, choose one of them with maximal value, apply a number of macro moves to move all cards on top of the bad son card to other locations, using up as few free cells and empty table positions as possible, then apply a macro move to move the pile of cards with the correct son on top of the bad son card and compact the tableau again. This card will now no longer be a bad son card. Clearly, this takes a polynomial number of macro moves.

Only cards of lower value than the corrected card c are moved in this process, which means that cards of a higher value than the corrected card cannot become bad son cards as a result of these movements. We do not care if cards of a lower value are made bad son cards, because this can be changed by further iterations of the correction procedure. A card c_C of an equal value but different colour cannot be made a bad son card because if c is red-even, all cards moved are red-even and thus cannot be moved on top of c_C, and if c is red-odd, all cards moved are red-odd and again c_C is unaffected.

Finally, the card c_S of an equal value and colour but different suit as c cannot be a bad son card afterwards because if c_S has a son in both the current and goal state, then it must be the correct one, because otherwise the (newly corrected) son of c would have to be incorrect, too. Thus, after a number of iterations of this correction procedure that is limited by the total number of cards, there are no longer any bad son cards.

Once there are no more bad son cards, all that needs to be done to reach the goal state is rearranging the piles in a way that is inverse to the compacting procedure that formed the first step. The number of macro moves in this third part of the action sequence can consequently be bounded by the same polynomial as was used for the first step.

Putting the different parts together, we conclude that there is an overall polynomial bound on the number of actions required to solve any solvable FreeCell instance, as we wanted to prove. ■

The **NP**-completeness proof for FreeCell is once again based on the 3SAT problem. For simplifying the presentation of the reduction, the generated FreeCell instances will not contain tableau piles of equal height. This is not really a problem, because their height could be made equal by adding additional cards of low value to the smaller piles, which can be moved to foundations immediately. For example, in standard FreeCell instances it is never bad to move an ace to a foundation pile, because no card can ever be moved on top of it, so it just uses up space on the tableau or in a free cell. Once all aces are moved to foundations, the same is true for twos, and so on. Indeed, moves of this kind are performed automatically by many FreeCell computer programs.

Theorem 4.26 NP-completeness of PlanEx-FreeCell
PlanEx-FreeCell *is* **NP***-complete, for any fixed number of free cells.*
Proof: Membership in **NP** was shown in the last theorem. Let *cells* be the fixed number of free cells. We will first assume that $cells = 0$ and then explain how the mapping could be adapted for other values of *cells*.

Let $F = (V, C) = (\{v_1, \ldots, v_{|V|}\}, \{\{l_{1,1}, l_{1,2}, l_{1,3}\}, \ldots, \{l_{|C|,1}, l_{|C|,2}, l_{|C|,3}\}\})$ be the 3SAT instance. The positive literal v_i is called *the* $(2i-1)$*-th literal*, written as l_{2i-1}, and the negative literal $\neg v_i$ is called the $2i$-th literal, written as l_{2i}.

For $k \in \{1, \ldots, 2|V|\}$, we define the *cumulated number of occurrences* of literals up to l_k as $cocc(k) = |\{\ (i,j) \in \{1, \ldots, |C|\} \times \{1,2,3\} \mid l_{i,j} = l_{k'}, k' \leq k\ \}|$. We will also talk about $l_{i,j}$ being the m-th occurrence of some literal within C, with the obvious meaning.

Furthermore, we define the *selection value* $val_{SEL} = |C| + 2|V| + 2$ and the *literal value* of the k-th literal as $val_k = val_{SEL} + 2k + 2cocc(k)$, the *clause value* $val_{CL} = val_{2|V|} + 2$ and the *bottom value* $val_{BOT} = val_{CL} + 6|C|$.

F maps to the FreeCell instance characterized by $(val, col, cells, setup)$, where

$$val \overset{\text{def}}{=} val_{BOT} + 4|C| - 2$$
$$col \overset{\text{def}}{=} 6|C| + 2|V| + 2$$
$$cells \overset{\text{def}}{=} 0$$
$$setup(1,1) \overset{\text{def}}{=} (val_{SEL}, \spadesuit)$$

For $i \in \{1, \ldots, |V|\}$:

$$setup(2i,1) \overset{\text{def}}{=} (val_{2i-1}, \spadesuit)$$
$$setup(2i,2) \overset{\text{def}}{=} (val_{SEL} - 2i, \spadesuit)$$
$$setup(2i,3) \overset{\text{def}}{=} (val_{SEL} - 2i + 1, \diamondsuit)$$
$$setup(2i+1,1) \overset{\text{def}}{=} (val_{2i}, \spadesuit)$$
$$setup(2i+1,2) \overset{\text{def}}{=} (val_{SEL} - 2i, \clubsuit)$$
$$setup(2i+1,3) \overset{\text{def}}{=} (val_{SEL} - 2i + 1, \heartsuit)$$

For $i \in \{1, \ldots, |C|\}$, $j \in \{1,2,3\}$, where $l_{i,j}$ is the m-th occurrence of l_k:

$$setup(6i + 2|V| - 5 + j, 1) \overset{\text{def}}{=} (val_{BOT} + 4(i-1), j)$$
$$setup(6i + 2|V| - 5 + j, 2) \overset{\text{def}}{=} (val_{CL} + 6(i-1) + 1, j)$$
$$setup(6i + 2|V| - 5 + j, 3) \overset{\text{def}}{=} (val_k - 2m, \spadesuit)$$
$$setup(6i + 2|V| - 5 + j, 4) \overset{\text{def}}{=} (val_k - 2m + 1, \diamondsuit)$$

For $i \in \{1, \ldots, |C|\}$:

$$\begin{aligned}
setup(6i + 2|V| - 1, 1) &\stackrel{\text{def}}{=} (val_{BOT} + 4(i-1), \clubsuit)\\
setup(6i + 2|V| - 1, 2) &\stackrel{\text{def}}{=} (val_{CL} + 6(i-1) + 4, \diamondsuit)\\
setup(6i + 2|V| - 1, 3) &\stackrel{\text{def}}{=} (val_{CL} + 6(i-1), \diamondsuit)\\
setup(6i + 2|V| + 0, 1) &\stackrel{\text{def}}{=} (val_{BOT} + 4(i-1) + 2, \diamondsuit)\\
setup(6i + 2|V| + 0, 2) &\stackrel{\text{def}}{=} (val_{CL} + 6(i-1) + 4, \heartsuit)\\
setup(6i + 2|V| + 0, 3) &\stackrel{\text{def}}{=} (val_{CL} + 6(i-1), \spadesuit)\\
setup(6i + 2|V| + 1, 1) &\stackrel{\text{def}}{=} (val_{BOT} + 4(i-1) + 2, \heartsuit)\\
setup(6i + 2|V| + 1, 2) &\stackrel{\text{def}}{=} (i, \clubsuit)\\
setup(6i + 2|V| + 1, 3) &\stackrel{\text{def}}{=} (val_{CL} + 6(i-1) + 3, \spadesuit)
\end{aligned}$$

$$setup(6|C| + 2|V| + 2, 4val - (6|V| + 21|C| + 1)) \stackrel{\text{def}}{=} (|C| + 1, \clubsuit)$$

All the other cards are in the last tableau pile, below the specified card, ordered by value such that cards with lower value are closer to the top.

