The Fast Downward Planning System

Malte Helmert HELMERT@INFORMATIK.UNI-FREIBURG.DE
Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee, Gebäude 052
79110 Freiburg, Germany

Abstract

Fast Downward is a classical planning system based on heuristic search. It can deal with general deterministic planning problems encoded in the propositional fragment of PDDL2.2, including advanced features like ADL conditions and effects and derived predicates (axioms). Like other well-known planners such as HSP (Bonet & Geffner, 2001) and FF (Hoffmann & Nebel, 2001), Fast Downward is a progression planner, searching the space of world states of a planning task in the forward direction. Unlike most other heuristic planners, whose state evaluations are based on ignoring negative interactions of operators, Fast Downward uses hierarchical decompositions of planning tasks for computing its heuristic function, which we call the *causal graph heuristic*. This approach has proven remarkably successful: Fast Downward won the "classical" (i. e. propositional, non-optimizing) track of the 4th International Planning Competition at ICAPS 2004, following in the footsteps of planners such as FF and LPG (Gerevini, Saetti, & Serina, 2003).

This article provides a detailed overview of Fast Downward's architecture and the underlying algorithms. The technical part explains the three phases of the planner: During *translation*, the propositional input task is transformed into a representation with multi-valued state variables more amenable to hierarchical decomposition. During *knowledge compilation*, the crucial information of the translated planning task is compiled into a number of efficient data structures. Finally, during *search*, the causal graph heuristic is used in a best-first search framework to compute a solution plan.

Some building blocks of the Fast Downward planning system have been presented in previous work (Edelkamp & Helmert, 1999; Helmert, 2004). We extend these techniques originally tailored to STRIPS tasks to the full generality of propositional PDDL and present some genuinely new algorithms for fast operator instantiation (*Horn exploration*), search control (*preferred operators*, *multi-heuristic best-first search*), and a new alternative search algorithm which uses the information encoded in the causal graph and domain transition graphs in a novel way (*focused iterative-widening search*). Our experimental data shows the merits of these techniques for the standard benchmark suite, comparing various configurations of Fast Downward to one another and to other efficient planning systems.

1. Introduction

Consider a typical transportation planning task: The postal service must deliver a number of parcels to their respective destinations using its vehicle fleet of cars and trucks. Let us assume that a car serves all the locations of one city, and that different cities are connected via highways that are served by trucks. For simplicity's sake, let us further assume that traveling on each segment of road or highway incurs the same cost. This is not a highly realistic assumption, but for the purposes of exposition, it will do. There can be any number of parcels, posted at arbitrary locations and with arbitrary destinations. Moreover, cities can be of varying size, there can be one or several cars

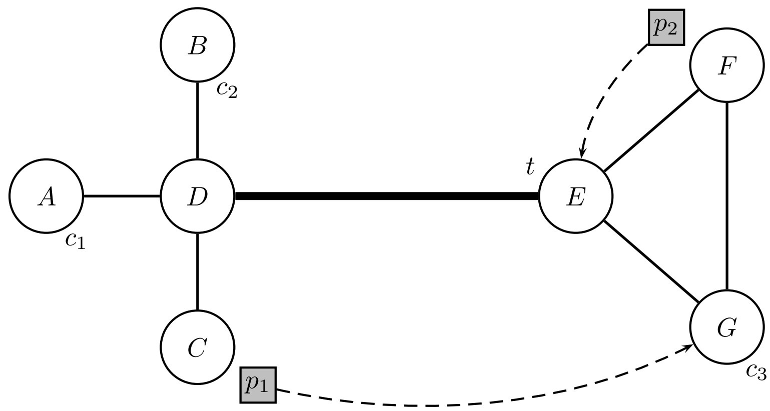

Figure 1: A transportation planning task. Deliver parcel p_1 from C to G and parcel p_2 from F to E, using the cars c_1, c_2, c_3 and truck t. The cars may only use inner-city roads (thin edges), the truck may only use the highway (thick edge).

within each city, and there can be one or several trucks connecting the cities. Cars will never leave a city. Fig. 1 shows an example task of this kind with two cities, three cars and a single truck. There are two parcels to be delivered, one of which (p_1) must be moved between the two cities, while the other (p_2) can stay within its initial city.

The astute reader familiar with the planning literature will have noticed by now that we are essentially describing the LOGISTICS domain, a standard benchmark for classical planning systems, extended to roadmaps that are not complete graphs. (Part of) a propositional STRIPS-like encoding of the task is shown in Fig. 2.

How would a human planner go about solving tasks of this kind? Very likely, they would use a hierarchical approach: For p_1, it is clear that the parcel needs to be moved between cities, which is only possible by using the truck. Since in our example each city can access the highway at only one location, we see that we must first load the parcel into some car at its initial location, then drop it off at the first city's highway access location, load it into the truck, drop it off at the other city's highway access location, load it into the only car in that city, and finally drop it off at its destination. We can commit to this "high-level" plan for delivering p_1 without worrying about "lower-level" aspects such as path planning for the cars. It is obvious to us that *any* good solution will have this structure, since the parcel can only change its location in a few clearly defined ways (Fig. 3). The same figure shows that the only reasonable plans for getting p_2 to its destination require loading it into the car in its initial city and dropping it off at its target location. There is no point in ever loading it into the truck or into any of the cars in the left city.

So say we have committed to the (partially ordered, as movements of the two parcels can be interleaved) "high-level plan" shown in Fig. 5. All we need to do to complete the plan is choose a linearization of the high-level steps and fill in movements of the vehicle fleet between them. We have thus decomposed the planning task into a number of subproblems. The parcel scheduling problem (where, and by which vehicles, a parcel should be loaded and unloaded) is separated from the path planning problem for each vehicle in the fleet (how to move it from point X to Y). Both

```
Variables:
  at-p1-a at-p1-b at-p1-c at-p1-d at-p1-e at-p1-f at-p1-g
  at-p2-a at-p2-b at-p2-c at-p2-d at-p2-e at-p2-f at-p2-g
  at-c1-a at-c1-b at-c1-c at-c1-d
  at-c2-a at-c2-b at-c2-c at-c2-d
  at-c3-e at-c3-f at-c3-g
  at-t-d at-t-e
  in-p1-c1 in-p1-c2 in-p1-c3 in-p1-t
  in-p2-c1 in-p2-c2 in-p2-c3 in-p2-t
Init:
  at-p1-c at-p2-f at-c1-a at-c2-b at-c3-g at-t-e
Goal:
  at-p1-g at-p2-e
Operator drive-c1-a-d:
  PRE: at-c1-a  ADD: at-c1-d  DEL: at-c1-a
Operator drive-c1-b-d:
  PRE: at-c1-b  ADD: at-c1-d  DEL: at-c1-b
Operator drive-c1-c-d:
  PRE: at-c1-c  ADD: at-c1-d  DEL: at-c1-c
...
Operator load-c1-p1-a:
  PRE: at-c1-a at-p1-a  ADD: in-p1-c1  DEL: at-p1-a
Operator load-c1-p1-b:
  PRE: at-c1-b at-p1-b  ADD: in-p1-c1  DEL: at-p1-b
Operator load-c1-p1-c:
  PRE: at-c1-c at-p1-c  ADD: in-p1-c1  DEL: at-p1-c
...
Operator unload-c1-p1-a:
  PRE: at-c1-a in-p1-c1  ADD: at-p1-a  DEL: in-p1-c1
Operator unload-c1-p1-b:
  PRE: at-c1-b in-p1-c1  ADD: at-p1-b  DEL: in-p1-c1
Operator unload-c1-p1-c:
  PRE: at-c1-c in-p1-c1  ADD: at-p1-c  DEL: in-p1-c1
...
```

Figure 2: Part of a typical propositional encoding of the transportation planning task (no actual PDDL syntax).