This mapping requires some explanation. Figure 4.2 gives an example of the mapping, which should help understand the idea behind it.

The piles of the initial tableau fall into three groups. The first $2|V| + 1$ piles are called the *literal selection piles*. The next $6|C|$ piles are called the *clausal piles*, organized into groups of six piles that relate to a specific clause, called *clausal groups*. The last pile, holding most of the cards, is called the *big pile*.

We first show that the FreeCell instance can be solved if there is a satisfying assignment to the variables of the logical formula. So assume there is such a satisfying assignment $\alpha \in V \to \{\top, \bot\}$. The FreeCell instance can then be solved as follows:

- For each $i \in \{1, \ldots, |V|\}$, move the top two cards from tableau pile $2i$ to the first tableau pile if $v_i = \top$, or else move the top two cards from tableau pile $2i + 1$ to the first tableau pile if $v_i = \bot$. This releases the bottom cards of some literal selection piles, spades cards which can then be used to move cards from the clausal piles. In the example of Figure 4.2, for the assignment $\{(v_1, \top), (v_2, \top), (v_3, \bot)\}$ these are the 15, 27 and 41 of spades. These are called the *literal choice cards*.

- The first three piles of each clausal group relate to the literals in that clause. The top two cards of such a pile can be moved to the literal selection piles if and only if the literal choice card of the corresponding literal has been revealed.

  Because we have a satisfying truth assignment, it is possible to satisfy a literal in each clause, and thus it is possible to remove the top two cards of one of the first three piles of any clausal group, releasing a card of value $val_{CL} + 6(i-1) + 1$ for the i-th clause. If the first or second literal is true, it will be a red card, and the top card from the fifth pile of the clausal group can be moved on top of it. If the third literal is true, the top card from the fourth pile of the clausal group can be moved on top of it. In either case, a red card of value $val_{CL} + 6(i-1) + 4$ is revealed, and the top card of the sixth pile of the clausal group can be moved on top of it.

- After this has been done for all clauses, the first $|C|$ cards of clubs are available in the sixth piles of the clausal groups and can be moved to foundations, allowing to move the top card of the big pile to foundations. This reveals many low-valued cards, and it is not hard to see that all cards of values up to val_{SEL} can be moved

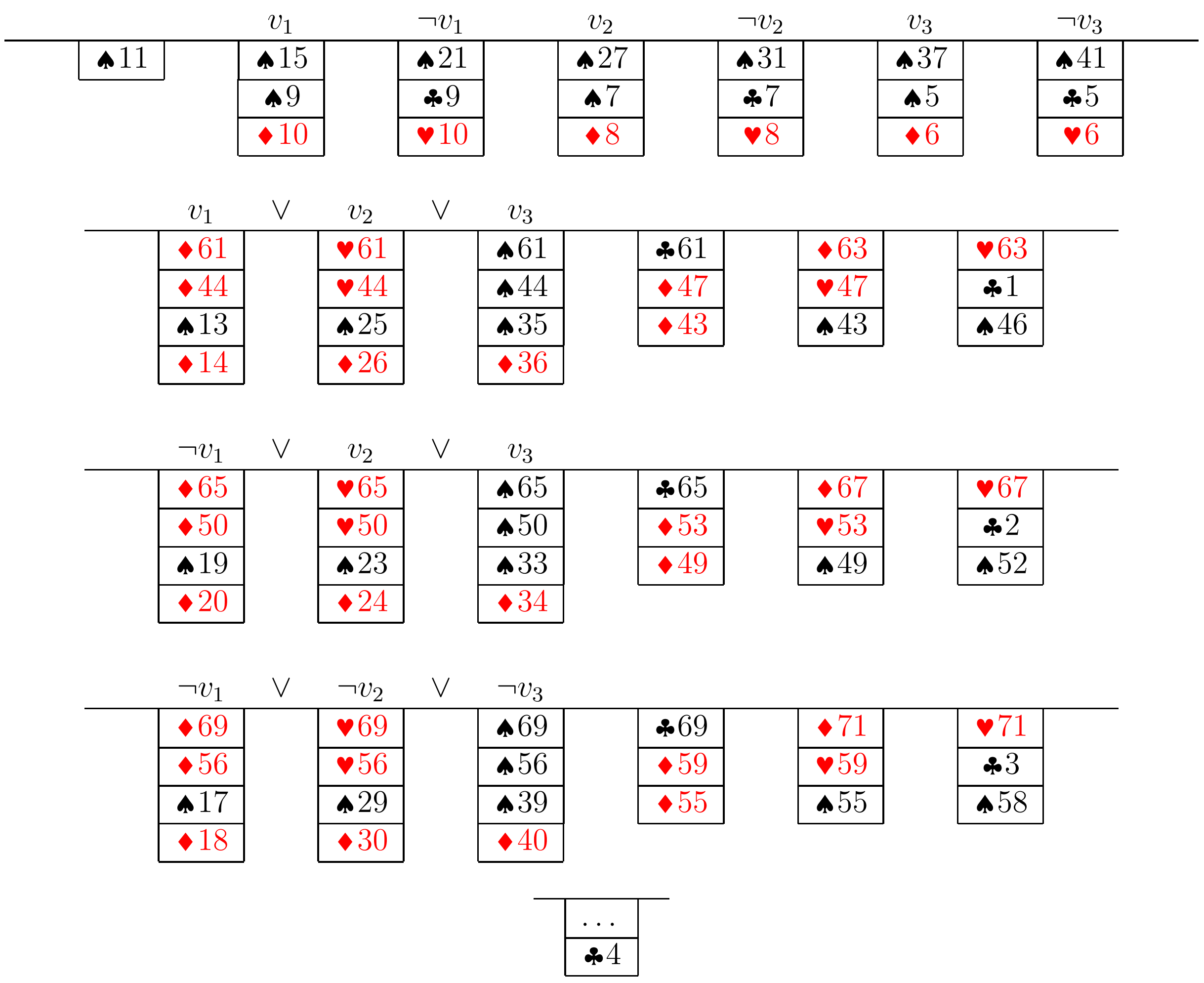


Figure 4.2: FreeCell instance corresponding to the formula with variable set $\{v_1, v_2, v_3\}$ and clause set $\{\{v_1, v_2, v_3\}, \{\neg v_1, v_2, v_3\}, \{\neg v_1, \neg v_2, \neg v_3\}\}$. Cards at the *top* of a pile are depicted nearer the *bottom* of the picture, following the convention of FreeCell implementations on computers. The first group of piles form the literal selection piles, the next three groups are clausal groups, and the big pile is shown at the bottom (most of its cards are omitted).

to foundations immediately. This reveals the literal choice cards of all literals that are false under the chosen assignment, allowing to move the top two cards of the clausal group piles relating to unsatisfied literals to the literal selection piles as well. After this has been done, all piles are ordered by value, with cards of lower value closer to the top, allowing to move all remaining cards to foundations, solving the instance.