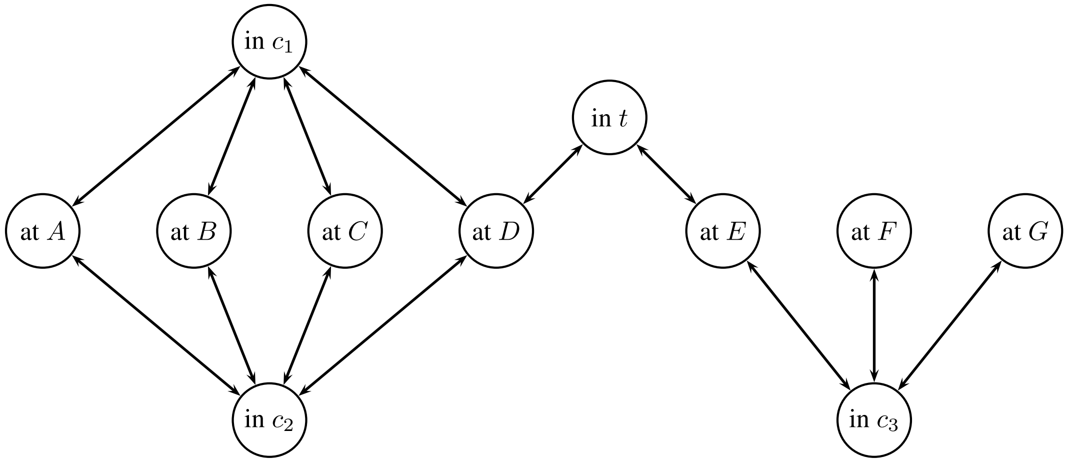

Figure 3: Domain transition graph for the parcels p_1 and p_2. Indicates how a parcel can change its state. For example, the arcs between "at D" and "in t" correspond to the actions of loading/unloading the parcel at location D with the truck t.

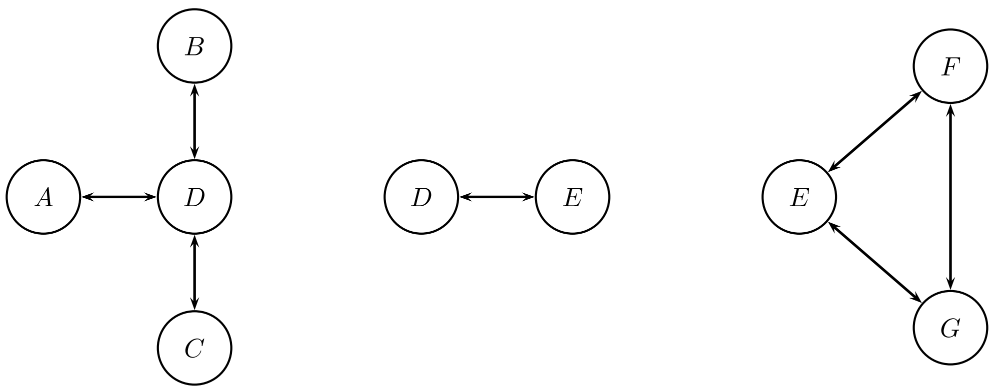

Figure 4: Domain transition graphs for the cars c_1 and c_2 (left), truck t (center), and car c_3 (right). Note how these correspond to the parts of the roadmap that can be traversed by the respective vehicles.

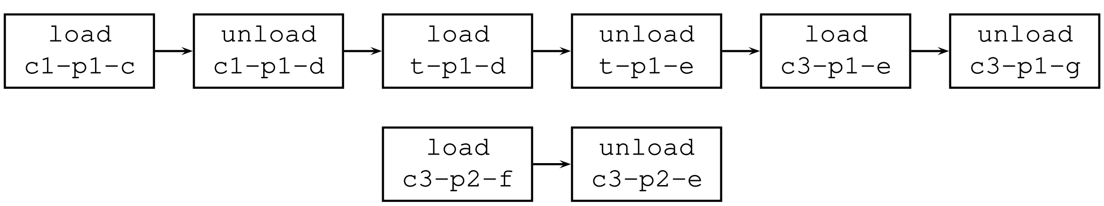

Figure 5: High-level plan for the transportation planning task.

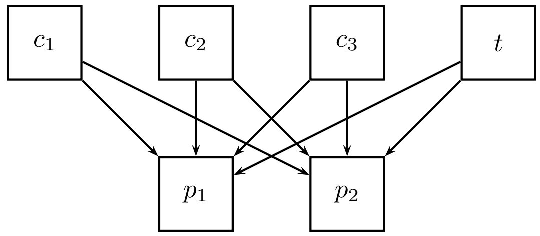

Figure 6: Causal dependencies in the transportation planning task.

of these are graph search problems, and the corresponding graphs are shown in Fig. 3 and Fig. 4. Graphs of this kind will be formally introduced as *domain transition graphs* in Section 5.

Of course these graph search problems interact, but they only do so in limited ways: State transitions for the parcels have associated conditions regarding the vehicle fleet, which need to be considered in addition to the actual path planning in Fig. 3. For example, a parcel can only change state from "at location A" to "inside car c_1" if the car c_1 is at location A. However, state transitions for the vehicles have no associated conditions from other parts of the planning task, and hence moving a vehicle from one location to another is indeed as easy as finding a path in the associated domain transition graph. We say that the parcels have *causal dependencies* on the vehicles because there are operators that change the state of the parcels and have preconditions on the state of the vehicles. Indeed, these are the only causal dependencies in this task, since parcels do not depend on other parcels and vehicles do not depend on anything except themselves (Fig. 6). The set of causal dependencies of a planning task is visualized in its *causal graph*.

We argue that humans often solve planning tasks in the hierarchical fashion outlined in the preceding paragraphs, and that algorithmic approaches to action planning can usefully apply similar ideas. Indeed, we are not the first to introduce the notions of domain transition graphs and causal graphs. Domain transition graphs were introduced by Jonsson and Bäckström in the context of SAS$^+$ planning (1998a). Very similar concepts to causal graphs have been used in the context of hierarchical problem solving by Knoblock (1994) and Bacchus and Yang (1994), who automatically generate abstractions for hierarchical planning algorithms. Their approaches aim at improving the search control of PRODIGY-style planners, a very different goal to the topic of our work, but the references show that the underlying concept of causal graphs can be put to a number of purposes in domain-independent planning.

More closely related to our work, a number of researchers have studied causal graphs for hierarchical planning in STRIPS-style domains: Jonsson and Bäckström (1998b) call them "dependency graphs" and use them to identify a class of planning problems for which plan existence is easy to decide but plan generation is intractable because shortest plans can be exponentially long. Williams and Nayak (1997), and more recently Domshlak and others (Domshlak & Dinitz, 2001; Domshlak & Brafman, 2002), coined the term "causal graphs", using them to prove the tractability and intractability of certain kinds of restricted planning problems.