Now assume that the FreeCell instance is solvable. It is not possible to move the bottom card of any tableau pile before the top card of the big pile is moved, because all cards that they could be moved on top of are buried in the big pile. Before the top card of the big pile is moved, it is not possible to move the bottom card of any pile to foundations either. This implies that the first movement of the top card of the big pile cannot go to an empty tableau position.

On the other hand, it can not be moved on top of any other card as its first movement, because all possible destination cards are buried under it. Together, this implies that its first (and thus only) movement must be directly to foundations.

This in turn requires all lower-valued clubs cards to be moved to foundations first, requiring movements within the clausal piles. For each clausal group, the top card of the sixth pile must be moved, and it can only be moved to the second card of the fourth or fifth pile, requiring the top card of either of these piles to be moved. These in turn can only be moved on top of the second (counting from the bottom) card from any of the first three piles of that clausal group. Thus, in each clausal group, the top two cards of one of the first three piles must be moved somewhere else for the instance to be solvable.

The only way this can be done is by uncovering the literal choice cards of corresponding literals in the way explained in the first part of the proof. As it is not possible to uncover the literal choice card for v_i and $\neg v_i$ at the same time (for any i), this requires the existence of a satisfying assignments to the truth variables, completing the proof for zero free cells.

If there are more than zero free cells ($cells > 0$), a very similar reduction can be used by ensuring that all free cells must be filled right at the beginning and cannot be cleared before the top card of the big pile is moved to foundations. This is done by first adding $cells \cdot col$ values below the range of the cards specified in the reduction above (i. e., all values of cards in the reduction are increased by $cells \cdot col$) and *cells* values above that range. Now the *cells* lowest-valued cards of clubs are added to the first pile in order, with the ♣1 on top, the next *cells* cards are added to the second pile, and so on. After that, the *highest cells* cards of clubs are added to the first pile, in any order. The new cards of the other suits can be added to the big pile below its original top card, maintaining the property that this subpile is ordered by card value.

If the zero-cell instance was solvable, then this new instance can be solved by moving the cards on top of the ♣1 to the free cells, then moving the first $cells \cdot col$ cards of clubs to foundations and then proceeding as described above.

If the zero-cell instance was not solvable, then the new instance cannot be solved either, because the first *cells* moves *must* go to free cells since the top *cells* cards of each pile are of the same colour and none of them can go to foundations, and no free cell can be released before the original top card of the big pile is moved to foundations. This concludes the proof. ■

With this result we conclude the discussion of FreeCell. Both decision problems for this domain are **NP**-complete. As our proofs show, the hardness in this domain does not (or not only) come from the difficulty in allocating free cells or empty table positions, but rather from the choice of *which* card to move on top of another one when there are two possible choices.

4.7 Assembly

The last domain to be discussed in this chapter is called Assembly and formed part of the ADL track of the AIPS 1998 competition. Although we consider it an important domain within the construction framework, as it combines many of the features discussed before, there are some peculiarities to it which make it difficult to handle.

First of all, although the domain was part of the competition, neither of the two participating ADL planners was able to solve a single competition instance, so any results we could provide on this domain are less valuable because of the lack of empirical data for comparison.

More importantly, some of the competition problems are given in incorrect ADL, making it impossible to feed them to an ADL-capable planning system that should in theory be able to solve them, such as **FF** [Hoffmann and Nebel, 2001]. Specifically, the competition instances 7, 12, 13, 14, 19 and 27 contain errors in their PDDL definitions.

Last and most importantly, there seem to be some severe errors in the definition of the domain itself, namely in the *remove* operator, which in its current state allows completely disassembling a composite object without making it unavailable or incomplete. For example, consider the (impossible) task of creating a rectangular table and an oval table out of four legs, a rectangular board and an oval board. In the domain as specified for the competition, it would be possible to build the rectangular table out of the board and the legs, remove the legs (which leaves the rectangular table in its *complete* state) and then use the legs and the oval board to build the oval table, creating two tables.

There is another minor oddity with regard to resource allocation, which allows to allocate resources such as a voltmeter to an object, incorporate this object into other objects while the resource stays allocated and then deallocate the resource although the object it has been allocated to is no longer available because of being buried inside another object. This oddity does not have any consequences for the complexity of the planning task however, because any "odd" late resource deallocations could easily be moved to a more appropriate earlier place within the plan without affecting its validity.

We will first introduce a corrected and simplified version of the Assembly domain and prove that in this domain, plans can have exponential length. We will then briefly discuss what causes this behaviour and point out differences between the corrected and the original version.

Definition 4.27 Assembly **domain**
The Assembly *domain is the special case of* Construct-$S_2E_2P_0$ *where for all signatures* $\Sigma = (B, B_C, B_D, B^{\mathcal{I}})$ *and instances* $(\Sigma, Pos, init, goal)$*, there is a finite set* M *of* multi-composites*, including a* goal multi-composite m_{goal}*, a function parts* $: M \rightarrow \mathbb{N}_0 \setminus \{0, 1\}$ *and sets of blueprints* B_{Hull}*,* B_{Triv}*,* B_{Ass}*,* B_{Compl} *and* B_{Res} *such that the following is true:*

$$\begin{aligned} B_{Hull} &= \{\ (m, 0) \mid m \in M\ \} \\ B_{Ass} &= \{\ (m, i) \mid m \in M, i \in \{1, \dots, parts(m)\}\ \} \\ B_{Compl} &= \{\ (m, parts(m)) \mid m \in M\ \} \cup B_{Triv} \end{aligned}$$

$$\begin{aligned}
B_{Triv} &\cap (B_{Hull} \cup B_{Ass}) = \emptyset \\
B_{Res} &\cap (B_{Hull} \cup B_{Ass} \cup B_{Triv}) = \emptyset \\
B &= B_{Hull} \cup B_{Triv} \cup B_{Ass} \cup B_{Res} \\
B_C = B_D = \operatorname{dom} B^{\mathcal{I}} &= B_{Ass}
\end{aligned}$$