The latter four articles all focus on *acyclic* causal graphs, and for a good reason: If the causal graph of a planning task exhibits a cycle, hierarchical decomposition is not possible, because the subproblems that must be solved to achieve an operator precondition are not necessarily smaller than the original task. As far as we are aware, we were the first to utilize causal graphs in the context of

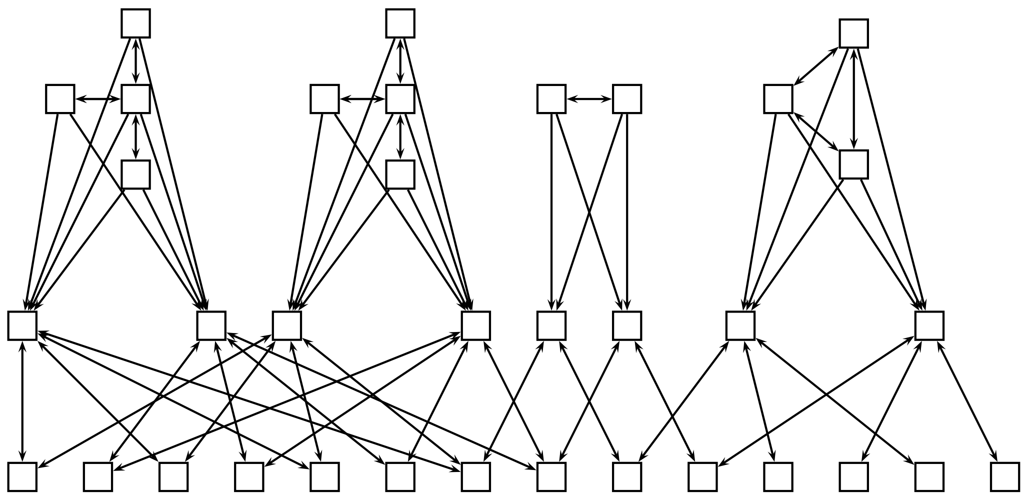

Figure 7: Causal graph for the STRIPS encoding of the transportation planning task.

general PDDL-style planning (Helmert, 2004), as the existing literature considers the acyclic case exclusively. However, our *causal graph heuristic* also requires acyclicity; in the general case, it considers a relaxed planning problem in which some operator preconditions are ignored to break causal cycles.

Knowing that cycles in causal graphs are undesirable, we take a closer look at the transportation planning task. Let us recall our informal definition of causal graphs: The causal graph of a planning task contains a vertex for each state variable and arcs from variables that occur in preconditions to variables that occur in effects of the same operator. So far, we may have given the impression that the causal graph of the example task has the well-behaved shape shown in Fig. 6. Unfortunately, having a closer look at the STRIPS encoding in Fig. 2, we see that this is not the case: The correct causal graph, shown in Fig. 7, looks very messy. This discrepancy between the intuitive and actual graph is due to the fact that in our informal account of "human-style" problem solving, we made use of (non-binary) state variables like "the location of car c_1" or "the state of parcel p_1", while STRIPS-level state variables correspond to (binary) object-location propositions like "parcel p_1 is at location A". It would be much nicer if we were given a multi-valued encoding of the planning task that explicitly contains a variable for "the location of car c_1" and similar properties. Indeed, the nice-looking acyclic graph in Fig. 6 is the causal graph of the multi-valued encoding shown in Fig. 8.

Having provided some intuition for its underlying concepts, let us now state our design goal for the Fast Downward planning system: *To develop an algorithm that efficiently solves general propositional planning tasks by exploiting the hierarchical structure inherent in their causal graphs.* We need to overcome three major obstacles in this undertaking:

- First, propositionally encoded planning tasks usually have very unstructured causal graphs. The intuitive dependencies often become visible in encodings with multi-valued state variables. To exploit this fact in an automated planner, we devise an automatic translation algorithm.

```
Variables:
  p1, p2 ∈ {at-a, at-b, at-c, at-d, at-e, at-f, at-g,
            in-c1, in-c2, in-c3, in-t}
  c1, c2 ∈ {at-a, at-b, at-c, at-d}
  c3     ∈ {at-e, at-f, at-g}
  t      ∈ {at-d, at-e}
Init:
  p1 = at-c, p2 = at-f
  c1 = at-a, c2 = at-b, c3 = at-g, t = at-e
Goal:
  p1 = at-g, p2 = at-e
Operator drive-c1-a-d:
  PRE: c1 = at-a  EFF: c1 = at-d
Operator drive-c1-b-d:
  PRE: c1 = at-b  EFF: c1 = at-d
Operator drive-c1-c-d:
  PRE: c1 = at-c  EFF: c1 = at-d
...
Operator load-c1-p1-a:
  PRE: c1 = at-a, p1 = at-a  EFF: p1 = in-c1
Operator load-c1-p1-b:
  PRE: c1 = at-b, p1 = at-b  EFF: p1 = in-c1
Operator load-c1-p1-c:
  PRE: c1 = at-c, p1 = at-c  EFF: p1 = in-c1
...
Operator unload-c1-p1-a:
  PRE: c1 = at-a, p1 = in-c1  EFF: p1 = at-a
Operator unload-c1-p1-b:
  PRE: c1 = at-b, p1 = in-c1  EFF: p1 = at-b
Operator unload-c1-p1-c:
  PRE: c1 = at-c, p1 = in-c1  EFF: p1 = at-c
...
```

Figure 8: Part of an encoding of the transportation planning task with multi-valued state variables.

- Second, no matter how clever the encoding is, most planning tasks are not completely hierarchical in nature. To deal with causal cycles, we consider relaxations where some causal dependencies are ignored and use solutions to the relaxed problem within a heuristic search algorithm.

- Third, even for planning tasks that can be solved hierarchically, finding such a solution is difficult (indeed, still PSPACE-complete). For this reason, our heuristic function only considers a fragment of a task at a time, namely subproblems induced by a single state variable and its predecessors in the causal graph. Even *that* planning problem is still NP-complete, so that we are content with an incomplete solution algorithm within the heuristic solver. This solution algorithm has theoretical shortcomings but never failed us in practice.

Having introduced the rationale of our approach, we define the problem of planning with multi-valued state variables more formally in the following section. We then begin our discussion of the Fast Downward planning system by explaining its general architecture in Section 3. Subsequently, we devote one section to each phase of the planner, discussing *translation* in Section 4, *knowledge compilation* in Section 5, and *search* in Section 6. We conclude with a presentation of experimental results in Section 7 and some discussion in Section 8.

2. Multi-valued planning tasks

In this section, we formally introduce the problem of planning with multi-valued state variables. To provide a reference for comparison, we first introduce the better-known PDDL formalism. PDDL is the language in which the standard benchmarks in classical action planning are expressed. In particular, the planning tasks of the international planning competitions are expressed in PDDL, so a planning system must be able to deal with this language in order to participate. The language has evolved somewhat over time, with PDDL2.2 introducing derived predicates (axioms) as the major addition (Edelkamp & Hoffmann, 2003). Our definition of PDDL tasks includes all "purely logical" aspects of the language, thus excluding numerical and temporal planning problems.

Definition 1 ***PDDL tasks***
A (non-numeric, non-temporal) ***PDDL task*** *is given by a 5-tuple* $\Pi = \langle \mathcal{L}, \chi_0, \chi_\star, \mathcal{A}, \mathcal{O} \rangle$ *with the following components:*

- $\mathcal{L}$ *is a finite first-order language, consisting of constant symbols (**objects**) and relation symbols (**predicates**). Predicates are partitioned into **fluent predicates** (affected by operators) and **derived predicates** (computed by evaluating axioms).*

- χ_0 *is a conjunction of ground atoms over objects and fluent predicates called the **initial state**.*

- $\chi_\star$ *is a closed first-order formula over* $\mathcal{L}$ *called the **goal formula**.*

- $\mathcal{A}$ *is a set of **schematic axioms**, which are pairs* $\langle \varphi, \psi \rangle$ *such that* φ *is an atom over the set of derived predicates and* ψ *is a formula over* $\mathcal{L}$ *with* $\mathit{free}(\psi) \subseteq \mathit{free}(\varphi)$*. We write the axiom* $\langle \varphi, \psi \rangle$ *as* $\varphi \mathrel{:-} \psi$ *and call* φ *the **head** and* ψ *the **body** of the axiom.*

  We require that $\mathcal{A}$ *is stratifiable, i. e. there exists a total preorder* $\preceq$ *on the set of derived predicates such that for each axiom where* Q *occurs in the head, we must have* $P \preceq Q$

for all derived predicates P occurring in the body, and $P \prec Q$ for all derived predicates P occurring in a negative literal in the translation of the body to negation normal form. Intuitively, $P \prec Q$ means that the truth value of atoms over P must be computed before the truth value of atoms over Q.