For $(m,i),(m',i') \in B_{Ass}$, *where* $(m,i)^{\mathcal{I}} = (B_1, B_2, B_E), (m',i')^{\mathcal{I}} = (B'_1, B'_2, B'_E)$*:*

$$\begin{aligned}
B_1 &= \{(m, i-1)\} \\
|B_2| &= 1 \\
B_2 &\subseteq B_{Compl} \setminus \{(m_{goal}, parts(m_{goal}))\} \\
B_2 = B'_2 &\Rightarrow (m,i) = (m',i') \\
B_E &\subseteq B_{Res} \cup B_{Compl} \setminus \{(m, parts(m))\} \\
m = m' &\Rightarrow B_E \cap B_{Res} = B'_E \cap B_{Res} \\
B_E \cap B'_E \cap B_{Compl} \neq \emptyset &\Rightarrow m = m' \wedge \forall i'' \in \{i, \dots, i'\} \textit{ with } (m,i'')^{\mathcal{I}} = (B''_1, B''_2, B''_E) : \\
& \quad\; B_E \cap B'_E \cap B_{Compl} \subseteq B''_E \\
Pos &= \{\ pos_a \mid a \in Atom_\Sigma\ \} \\
init &= \{\ (pos_a, a) \mid a \in Atom_\Sigma\ \} \\
goal &= \{\ (pos_r, r) \mid r \in B_{Res}\ \} \cup \{(pos_{(m_{goal},0)}, o)\} \\
& \quad\; \textit{for } o \in O_\Sigma \textit{ with } \tau_\Sigma(o) = (m_{goal}, parts(m_{goal}))
\end{aligned}$$

In the original ASSEMBLY domain, it is possible to have objects with more than two immediate subparts. In our modelling, we have to introduce intermediary composites for partially built objects to achieve a similar affect. A composite in the original domain relates to a multi-composite in our modelling, and a composite assembly (m,i) refers to an object of (multi-composite) type m where i parts have been attached already. Objects with all subparts attached (including atomic assemblies, which do not have subparts) are called *complete objects* and have types from B_{Compl} (the set of *complete assemblies*).

Rephrasing the above definition in words, we see that instances of ASSEMBLY have the following properties that distinguish them from arbitrary CONSTRUCT instances with unlimited table positions:

- There are four disjoint classes of blueprints, B_{Hull} for the empty “hulls” of composite assemblies (which are implicit in the AIPS definition), B_{Triv} for atomic (trivial) assemblies, B_{Ass} for composite assemblies and B_{Res} for resources. Resources are always atomic and cannot be incorporated into other objects. Hulls are always atomic and form the base parts of composite blueprints (more precisely, the “first part” of a multi-composite object). Trivial assemblies are always atomic. The only composite blueprints are composite assemblies, and these can be composed and decomposed.

- Composite assemblies must have the previous composite assembly of their multi-composite as their base part or the corresponding hull if they correspond to the first subpart of the multi-composite. There is only one allowed type for the added part, and this must be a complete assembly other than the one from the goal multi-composite. This implies that all objects of the same type are equal. Each assembly can form part of at most one other composite.

- Only resources and complete assemblies are used as equipment, and a complete assembly will not be used as equipment for a composite of its own multi-composite. Two assemblies from the same multi-composite require the same resource equipment.

Non-resources (complete assemblies) can only be used as equipment for at most one multi-composite, and if they are needed for two of its composite assemblies, then they are needed for all composite assemblies in between these two in the multi-composite sequence as well.

- In the initial state, one exemplar of each atom (hull, atomic assembly or resource) is located at some table position. Since there is only one instance of each atom and objects of a given type can only be built up in one way, there can be at most one object of any given type in any state. In goal states, in addition to the resource objects, there must be one complete object of the goal multi-composite type, containing all atomic assemblies and hulls as immediate or remote subparts. This goal object is not specified in detail in the above definition because there is only one possible object in O_Σ that is of type $(m_{goal}, parts(m_{goal}))$.

This definition differs somewhat from the Assembly domain as used in the AIPS competition. Apart from the two modelling oddities in the competition version that have been mentioned before and are not present in the domain as defined here, there are two more differences. Firstly, the original Assembly domain allows to specify an arbitrary partial order on the subparts of any multi-composite assembly, and parts can be attached to the multi-composite as long as they adhere to that partial order, whereas our version is more strict and enforces a linear order on subparts. With regard to that property, our domain is a special case of the AIPS domain.

The second difference lies in the use of assemblies as equipment. This is an attempt of modelling the notion of "transient parts" in the original domain, where it might be necessary to temporarily incorporate a given assembly into another assembly, following some partial order constraints. For example, in constructing a "widget" multi-composite, a "tube" might need to be incorporated as a transient part before the "frob" is added and it may not be removed before the "socket" is added. In our modelling, we would require the tube as equipment for all construction steps of the multi-composite from adding the frob to adding the socket. This is an equivalent way of modelling transient parts except for the fact that in our modelling, the transient part can be released and recommitted in between different steps of working on the multi-composite, which is not allowed in the original modelling.

However, since this difference *adds* to the set of applicable actions, it can only lead to *shorter* plans if it makes any difference at all. Thus, our next proof (which shows that plan lengths can grow exponentially in the Assembly domain) still holds in the less flexible (with regard to the allocation of equipment) version of Assembly that we would like to call the *corrected competition version.*

Theorem 4.28 Exponential plans for Assembly
Shortest plan lengths in Assembly *can grow exponentially in the encoding length of the instance, even in the absence of resources.*
Proof: We devise a sequence $(I_n)_{n \in \mathbb{N}_0}$ of instances for which encoding lengths grow polynomially but shortest plan lengths grow exponentially.

For $n \in \mathbb{N}_0$, the multi-composite set M_n and instance signature $\Sigma_n = (B, B_C, B_D, B^{\mathcal{I}})$ are defined as follows:

$$M_n \overset{\text{def}}{=} \bigcup_{i \in \{1,\dots,n\}} \{A_i, B_i\}$$
$$B_{Hull} \overset{\text{def}}{=} \bigcup_{i \in \{1,\dots,n\}} \{(A_i, 0), (B_i, 0)\}$$

$$\begin{aligned}
B_{Triv} &\overset{\text{def}}{=} \bigcup_{i\in\{1,\ldots,n\}}\{x_i, y_i, z_i\} \cup \{(B_0, 2)\}\\
B_{Ass} &\overset{\text{def}}{=} \bigcup_{i\in\{1,\ldots,n\}}\{(A_i, 1), (A_i, 2), (A_i, 3), (B_i, 1), (B_i, 2)\}\\
B &\overset{\text{def}}{=} B_{Hull} \cup B_{Triv} \cup B_{Ass}
\end{aligned}$$

$$B_C \overset{\text{def}}{=} B_D \overset{\text{def}}{=} B_{Ass}$$

$$\begin{aligned}
(A_1, 1)^{\mathcal{I}} &\overset{\text{def}}{=} (\{(A_1, 0)\}, \{x_1\}, \emptyset)\\
(A_i, 1)^{\mathcal{I}} &\overset{\text{def}}{=} (\{(A_i, 0)\}, \{x_i\}, \{(A_{i-1}, 3)\}) \text{ for } i \in \{2, \ldots, n\}\\
(A_1, 2)^{\mathcal{I}} &\overset{\text{def}}{=} (\{(A_1, 1)\}, \{y_1\}, \emptyset)\\
(A_i, 2)^{\mathcal{I}} &\overset{\text{def}}{=} (\{(A_i, 1)\}, \{y_i\}, \{x_{i-1}\}) \text{ for } i \in \{2, \ldots, n\}\\
(A_i, 3)^{\mathcal{I}} &\overset{\text{def}}{=} (\{(A_i, 2)\}, \{(B_{i-1}, 2)\}, \emptyset) \text{ for } i \in \{1, \ldots, n\}\\
(B_i, 1)^{\mathcal{I}} &\overset{\text{def}}{=} (\{(B_i, 0)\}, \{z_i\}, \emptyset) \text{ for } i \in \{1, \ldots, n\}\\
(B_i, 2)^{\mathcal{I}} &\overset{\text{def}}{=} (\{(B_i, 1)\}, \{(A_i, 3)\}, \emptyset) \text{ for } i \in \{1, \ldots, n\}
\end{aligned}$$

For $n \in \mathbb{N}_0$, $n \geq 1$, I_n is defined as the Assembly instance with instance signature Σ_n and goal multi-composite B_n (There is only one such instance, because the instance signature and goal multi-composite define *Pos*, *init* and *goal*). Clearly, encoding length of I_n is polynomial in n, and these are valid Assembly instances.

All of these instances are solvable using the following strategy (not spelled out in all detail): I_1 is trivial to solve, and for instances I_n for $n \geq 2$, we can begin by solving the I_{n-1} subproblem except for the step that creates $(B_{n-1}, 2)$ out of $(B_{n-1}, 1)$ and $(A_{n-1}, 3)$, then commit the table position of $(A_{n-1}, 3)$ to the one holding $(A_n, 0)$ and attach x_n to it to form $(A_n, 1)$, release $(A_{n-1}, 3)$ and completely decompose it (a problem inverse to a subproblem of I_{n-1}), commit the table position of x_{n-1} to the one holding $(A_n, 1)$, attach y_n to it to form $(A_n, 2)$, release x_{n-1}, create $(B_{n-1}, 2)$ by solving another subproblem like I_{n-1}, attach it to $(A_n, 2)$ to form $(A_n, 3)$ and finally attach z_n and then $(A_n, 3)$ to $(B_n, 0)$ to solve the instance.

As for minimal plan length, we only count the number of steps required for building an object of type $(A_n, 3)$, which is a lower bound for minimal plan length because this needs to be done at some point in order to reach a goal state.

For $n = 1$, building this object takes more than zero steps, so it suffices to observe that for any $n \geq 2$, the shortest action sequence for generating an object of type $(A_n, 3)$ from the initial state is at least twice as long as the shortest action sequence for the corresponding problem with $(A_{n-1}, 3)$.

Constructing an object of type $(A_n, 3)$ first requires generating an object of type $(A_{n-1}, 3)$ to be used as equipment for $(A_n, 1)$. At some point after an object of type $(A_n, 1)$ has been generated for the first time, the object of type $(A_{n-1}, 3)$ must be disassembled to obtain an object of type x_{n-1}. For decomposing $(A_{n-1}, 2)$ into $(A_{n-1}, 1)$, an object of type x_{n-2} is required (at least for $n \geq 3$), which can only be obtained by recursively decomposing the object of type A_{n-2}, and so on, so in fact everything except the object of type $(A_n, 1)$ must be decomposed into atoms at that stage. So when an object of type $(B_{n-1}, 2)$ is needed later, it must be assembled from scratch, requiring building an object of type $(A_{n-1}, 3)$ a second time. This proves our claim. ■

Note that the exponential growth of plan length critically relies on the use of non-atomic equipment, called *transient parts* in Assembly terminology. Without transient parts, Assembly becomes a special case of Construct-$S_2E_1P_0$, for which we have proved polynomial plan length and indeed polynomial time plan generation in Theorem 4.21 on page 67.

Our proof of **NP**-hardness for finding short plans in that domain was based on the Blocksworld domain. However, Blocksworld becomes easy if there are no composites in the initial state, as is always the case for Assembly. Another possible source of hardness, resource allocation, is no issue for Assembly without transient parts either, because there is only ever one object of each resource type and resource allocation does not require making any difficult choices. We will assign the name Assembly-Simple to the special case of Assembly without transient parts.

Theorem 4.29 PlanLen-Assembly-Simple $\in$ **P**
PlanLen-Assembly-Simple *can be solved in polynomial time.*
Proof: The following greedy algorithm finds an optimal plan for a given instance. As long as the goal state has not been reached, choose any multi-composite for which all parts are available. Allocate all resources that are required for that multi-composite to the table position of its hull. Add all parts. Deallocation of equipment is done in a lazy fashion, i. e. a piece of equipment is released whenever it is needed at another location or when the object at the table position it is allocated to shall be picked up.

It is obvious that this solves the instance and does not contain any unnecessary actions. This concludes the proof. ■

Basically the same algorithm can be applied in the (corrected or uncorrected) competition version of Assembly if no transient parts are present. The different behaviour in decomposing does not matter here, because no decomposition takes place in optimal plans in the absence of transient parts. The presence of partial orders on subparts does not make the problem any harder. In the uncorrected version, some care must be taken to avoid unnecessary *release* actions because equipment need *not* be released from a table position to be able to pick up the object at that position. Thus, in this domain, equipment should only be released when it is required for another construction.

Note, however that the previous result (Theorem 4.28), while applying to the corrected competition version of Assembly as pointed out before, does *not* hold in the uncorrected competition version if transient parts are allowed. Because objects that have been constructed once in that domain never have to be constructed again (they can be decomposed and still be treated as if they were complete), a given part should only ever be added to and removed from a composite at most once, giving a polynomial bound on the number of *compose* and *decompose* actions, which in turn implies a polynomial bound on the number of other actions in reasonable plans. Indeed, a greedy strategy similar to the one used in the last proof, decomposing an object into its subparts after it has been built completely, can be used to (non-optimally) solve all solvable instances of that version of Assembly.

Because of the limited practical relevance of the competition domain, however, we will not discuss this issue further and conclude this section and indeed our discussion of construction domains.

4.8 Summary

We have seen a number of quite different planning domains in this chapter, and it is worth recapitulating the results. We mentioned in the introductory section that our family of manipulation domains is far less homogenous than the transportation family we have studied in the previous chapter, and indeed this shows in the results. Whereas in all but the most trivial transportation domains (namely Gripper), finding optimal plans was an **NP**-equivalent problem and finding any plan was either a polynomial problem or **NP**-equivalent, there is a greater variety in results for the domains within the manipulation framework.