- *$\mathcal{O}$ is a finite set of **schematic operators** over $\mathcal{L}$. A schematic operator $\langle \chi, e \rangle$ consists of a first-order formula χ over $\mathcal{L}$ called its **precondition** and its **effect** e. Effects are recursively defined by finite application of the following rules:*

  - *A literal l over $\mathcal{L}$, excluding derived predicates, is an effect called a **simple effect**.*
  - *If $e_1, \dots, e_n$ are effects, then $e_1 \wedge \dots \wedge e_n$ is an effect called a **conjunctive effect**.*
  - *If χ is a first-order formula over $\mathcal{L}$ and e is an effect, then $\chi \rhd e$ is an effect called a **conditional effect**.*
  - *If $X_1, \dots, X_k$ are first-order variables and e is an effect, then $\forall X_1, \dots, X_k : e$ is an effect called a **universally quantified effect** or **universal effect**.*

  Free variables of an effect are defined recursively as in first-order logic, where the set of free variables of a conditional effect is defined as free$(\chi \rhd e) =$ free$(\chi) \cup$ free(e).

  *The set of free variables of a schematic operator is defined as free$(\langle \chi, e \rangle) =$ free$(\chi) \cup$ free(e). Free variables are also referred to as the **parameters** of the schematic operator.*

We assume that the reader is already familiar with PDDL semantics and point to the reference (Edelkamp & Hoffmann, 2003) for more information. Our definition allows general first-order conditions as well as (possibly nested) conditional and quantified effects and axioms. The stratifiability condition for axioms ensures that the interpretation of derived predicates is well-defined. Without this condition, there could be rules of the form "$P(x)$ is true whenever $P(x)$ is false."

Apart from syntax, there are three aspects of non-numerical, non-temporal PDDL2.2 not captured by our definition:

- There are no operator names. The translator deals with operator names in such a way that the translated operators are referred to by the same name as its PDDL2.2 counterpart, so that the plans generated by the planner need not undergo any form of post-processing. This is all fairly simple, and we will not discuss this matter further.

- There is no distinction between domain constants and objects of the problem instance, or indeed between the domain and problem instance specification in general. At the level of individual problem instances at which the translator works, there is no need for such a distinction.

- There are no types. The translator compiles away types into unary predicates straight away, so a PDDL specification stating that `a` is an object of type `vehicle` is treated equivalently to an untyped specification stating that `(vehicle a)` is true in the initial state. Types occurring in quantified conditions or effects are translated accordingly; e. g. a precondition `(exists (?v - vehicle) (empty ?v))` is translated to $\exists v : \mathtt{vehicle}(v) \wedge \mathtt{empty}(v)$, and an effect `(forall (?v - vehicle) (empty ?v))` is translated to $\forall v : (\mathtt{vehicle}(v) \rhd \mathtt{empty}(v))$.

With PDDL as a starting point, let us now introduce the kinds of planning tasks that we are chiefly interested in. These are based on the SAS$^+$ planning model (Bäckström & Nebel, 1995; Jonsson & Bäckström, 1998a), extended to allow for derived predicates and conditional effects.

The definition will exhibit a number of similarities, but also a few differences between PDDL tasks and our planning model. Most notably, PDDL tasks use first-order concepts such as schematic operators whose variables can be instantiated in many different ways, while our formalism is grounded. Moreover, our formalism only allows simple conjunctions in goals, axioms and operators, and conditional effects cannot be nested. The former difference necessitates operator instantiation as part of the translation process, while the others require normalization of conditions.

Definition 2 ***Multi-valued planning tasks (MPTs)***
*A **multi-valued planning task (MPT)** is given by a 5-tuple $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$ with the following components:*

- *$\mathcal{V}$ is a finite set of **state variables**, each with an associated finite domain $\mathcal{D}_v$. State variables are partitioned into **fluents** (affected by operators) and **derived variables** (computed by evaluating axioms). The domains of derived variables must contain the **undefined value** $\perp$.*

  *A **partial variable assignment** or **partial state** over $\mathcal{V}$ is a function s on some subset of $\mathcal{V}$ such that $s(v) \in \mathcal{D}_v$ wherever $s(v)$ is defined. A partial state is called an **extended state** if it is defined for all variables in $\mathcal{V}$ and a **reduced state** or **state** if it is defined for all fluents in $\mathcal{V}$. In the context of partial variable assignments, we write $v = d$ for the variable-value pairing $\langle v, d \rangle$ or $v \mapsto d$.*

- *s_0 is a state over $\mathcal{V}$ called the **initial state**.*

- *$s_\star$ is a partial variable assignment over $\mathcal{V}$ called the **goal**.*

- *$\mathcal{A}$ is a finite set of (MPT) **axioms** over $\mathcal{V}$. Axioms are triples of the form $\langle cond, v, d \rangle$, where cond is a partial variable assignment called the **condition** or **body** of the axiom, v is a derived variable called the **affected variable** and $d \in \mathcal{D}_v$ is called the **derived value** for v. The pair $\langle v, d \rangle$ is called the **head** of the axiom and can be written as $v := d$.*

  *The axiom set $\mathcal{A}$ is partitioned into a totally ordered set of **axiom layers** $\mathcal{A}_1 \prec \dots \prec \mathcal{A}_k$ such that within the same layer, each affected variable may only be associated with a single value in axiom heads and bodies. In other words, within the same layer, axioms with the same affected variable but different derived values are forbidden, and if a variable appears in an axiom head, then it may not appear with a different value in a body. This is called the **layering property**.*

- *$\mathcal{O}$ is a finite set of (MPT) **operators** over $\mathcal{V}$. An operator $\langle pre, eff \rangle$ consists of a partial variable assignment pre over $\mathcal{V}$ called its **precondition**, and a finite set of **effects** eff. Effects are triples $\langle cond, v, d \rangle$, where cond is a (possibly empty) partial variable assignment called the **effect condition**, v is a fluent called the **affected variable** and $d \in \mathcal{D}_v$ is called the **new value** for v.*

For axioms and effects, we also use the notation $cond \rightarrow v := d$ in place of $\langle cond, v, d \rangle$ and omit the set brackets around cond where this does not lead to confusion.

Since we will mostly be concerned with the MPT formalism in the following, we provide a formal semantics for MPT planning, starting with a formalization of axioms:

Definition 3 ***Extended states defined by a state***
Let s be a state of an MPT Π with axioms $\mathcal{A}$, layered as $\mathcal{A}_1 \prec \cdots \prec \mathcal{A}_k$. The ***extended state defined by*** *s, written as $\mathcal{A}(s)$, is the result s' of the following algorithm:*

algorithm evaluate-axioms($\mathcal{A}_1, \ldots, \mathcal{A}_k$, s):
  for each variable v:
    $$s'(v) := \begin{cases} s(v) & \text{if } v \text{ is a fluent variable} \\ \bot & \text{if } v \text{ is a derived variable} \end{cases}$$
  for $i \in \{1, \ldots, k\}$:
    while there exists an axiom $(\mathit{cond} \rightarrow v := d) \in \mathcal{A}_i$ with $\mathit{cond} \subseteq s'$ **and** $s'(v) \neq d$:
      Choose such an axiom $\mathit{cond} \rightarrow v := d$.
      $s'(v) := d$

In other words, axioms are evaluated in a layer-by-layer fashion using fixed point computations, which is very similar to the semantics of stratified logic programs. It is easy to see that the layering property from Definition 2 guarantees that the algorithm terminates and produces a deterministic result. Having defined the semantics of axioms, we can now define the state space of an MPT:

Definition 4 ***MPT state spaces***
The ***state space*** *of an MPT $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$, denoted as $\mathcal{S}(\Pi)$, is a directed graph. Its vertex set is the set of states of $\mathcal{V}$, and it contains an arc (s, s') iff there exists some operator $\langle \mathit{pre}, \mathit{eff} \rangle \in \mathcal{O}$ such that:*