We first observed that the PlanEx and PlanLen problem for manipulation domains becomes **PSPACE**-complete as soon as both transformations and composites are allowed, i. e. for all domains in the hierarchy that fall between Manipulate-$S_1T_1E_0P_0$ and the most general Manipulate-$S_2T_1E_2P_1$. This implies the presence of exponentially long plans in these domains, and in fact there are several other conditions under which plan lengths can grow exponentially, summarized in Figure 4.3.

Plan lengths in Manipulate-$S_iT_jE_kP_l$ can grow exponentially with the encoding length of instances in any of these cases:

- There are both composite objects and transformations in the domain (i. e., $i \geq 1$ and $j = 1$). In fact, deciding PlanEx for these domains is **PSPACE**-complete (Theorems 4.8 and 4.9).
- Arbitrary equipment is allowed, in either a transformation or construction domain (i. e., $k = 2$ and either $i \geq 1$ or $j = 1$). This is also true in the Assembly domain, which is a special case of Construct-$S_2E_1P_0$ (Theorems 4.14, 4.20 and 4.28).
- In a construction domain, the number of table positions is limited (i. e. $i \geq 1$ and $l = 1$). However, in FreeCell, which is also a construction domain with limited table positions, this is not the case (Theorems 4.19 and 4.25).

In all other domains we have discussed, plan lengths can be bounded by a polynomial. This is obvious for domains featuring neither composites nor transformations and has been proved for the other cases, special cases of Transform-E_1, the Schedule domain and special cases of Construct-$S_2E_1P_0$ (Theorems 4.11, 4.18 and 4.21).

Figure 4.3: Manipulation domains and plan length.

In most of these cases, plans can get long because it is possible to enforce that basically the same subproblem must be solved twice because of resource requirements. If table positions are not an issue and the available equipment cannot change through the execution of actions, this kind of construction is not possible, and plan lengths become polynomial.

Indeed, instances of the latter kind can usually be solved by a greedy strategy, making monotonic progress towards the goal, and for all of these generating any plan is a problem that can be solved in polynomial time. This is true for Transform-E_1, Schedule and Construct-$S_2E_1P_0$, and in fact we have only encountered one domain where generating a plan is an **NP**-equivalent problem, FreeCell.

Of the domains for which generating a plan can be done in polynomial time, only the most simple ones allow finding *optimal* plans in polynomial time unless **P** = **NP**. The only construction domain for which this is possible is Assembly-Simple, and there are two transformation domains, Transform-E_0 and Schedule, which share that property. For the other domains with easy plan generation, finding optimal plans is either made hard because of the scheduling of equipment, as in Transform-E_1, or the ordering of subgoals, as in Blocksworld.

In the three easy domains, these two issues do not arise because subgoals can be handled independently (as in Transform-E_0 or Schedule) or any reasonable ordering is equally good (as in Assembly-Simple), and equipment is either non-existent (as in Transform-E_0), easy to schedule because the correct order of composition is obvious (as in Assembly-Simple), or the cardinality of objects to be used as equipment is fixed (as in Schedule).

Figure 4.4 summarizes these results.

In the manipulation domains with polynomial plan lengths we have investigated, the status of the PlanEx problem is as follows:

- It is **NP**-complete for FreeCell (Theorem 4.26).
- Plans can be generated in polynomial time in all other cases, namely Transform-E_1, Schedule and Construct-$S_2E_1P_0$ (Theorems 4.11, 4.18 and 4.21).

Of course, this implies that PlanLen-FreeCell is **NP**-complete. For the other domains, the status of PlanLen is as follows:

- PlanLen-Transform-E_1 is **NP**-complete (Theorem 4.13).
- PlanLen-Blocksworld, and thus PlanLen for any domain in the hierarchy between Construct-$S_2E_1P_0$ and Construct-$S_1E_0P_0$, is **NP**-complete (Theorem 4.23).
- Optimal plans in Transform-E_0 and Schedule can be generated in polynomial time (Theorems 4.10 and 4.18).
- Optimal plans in Assembly-Simple can be generated in polynomial time (Theorem 4.29).

Figure 4.4: Complexity of manipulation domains with polynomial plan length.

The results for the domains of the AIPS competitions are summarized in Figure 4.5. Figure 4.6 summarizes the generalization/specialization relationship between the domains discussed in this chapter and recapitulates our results, with page references.

Before closing the chapter, we want to briefly raise the question whether allowing for parallel activity makes any difference in the manipulation domains we discussed. Certainly this can only be the case for domains with polynomial length plans for which PlanEx is not already **NP**-complete.

Domain name	PlanEx	PlanLen
Assembly	exponential length plans	
Blocksworld	polynomial	**NP**-complete
FreeCell	**NP**-complete	**NP**-complete
Schedule	polynomial	polynomial

Figure 4.5: Complexity results for the manipulation domains from the AIPS 1998 and AIPS 2000 planning competitions. Note that the Assembly result only applies to the corrected version of the domain.

For transformation domains (including Schedule), this is not the case, as has already been mentioned in Section 4.4. For construction domains, the way we introduced them, there cannot be any parallel activity except for the parallel allocation and deallocation of equipment, which does not make a difference for Blocksworld, so the results for the Construct family are the same as well, with the possible exception of Assembly-Simple which might actually be harder to solve optimally in parallel because the resource allocation subproblem becomes more involved.[6]

However, reformulating the domain in a way that allows for direct movement between table positions rather than separate pickup and drop actions could allow for an additional kind of parallelism, and it is indeed not hard to think of an alternative version of Blocksworld using this formalism for which optimal parallel plans could be generated in polynomial time. However, this affects only a very small branch of the Manipulate family, namely those domains with unlimited table positions that do not require any equipment at all (Construct-$S_1E_0P_0$ and Construct-$S_2E_0P_0$).[7]

[6]For the competition version of Assembly without transient parts, this is the case. Optimal plans can be found in polynomial time there, but deciding the existence of a parallel plan of bounded length is an **NP**-complete problem, even if the number of resource objects is fixed (to be three). This can be proved by a reduction of the **NP**-complete Precedence Constrained Scheduling problem. However, this proof requires allowing arbitrary partial orders for subparts of an object and thus does not carry over to our version of Assembly-Simple.

[7]We claim that parallel plan length for Construct-$S_1E_1P_0$ is **NP**-complete, which can be proved by a reduction of the strongly **NP**-complete 3-Partition problem. However, as it requires more effort than we are willing to spend here to define the parallel Construct-$S_1E_1P_0$ domain, we will not go into detail.