- *$\mathit{pre} \subseteq \mathcal{A}(s)$,*
- *$s'(v) = d$ for all effects $\mathit{cond} \rightarrow v := d \in \mathit{eff}$ such that $\mathit{cond} \subseteq \mathcal{A}(s)$, and*
- *$s'(v) = s(v)$ for all other fluents.*

Finally, we can define the MPT planning problem:

Definition 5 ***MPT planning***
MPT-PLANEX *is the following decision problem: Given an MPT Π with initial state s_0 and goal $s_\star$, does $\mathcal{S}(\Pi)$ contain a path from s_0 to some state s' with $s_\star \subseteq s'$?*

MPT-PLANNING *is the following search problem: Given an MPT Π with initial state s_0 and goal $s_\star$, compute a path in $\mathcal{S}(\Pi)$ from s_0 to some state s' with $s_\star \subseteq s'$, or prove that none exists.*

The MPT-PLANEX problem is easily shown to be **PSPACE**-hard because it generalizes the plan existence problem for propositional STRIPS, which is known to be **PSPACE**-complete (Bylander, 1994). It is also easy to see that the addition of multi-valued domains, axioms and conditional effects does not increase the theoretical complexity of MPT planning beyond propositional STRIPS. Thus, we conclude our formal introduction of MPT planning by stating that MPT-PLANEX is **PSPACE**-complete, and turn to the practical side of things in the following section.

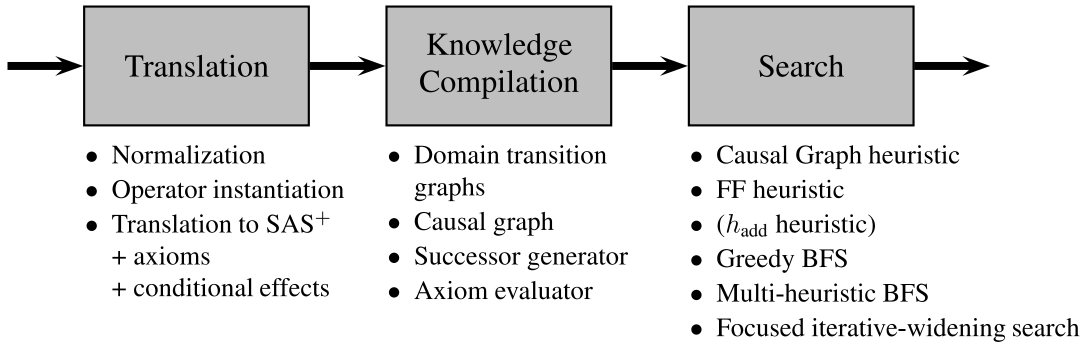

Figure 9: The three phases of Fast Downward's execution.

3. Fast Downward

Fast Downward is a classical planning system that implements the ideas introduced in the motivating section. It can deal with the full range of propositional PDDL2.2, i. e. in addition to STRIPS planning, it allows using arbitrary formulae in operator preconditions and goal conditions, it can deal with conditional and universally quantified effects, and it supports derived predicates (axioms).

The name of the planner derives from two sources: Of course, one of these sources is Hoffmann's very successful FF ("Fast Forward") planner (Hoffmann & Nebel, 2001). Like FF, Fast Downward is a heuristic progression planner, i. e. it computes plans by heuristic search in the space of world states reachable from the initial situation. However, compared to FF, Fast Downward uses a very different heuristic evaluation function called the *Causal Graph heuristic*. The heuristic evaluator proceeds "downward" in so far as it tries to solve planning tasks in the hierarchical fashion outlined previously. Starting from top-level goals, the algorithm recurses further and further down the causal graph until all remaining subproblems are basic graph search tasks.

Similar to FF, the planner has shown excellent performance: The original implementation of the Causal Graph heuristic, plugged into a standard best-first search algorithm, outperformed the previous champions in that area, FF and LPG (Gerevini et al., 2003), on the set of STRIPS benchmarks from the first three international planning competitions (Helmert, 2004). Fast Downward itself followed in the footsteps of FF and LPG by winning the propositional, non-optimizing track of the 4th International Planning Competition at ICAPS 2004 (referred to as IPC4 from now on).

Currently, Fast Downward is only available in binary form, and can only be obtained by contacting the members of the Fast Downward team directly. We plan to create a web site for the planner and prepare a source release in the near future. In the meantime, we hope that this article addresses all important technical questions about the planner. As mentioned in the introduction, Fast Downward solves a planning task in three phases (Fig. 9):

- The *translation* component is responsible for transforming the PDDL2.2 input into an MPT. In addition to generating a multi-valued representation of the input task, it applies a number of normalizations to compile away syntactic constructs like disjunctions which are not directly supported by the Causal Graph heuristic and performs grounding of axioms and operators. The translation component is described in the following section.

- The *knowledge compilation* component generates four kinds of data structures that play a central role during search: *Domain transition graphs* encode how, and under what conditions, state variables can change their values. The *causal graph* represents the hierarchical dependencies between the different state variables. The *successor generator* is an efficient data structure for determining the set of applicable operators in a given state. Finally, the *axiom evaluator* is an efficient data structure for computing the current values of derived predicates. The knowledge compilation component is described in Section 5.

- The *search* component implements three search algorithms; which of these is actually used is determined by command-line switches. Two of these algorithms make use of heuristic evaluation functions. One is the well-known greedy best-first search algorithm, the other is called *multi-heuristic best-first search*, a variant of greedy best-first search that tries to combine several heuristic evaluators in an orthogonal way. For use by these search algorithms, we have implemented three different heuristic evaluators: Causal Graph heuristic, FF heuristic, and HSP's h_{add} heuristic. Currently, greedy best-first search always uses the Causal Graph heuristic and multi-heuristic best first search always uses the Causal Graph and FF heuristics, but other combinations are easy to set up. The third search algorithm implemented is called *focused iterative-widening search*. It is not a heuristic search algorithm in the sense that it does not use an explicit heuristic evaluation function. Instead, it uses the information encoded in the Causal Graph to estimate the "usefulness" of operators towards satisfying the goal of the plan.

  The search component can be configured to either run a certain search algorithm exclusively, or to try several search algorithms concurrently, for which purpose it implements a simple scheduler which assigns time to the different algorithms in a round-robin fashion.

  The two variants of Fast Downward that participated in IPC4 only differed in the choice of parameters for the search component. *Fast Downward* used greedy best-first search (using the Causal Graph heuristic) and focused iterative-widening search, while *Fast Diagonally Downward* used multi-heuristic best-first search (using the Causal Graph and FF heuristics) and focused iterative-widening search. Both configurations used the internal scheduler to distribute execution time between the two search algorithms used.

  The search component is described in Section 6.

In the current implementation, the split into three components is physical, not just conceptual: Fast Downward indeed consists of three separate programs, which are called in sequence and communicate with one another by writing intermediate results to files on disk. This arrangement has some obvious disadvantages, chiefly among them the fact that on some tasks, the planner spends a large amount of time writing data to disk and reading it into memory. This mostly affects planning domains with very large tasks but comparatively simple structure, like LOGISTICS and SATELLITE. For some benchmarks in these domains, Fast Downward's runtime is dominated by I/O.

However, the architectural split also has some notable positive aspects. First and foremost, having three separate programs facilitates the joint development of a system, as communication between the different modules is limited to small, well-defined interfaces. Second, it allows using different programming languages for the different components without the difficulties of integration usually associated with mixed-language development.

Module	Language	Lines of Code
Translation	Python	3082
Knowledge Compilation	C++	1871
Search	C++	2966

Figure 10: Programming languages and program lengths for the three components of Fast Downward (IPC4 version).