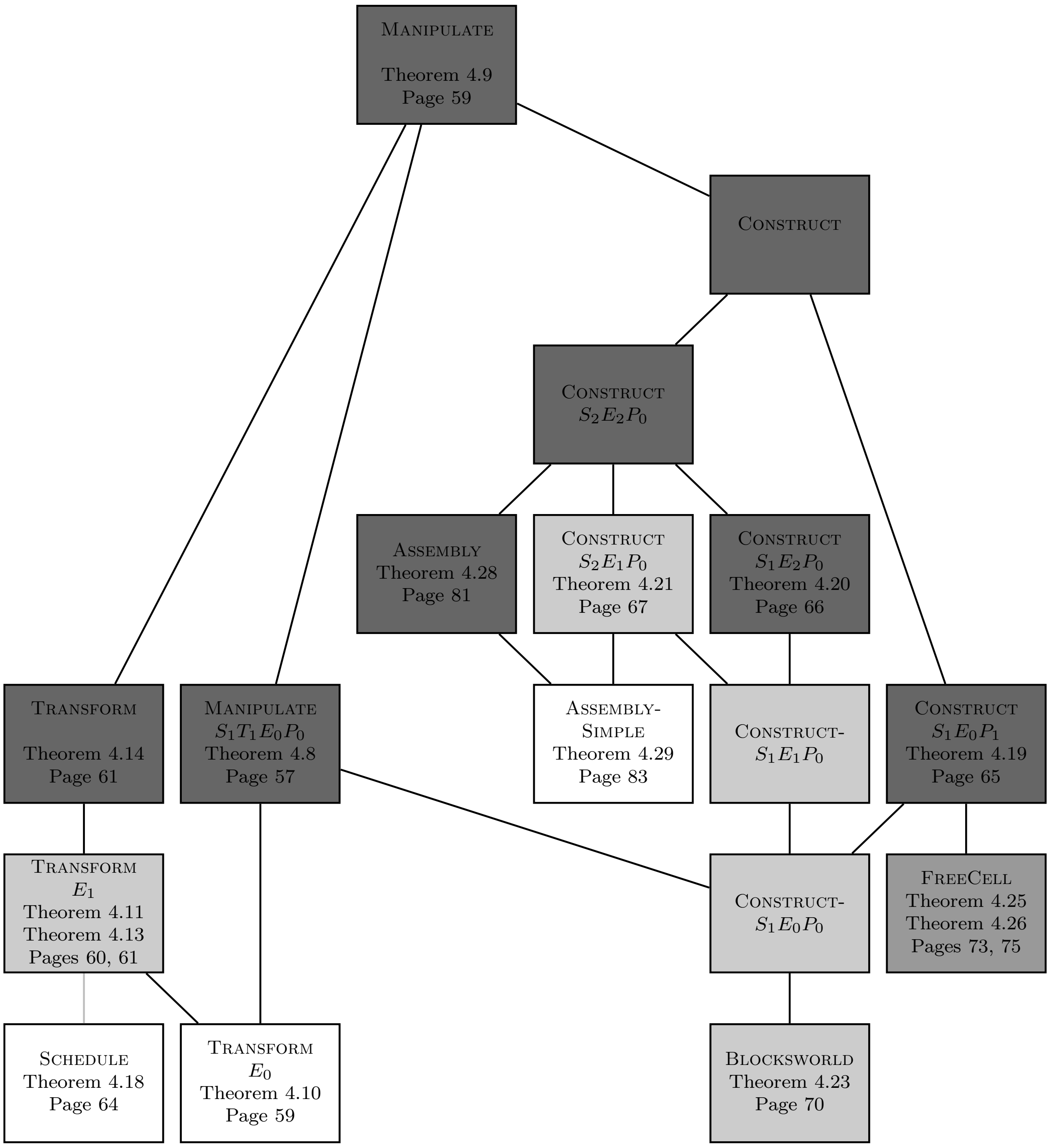


Figure 4.6: The manipulation domains hierarchy. Domains further up in the graph are more general than their descendants. A white box is used for domains for which the decision (and search) problems are polynomial. A light grey box indicates that plans can be generated in polynomial time, but the PLANLEN problem is **NP**-complete. For domains in medium grey boxes, both decision problems are **NP**-complete. In domains with dark grey boxes, plans can be exponentially long. The domains linked with a light grey line are similar, but neither is more or less general than the other (cf. Section 4.4).

Chapter 5

Conclusions

In this thesis, we defined two planning domains called Transport and Manipulate and analyzed the computational complexity of deciding the PlanEx and PlanLen decision problems with regard to these two domains. For both of them we defined special cases which we considered particularly relevant, analyzed their complexity and related the results to some well-known planning domains that formed part of the AIPS 1998 and 2000 competitions.

One of our intentions in looking at hierarchies of planning domains rather than isolated decision problems was to find the boundary between easy and hard problems within the general domains. For Transport, this boundary is clearly defined: For domains with restricted fuel, it is hard to generate plans, whereas for domains with unlimited fuel, solving this task it easy. One possible explanation why fuel restrictions make these problems harder is that, by bounding the number of movements that can be performed, they effectively limit the length of the generated plans, and hence the problem of just finding *any* plan becomes akin to the problem of finding an *optimal* plan.

This problem is hard to solve for all but the most trivial special cases of Transport (i. e., for all except Gripper). The main source of this hardness that we could identify is a difficulty in *ordering*: While it is evident how individual portables should be delivered, the interactions between different portables make optimally solving the overall task a hard problem.

The previous two paragraphs cover our results for the competition domains Logistics, Gripper, Mystery, Mystery' and Grid. In the Grid domain, we were able to identify another source of hardness in the decision of which of the different existing possibilities to clear the (locked) path to the goal locations should be pursued.

The Miconic-10 elevator domain, which is the only transportation domain we analyzed where fuel is not restricted and yet generating plans is hard, shows how side effects of actions can make a greedy strategy inappropriate. Specifically, the idea of picking up one person after the other and moving them to their goal destination does not work because it is not possible to keep other people (who we do not intend to move to their destination yet) from boarding the elevator, and their boarding can render the task of moving the other passengers to their destinations more difficult or impossible because of access restrictions or required attendance.

Because of its greater diversity, the boundary of hardness for Manipulate is a bit harder to identify. Plans in the Manipulate domain mainly build on two kinds of operations: composition/decomposition and transformation. We observed that if both are present in a domain, deciding the existence of a plan is very hard (**PSPACE**-complete) and hence focused our analysis on the cases that contain either of these two kinds of operations, but not both. The resulting subfamilies were called Construct and Transform.

A difficulty that arose repeatedly within these domains was the existence of planning instances for which shortest lengths are exponentially long in the size of the instance encoding. Like in the well-known Towers of Hanoi problem, this difficulty arises in the presence of limited table positions, but it also occurs as long as there are no restrictions on the kind of equipment requirements that may be specified in the domain. This is true for both Construct and Transform. While none of the competition domains explicitly models this kind of equipment requirements, the "transient parts" in the Assembly domain are related to this problem, and we showed that a corrected version of the Assembly domain features exponentially long plans.