Fig. 10 shows the programming languages in which the different components were developed and illustrates their relative size. We see that the largest component, translation, is implemented in a very-high level language, Python, whereas the most time-critical component, search, is implemented in a language that generates much faster programs, C++. In our experience, supported by empirical studies, C++ programs are on average about 5-10 times as long as equivalent Python programs, so we would like to avoid implementing the translation component in C++. On the other hand, a search component implemented in Python would be unacceptably slow. For these reasons, it is not likely that the physical split of Fast Downward into separate programs will go away in the near future. However, we are working towards integrating the two C++ components, which were split mostly to facilitate joint development.

There is a third benefit to having separate programs for the three phases of execution, and this is for users of the planner who are only interested in processing one or two of the three phases. For example, tasks already formulated in the multi-valued representation that the later phases use need not undergo translation. More importantly, the translation module can be used for other approaches to planning that rely on, or work better for, multi-valued representations. Fast Downward's translation component has already been applied as a front end to a planning system based on mixed integer programming with very encouraging results (van den Briel, Vossen, & Kambhampati, 2005). We are certain that the translator can be put to other useful purposes.

4. Translation

The purpose of the translation component is to transform the input planning task, specified in the (first-order) PDDL 2.2 formalism, into a fully instantiated multi-valued representation based on the SAS^+ formalism.

The first implementation of Fast Downward employed the preprocessing algorithm of MIPS 1.0 (Edelkamp & Helmert, 1999) for this purpose. However, that algorithm is fairly limited, as it cannot deal with ADL-style conditions, conditional effects, or derived predicates. In other words, it is only applicable to STRIPS domains. Moreover, MIPS 1.0 cannot handle some features of the STRIPS fragment of PDDL 2.2 either, such as domain constants and types. Even in STRIPS domains, the runtime of the MIPS preprocessor scales exponentially with the number of schematic (first-level) operators of a planning task. While this is not problematic for traditional benchmarks, where the number of schematic operators is a small constant for a given domain, some of the IPC4 domains contain partially pre-instantiated operators leading to very large domain files. For example, 15 of the 50 AIRPORT tasks (STRIPS formulation) of IPC4 contain more than 1300 schematic operators each.

Because of these shortcomings of the preexisting implementation, we decided to develop a new translation algorithm for IPC4. As mentioned in the high-level overview of Fast Downward's architecture, this new translator is implemented in the Python language, which usually leads to slower programs than languages like C++ which are closer to the underlying CPU architecture. For this reason, during IPC4 we used the "old" translator for the SATELLITE benchmark suite, which is expressed in untyped STRIPS and includes some very large instances. We used the Python implementation for all others domains, which all require language features not supported by the old translator, such as ADL conditions or derived predicates (AIRPORT, PROMELA, PSR) or constants and types (PIPESWORLD). For domains which can be handled by both translators, their output is mostly equivalent. Therefore, choice of translation method can be based on execution time in these cases. In the rest of this section, we give an account of the new translation component. For the old C++ implementation, please consult the reference by Edelkamp and Helmert (1999).

4.1 Overview of the translation algorithm

The translation component is the most complex of the parts of Fast Downward by far and has not yet been thoroughly described, so that we dedicate much more space to its discussion than to the other two components.

Translation is performed in a sequence of transformation steps. Starting from a PDDL specification, we apply some well-known logical equivalences to compile away types and simplify conditions and effects in the *normalization* step. Next, the *invariant synthesis* step computes mutual exclusion relations between atoms, which are later used for synthesizing the MPT variables. The *grounding* step performs a relaxed reachability analysis to compute the set of ground atoms, axioms and operators that are considered relevant for the planning task and computes a grounded PDDL representation. Invariant synthesis and grounding are not related to one another and could just as well be performed in the opposite order. Finally, the *MPT generation* step chooses the final set of state variables by using the information from invariants and grounding and produces the MPT output.

The translation process in outlined in Fig. 11. We will now discuss the various transformation steps in sequence.

4.2 Normalization

The normalization step has three responsibilities: Compiling away types, simplifying conditions, and simplifying effects. The current implementation of the translator cannot handle PDDL types in their full generality: Type inheritance and the `either` construct are not supported. It would not be difficult to add these to the mix, but these seem to be unused language features, and we did not want to waste implementation effort on them. So we only deal with primitive types and the built-in standard type `object`, to which all objects belong.

4.2.1 COMPILING AWAY TYPES

As indicated earlier, types are compiled away as soon as the planning task is read in. For each type occurring in the input, and for the type `object`, we introduce a new unary predicate with the same name. Typed constructs occur in PDDL2.2 specifications in a semantically meaningful way in three places:

1. Definition of domain constants and objects of the task (*typed objects*).

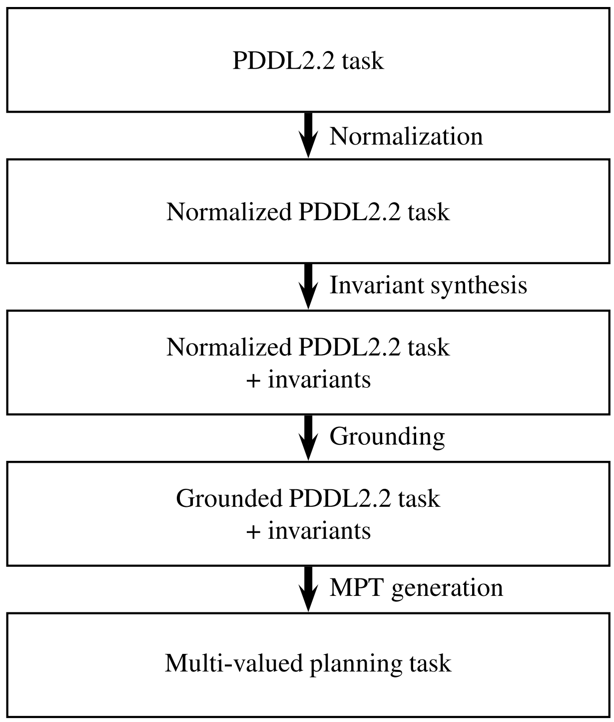

Figure 11: Overview of the translation algorithm.

2. Definition of formal parameters of schematic operators (*typed operators*).

3. Definition of quantified variables in existential and universal conditions and universal effects (*typed quantifiers*).

Typed objects are translated into new atoms for the initial state. For example, the specification `someobj - sometype` leads to a new initial atom `(sometype someobj)`. Moreover, we introduce an initial atom `(object someobj)` for each object `someobj`.

Typed operators are transformed by introducing new preconditions. For example, for an operator with parameter specification `:parameters (?par1 - type1 ?par2 - type2)` and precondition φ, the parameter specification is replaced by `:parameters (?par1 ?par2)` and the precondition is replaced by `(and (type1 ?par1) (type2 ?par2)` φ`)`.

Typed quantifiers in conditions are compiled away with the usual first-order logic idioms, so that `(exists (?v - type)` φ`)` translates to `(exists (?v) (and (type ?v)` φ`))` and `(forall (?v - type)` φ`)` translates to `(forall (?v) (imply (type ?v)` φ`))`.

Similarly, typed quantifiers in effects are compiled into conditional effects, so that the effect `(forall (?v - type)` η`)` becomes `(forall (?v) (when (type ?v)` η`))`.

After types have been eliminated, we are left with a PDDL task in the sense of Definition 1. We will thus use the more concise logical notation from that definition in the following, rather than the more lengthy PDDL syntax. For example, we will write $\varphi \vee \psi$ instead of `(or` φ ψ`)` and $\varphi \rhd e$ instead of `(when` φ e`)`.

4.2.2 SIMPLIFYING CONDITIONS

In PDDL tasks, general first-order formulae may occur in many places: goal formula, axiom bodies, operator preconditions and conditions of conditional effects. Our aim is to replace all these with simple conjunctions of literals.

Towards this goal, we first eliminate implications with the equivalence $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$ and translate the resulting conditions into first-order negation normal form using de Morgan's laws for first-order logic.