In the only competition domain with restricted table positions, FreeCell, plan lengths can be bounded by a polynomial in the number of cards. However, the scarceness of free positions contributes to the hardness of this problem, too. Of all the manipulation problems with polynomially long plans we investigated, it is the only one for which generating a plan is an **NP**-equivalent problem, and this is because of the limited number of table positions, since removing this constraint would make it a special case of the Construct-$S_1E_0P_0$ domain, for which plans can be generated in polynomial time.

Indeed, we showed that for all special cases of Construct with unlimited table positions and without arbitrary equipment requirements, plans can be generated in polynomial time. Because of a well-known result for the Blocksworld domain, we also know that finding optimal plans is an **NP**-equivalent problem in these domains.

For the branch of the Manipulate hierarchy containing the transformation domains, we already mentioned that allowing for general equipment can lead to an explosion of plan lengths. If equipment is not being generated or destroyed itself, then plans can be generated in polynomial time. However, finding an optimal plan is still hard, except if no equipment is needed at all.

The Schedule domain used in the AIPS 2000 competition lies somewhere in between those two extremes, since it allows for equipment, but the objects to be used as equipment are fixed for the domain rather than being part of the individual instances. This leads to a complexity somewhere between these two cases of Transform: Although optimal plans for Schedule can be generated in polynomial time, the algorithm we could provide is not really practical, and we will not go so far as to say that generating optimal plans for Schedule is an easy problem.

So if we were to describe the border between easy and hard problems in both families in one sentence, we could say that transportation domains are mainly difficult in the presence of fuel constraints, and manipulation domains are mainly difficult when limited table positions and complex procurement of equipment come into play. However, we think that this summary would be missing the point, because we believe the real divide to be somewhere else, namely between PlanGen and PlanOpt.

For many relevant domains, including all variants of Transport without fuel and a big subhierarchy of Construct, there is a difference in complexity between the problem of generating any plan and generating optimal plans. This is interesting to note, because it is a point that must not be neglected when evaluating planning algorithms. Optimal and non-optimal planning systems cannot be compared to one another in terms of performance in a meaningful way, because they are solving different problems. While this fact is by no means new, it is interesting to note that it actually applies to many of the benchmark domains that are regularly used for comparing planning systems.

Moreover, this helps explain why planning systems based on local search, most notably the **FF** system, are doing so much better than ones based on **Graphplan**, generating optimal parallel plans, which we have seen to be just as hard a task as finding optimal sequential plans in most cases.

We want to close our conclusions with this remark. While we have answered some questions in the previous chapters, other open issues remain. In some cases it would be interesting to enlarge the hierarchies of domains we have analyzed by adding more special cases. For example, we showed that plans can be generated in polynomial time in the simplified Miconic-10 domain, but the corresponding problem is **NP**-equivalent if all types of special passengers and access restrictions from the full domain are allowed. But what is the complexity of the problem if only *some* of the enhancements are made?

As a second point, for domains for which generating plans is **NP**-equivalent, it would be interesting to discover what hard instances look like, trying to discover a *phase transition* between (usually easy) under-constrained and (usually easy) over-constrained instances, and if it can be found, characterize the part of the instance space containing the "hard" instances.

As a last point, the distinction between PlanGen and PlanOpt is quite coarse, and for the domains which exhibit differences in complexity for these two, the question arises if it is possible to find *good* (if not optimal) plans in polynomial time, for example plans that are guaranteed not to exceed the length of optimal ones by more than a constant factor. It is evident that such *performance guarantees* are not hard to give in Logistics or Blocksworld, but what about Grid?

We close the chapter with the following figure, which looks suspiciously like the one we started out with in Chapter 1, but with one column added that makes all the difference:

Year	Domain name	Domain family	Complexity
1998	Assembly	Construction	exponential
	Grid	Transportation	polynomial/**NP**-equivalent
	Gripper	Transportation	polynomial
	Logistics	Transportation	polynomial/**NP**-equivalent
	Movie	Other	polynomial
	Mystery	Transportation	**NP**-equivalent
	Mystery'	Transportation	**NP**-equivalent
2000	Blocksworld	Construction	polynomial/**NP**-equivalent
	FreeCell	Construction	**NP**-equivalent
	Logistics	Transportation	polynomial/**NP**-equivalent
	Miconic-10	Transportation	**NP**-equivalent
	Schedule	Transformation	polynomial

Figure 5.1: Domains from the AIPS 1998 and AIPS 2000 planning competitions. For more detailed results, see Figure 3.9 on page 45 and Figure 4.6 on page 87.

Bibliography

[Allen et al., 1990] Allen, J., Hendler, J., and Tate, A., editors (1990). *Readings in Planning*. Morgan Kaufmann.

[Bacchus, 2001] Bacchus, F. (2001). The AIPS'00 planning competition. *AI Magazine*, 22(3):47–56.

[Blum and Furst, 1997] Blum, A. and Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):281–300.

[Bonet and Geffner, 2000] Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In Chien, S., Kambhampati, S., and Knoblock, C. A., editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 52–61. AAAI Press.

[Bylander, 1994] Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204.

[Erol et al., 1995] Erol, K., Nau, D. S., and Subrahmanian, V. S. (1995). Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):65–88.

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman.

[Gupta and Nau, 1992] Gupta, N. and Nau, D. S. (1992). On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2–3):223–254.

[Hoffmann and Nebel, 2001] Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

[Koehler and Schuster, 2000] Koehler, J. and Schuster, K. (2000). Elevator control as a planning problem. In Chien, S., Kambhampati, S., and Knoblock, C. A., editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 331–338. AAAI Press.

[Lifschitz, 1987] Lifschitz, V. (1987). On the semantics of STRIPS. In Georgeff, M. and Lansky, A., editors, *Reasoning about Actions and Plans*, pages 1–9. Morgan Kaufmann.

[Long and Fox, 2000] Long, D. and Fox, M. (2000). Automatic synthesis and use of generic types in planning. In Chien, S., Kambhampati, S., and Knoblock, C. A., editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 196–205. AAAI Press.

[Long et al., 2000] Long, D., Kautz, H., Selman, B., Bonet, B., Geffner, H., Koehler, J., Brenner, M., Hoffmann, J., Rittinger, F., Anderson, C. R., Weld, D. S., Smith, D. E., and Fox, M. (2000). The AIPS-98 planning competition. *AI Magazine*, 21(2):13–33.

[McDermott, 2000] McDermott, D. (2000). The 1998 AI Planning Systems competition. *AI Magazine*, 21(2):35–55.

[Mehlhorn and Näher, 1999] Mehlhorn, K. and Näher, S. (1999). *LEDA — A Platform for Combinatorial and Geometric Computing.* Cambridge University Press.

[Selman, 1994] Selman, B. (1994). Near-optimal plans, tractability, and reactivity. In Doyle, J., Sandewall, E., and Torasso, P., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR'94)*, pages 521–529. Morgan Kaufmann.