The next step is slightly tricky. If there are any universally quantified conditions, we rewrite the outermost universal quantification in all conditions with the equivalence $\forall x\varphi \equiv \neg\exists x\neg\varphi$. This might seem somewhat silly because this transformation destroys negation normal form, so after the rewrite, we introduce a new axiom for the subformula that violates the normal form property, $\exists x\neg\varphi$. Formally, if free$(\exists x\neg\varphi) = \{v_1, \dots, v_k\}$, we introduce a new derived predicate `new-axiom` of arity k, defined by the axiom `new-axiom`$(v_1, \dots, v_k)$`:-`ψ, where ψ is the translation of $\exists x\neg\varphi$ to negation normal form. We can then replace the original condition $\forall x\varphi$ by $\neg$`new-axiom`$(v_1, \dots, v_k)$. If several variables are universally quantified together within the same expression, we transform them together, introducing only one new derived predicate for the quantifier group. We repeat this step until there are no more universally quantified conditions. Note that only universally quantified *conditions* are translated, not universal *effects*, which also use the $\forall$ notation. Universal effects cannot simply be compiled away, so we deal with them separately in a later stage.

If after elimination of universal quantifiers the goal condition is *not* a simple conjunction, we replace it by a new axiom, since the following transformations sometimes require splitting several conditions into two, which is easy to do for axiom bodies, operator preconditions and effect conditions, but not possible in our formalism for goal conditions, of which there can be only one. So for example, if the goal is $\varphi \vee \psi$, we introduce a new parameter-less derived predicate `goal-axiom` and a new axiom `goal-axiom :-` $\varphi \vee \psi$, replacing the original goal with the atom `goal-axiom`.

The next step is the elimination of disjunctions. We move disjunctions to the roots of conditions by applying the equivalences $\exists x(\varphi \vee \psi) \equiv \exists x\varphi \vee \exists x\psi$ and $\varphi \wedge (\psi \vee \psi') \equiv (\varphi \wedge \psi) \vee (\varphi \wedge \psi')$ and the laws of associativity and commutativity. In theory, moving disjunctions over conjunctions could lead to an exponential increase in formula size, which we could avoid by introducing new axioms for component formulae. In practice, the conditions encountered in actual planning domains are not problematic in this regard, and we decided that the potential savings in the size of the representation were not worth the overhead of maintaining the state of another derived variable during search.

After disjunctions have been moved to the root of all formulae, we can eliminate them by splitting the surrounding structures: If the disjunction $\varphi \vee \psi$ is part of an axiom body, we generate two axioms with identical head, one with body φ and one with body ψ. If the disjunction is part of an operator precondition, we replace the operator by two copies of the original, one with precondition φ and one with precondition ψ. Finally, if the disjunction is part of an effect condition, we replace the conditional effect $(\varphi \vee \psi) \rhd e$ by $(\varphi \rhd e) \wedge (\psi \rhd e)$.

Next, we move existential quantifiers out of conjunctions by applying the equivalence $(\exists x\varphi) \wedge \psi \equiv \exists x(\varphi \wedge \psi)$. The equivalence only holds when $x \notin \text{free}(\psi)$, so to avoid trouble here and later, we first rename all variables bound by quantifiers to some unique name.

Having moved existential quantifiers to the root of conditions, we eliminate them as follows: For axioms, we simply drop them, following the PROLOG convention that all free variables in the body that are not part of the head are implicitly existentially quantified. For operator preconditions, we also drop them, adding the existentially quantified variables to the parameter list of the schematic operator. For effect conditions, we replace $(\exists x\varphi) \rhd e$ by $\forall x : (\varphi \rhd e)$.

4.2.3 SIMPLIFYING EFFECTS

After the somewhat laborious simplification of conditions, effect simplification is conceptually very simple. First, universal and conditional effects are moved into conjunctive effects by the equivalences $\forall x : (e \wedge e') \equiv (\forall x : e) \wedge (\forall x : e')$ and $\varphi \rhd (e \wedge e') \equiv (\varphi \rhd e) \wedge (\varphi \rhd e')$. Second, conditional effects are moved into universal effects by the equivalence $\varphi \rhd (\forall x : e) \equiv \forall x : (\varphi \rhd e)$. Finally, nested effects of the same type are flattened, i. e. conjunctive effects containing conjunctive effects are collapsed into a single conjunctive effects with more conjuncts, universal effects containing universal effects are collapsed into a single universal effect quantifying over more variables, and nested conditional effects of the type $\varphi \rhd (\psi \rhd e)$ are transformed to $(\varphi \wedge \psi) \rhd e$. Note that this latter modification preserves the previously generated normal form for effect conditions.

After these transformations, the possible nesting of effects is thus restricted to the simple chain *conjunctive effect* $\succ$ *universal effect* $\succ$ *conditional effect* $\succ$ *simple effect*. However, not all effect types must necessarily be present, e. g. conditional effects *need not* occur within universal effects, etc. To enforce a regular effect structure, we replace simple effects e not surrounded by conditional effects by $\top \rhd e$ ($\top$ is seen as the empty conjunction, so this condition is in normal form), conditional effects e not surrounded by universal effects by $\forall : e$ (quantifying over zero variables), and universal effects e not surrounded by conjunctive effects by a conjunctive effect containing the singleton e.

As a result, after normalization each operator has a list (conjunction) of effects, each a simple effect with an associated set of universal quantifiers and an associated condition, both of which can be trivial. Thus it is not necessary to store normalized operator effects in a tree structure; a flat vector is sufficient.

4.2.4 NORMALIZATION RESULT

This concludes the normalization step. In Fig. 11, we referred to the output of the normalization phase as a *normalized PDDL2.2 task*. Let us formalize this notion here for the benefit of further discussion:

Definition 6 ***Normalized PDDL tasks***
A ***normalized PDDL task*** *is a PDDL task that satisfies the following structural restrictions:*

- *The goal formula is a conjunction of literals.*
- *All axiom bodies are conjunction of literals (except for the possible implicit existential quantification of free variables not occurring in the axiom head).*
- *All operator preconditions are conjunctions of literals.*
- *All effect conditions are conjunctions of literals.*
- *All operator effects are conjunctions of universally quantified conditional simple effects.*

In the following, we will refer to the individual simple effects of an operator in a normalized PDDL task as being arranged in an *effect list*. For the simple effect e occurring within the universal conditional effect $\forall vars : \varphi \rhd e$, we will refer to *vars* as the set of *bound variables* of e and to φ as the *condition* of e. If e is a positive literal, we will call it an *add effect*, otherwise a *delete effect*.

4.3 Invariant synthesis

The invariant synthesis step is the part of the translation component which is most closely based on the original MIPS preprocessor (Edelkamp & Helmert, 1999). Our most important modification is the extension of the algorithm to non-STRIPS tasks. There are some other differences and generalizations, but these are mostly minor.

An invariant is a property of a world state in a planning task which is satisfied by all world states that are reachable from the initial state. Many invariants are uninteresting; for example, the property "At least five state variables are true" is an invariant in most propositional STRIPS planning tasks, but knowing this does not seem to be a useful piece of information for a planner. Other invariants would be useful to know but are too difficult to verify. For example, "This state is not a goal state" is an invariant iff the planning task is not solvable, so confirming the invariance of that state property is **PSPACE**-hard.

Nevertheless, invariants are a useful tool for many planning systems, which is why they have been studied by many researchers in a variety of contexts (Fox & Long, 1998; Gerevini & Schubert, 1998; Rintanen, 1998, 2000), often involving SAT-based planning. For the purposes of translating propositional planning problems to a multi-valued formalism, *mutual exclusion* (*mutex*) invariants are especially interesting. A mutex invariant states that certain propositions will never be true at the same time. This affects translation because a set of propositions which are pairwise mutually exclusive can be easily encoded as a *single* state variable stating *which* of the proposition is true (if any is true at all), rather than as a number of state variables encoding the truth value for each proposition individually.

Invariance is usually proven inductively: First, we prove that a state property is true in the initial state. Then, we show that if the property is true in some state, it must also be true in all successor states. Together, this implies that the property is an invariant. As mentioned before, the automatic discovery of invariants is a hard problem in general, but for many relevant types of state properties, sufficient conditions exist that can be checked quickly.

Still, synthesizing invariants is costly, and for this reason, we want our algorithm to work directly with the first-order description, not on an instantiated representation. Indeed, our algorithm does not rely on the information in the *task* file of the PDDL input at all, solely exploiting information present in the *domain* file, which is a nice property. However, this rules out mutex conditions, because a mutex cannot be established without checking the initial state. Instead, we use a slight generalization of mutexes:

Definition 7 ***Monotonicity properties***
*A **monotonicity property** for a PDDL task Π is given by a pair $\mathcal{P} = \langle V, \Phi \rangle$, where V is a set of first-order variables called the **parameters** of the property, and Φ is a set of atoms. Variables occurring freely in Φ which are not parameters are called **counted variables** of the property.*

For $V = \{v_1, \dots, v_m\}$ and $\Phi = \{\varphi_1, \dots, \varphi_k\}$, we write the monotonicity invariant $\mathcal{P}$ symbolically as $\forall v_1 \dots v_m\ \varphi_1 + \dots + \varphi_k \downarrow$. In the special case $V = \emptyset$, we write $\forall \cdot\ \varphi_1 + \dots + \varphi_k \downarrow$.

This defines the syntax for monotonicity properties; we now have to provide the semantics. Since the definition is somewhat involved, we provide an example from a transportation domain first. Consider the invariant $\langle \{p\}, \{\mathtt{at}(p, l), \mathtt{in}(p, v)\} \rangle$, where p, l and v are variable symbols. We write this as $\forall p\, \mathtt{at}(p, l) + \mathtt{in}(p, v) \downarrow$ and read it as "For all packages p, the number of locations l such that $\mathtt{at}(p, l)$ is true plus the number of vehicles v such that $\mathtt{in}(p, v)$ is true, is non-increasing." In our terminology, p is the parameter of the property, while l and v are the counted variables. This property is an invariant, and it is one of the invariants found by our algorithm in a LOGISTICS-like domain. Now for the formal semantics:

Definition 8 ***Monotonicity invariants***
In the following, let Π be a PDDL task and let $\mathcal{P} = \langle V, \Phi \rangle$ be a monotonicity property for Π.

*An **instance** of $\mathcal{P}$ is a function α mapping the variables in V to objects of the planning task Π.*

*The set of **covered facts** of an instance α is the set of all ground atoms of the planning task Π which unify with some $\varphi \in \Phi$ under α, i. e. the set of all ground atoms φ_0 of Π for which there exists a variable map $\beta \supseteq \alpha$ such that $\beta(\varphi) = \varphi_0$ for some $\varphi \in \Phi$.*

*The **weight** of an instance α in a state s is the number of covered facts of α which are true in s.*

*The monotonicity property $\mathcal{P}$ is called a **monotonicity invariant** iff for all instances α of $\mathcal{P}$, all states s reachable from the initial state of Π and all successor states s' of s, the weight of α in s' is no greater than the weight of α in s.*

To understand this definition, observe its relation to the example invariant above. As hinted before, monotonicity invariants are useful for grouping a number of related propositions into a single multi-valued variable: If a given monotonicity invariant is true for a planning task *and* a given instance of that invariant has weight 1 in the initial state, then the facts covered by that instance are pairwise mutually exclusive. This is how we will use the synthesized invariants during the later stages of translation.

So how do we generate invariants? Since the number of feasible monotonicity properties is too high for a simple guess-and-check algorithm, we follow a *guess, check and repair* approach: Starting from a set of a few simple initial elements, we try to prove that our candidate properties are indeed invariants. Whenever this is the case, we keep the invariant and do not consider it further. However, when the proof fails, we try to detect *why* this is the case and refine the property to generate more candidates. From a high-level perspective, this is basically a search problem, and indeed we solve it using standard breadth-first search (with a closed list). What remains to be said is how the search space of the algorithm is defined:

- *Initial states:* What are the initial candidate properties?
- *Termination test:* How do we prove that a candidate is an invariant?
- *Successor set:* How do we refine a property for which verification fails?

We will deal with these questions in the following.

4.3.1 INITIAL CANDIDATES

Before starting the actual invariant synthesis algorithm, we check which predicates are affected by operators at all: Some predicates, including but not limited to those representing types, are *constant* in the sense that atoms over these predicates have the same truth values in all states. Such predicates are no longer needed after grounding, so we need not consider them for monotonicity invariants. Of course, a constant predicate trivially satisfies a monotonicity invariant, but we do not consider these as they are not very useful.

Therefore, we limit the set of interesting predicates to all *modifiable fluent predicates*, i. e. predicates which occur within operator effects (as part of a simple effect, not merely as part of an effect condition). Note that this also excludes derived predicates. In theory, there is no reason why there could be no monotonicity invariants involving derived predicates, but in practice we have not seen examples of this, and detecting them would require a more global view of the task definition and hence more effort than we would like to spend. We will come back to the issue of derived predicates when discussing our method for proving invariance.

The set of initial candidate properties consists of all those properties (up to isomorphism, i. e. renaming of variables) which contain at most one counted variable and exactly one atom, over a modifiable fluent predicate, whose parameters are distinct variables. In our experience, invariants with several counted variables per atom are exceedingly rare; in fact, we have not seen an example in practice.

To illustrate the initialization of candidate properties, we show the three items generated for the binary `at` predicate in the LOGISTICS domain:

$$\forall x\, \mathtt{at}(x, l) \downarrow \tag{1}$$

$$\forall l\, \mathtt{at}(x, l) \downarrow \tag{2}$$

$$\forall x, l\, \mathtt{at}(x, l) \downarrow \tag{3}$$

Similar candidates are introduced for the `in` predicate. Intuitively, the first property states that no object can be at more locations in the successor state than in the current state, the second property states that no location can be occupied by more objects in the successor state than in the current state, and the third property states that a given object cannot occupy a given location in the successor state if this is not the case in the current state.

Properties (2) and (3) are obviously not invariants. Property (1) is not an invariant either because an object which is currently inside a vehicle can be at some location in the successor state while being at no location in the current state. However, we will see that we will be able to refine (1) into an invariant.

4.3.2 PROVING INVARIANCE

When is a monotonicity property an invariant? We stated that invariants are usually proven by establishing their truth in the initial state and using an inductive argument for operator application. For monotonicity properties, only the inductive step is necessary; there is nothing special to prove about the initial state. So in order to prove that a given monotonicity property is an invariant, we must show that no operator can increase the weight of any of its instances. An operator increases the weight of some instance of a monotonicity property iff the number of covered facts that it makes true is greater than the number of covered facts that it makes false. If an operator does not increase the weight of any instance, then we say that it is *balanced* with regard to the invariant.

Ultimately, we are interested in instances of monotonicity invariants that give rise to mutexes, so that only instances of weight 1 are relevant for us. For this reason, we use the following condition which is slightly stronger than balance:

Definition 9 ***Threatened monotonicity properties***
*A monotonicity property $\mathcal{P}$ is **threatened** by a schematic operator iff one of the following two conditions holds:*

- *The operator has an add effect that can increase the weight of an instance of $\mathcal{P}$ in some state, but no delete effect that is guaranteed to decrease the weight of the same instance in the same state. In this case, we say that the operator is **unbalanced** with regard to $\mathcal{P}$.*
- *The operator has two add effects, involving different atoms, that can simultaneously increase the weight of the same instance of $\mathcal{P}$ in some state. In this case, we say that the operator is **too heavy** for $\mathcal{P}$.*

Clearly, not being threatened by any schematic operator is a sufficient condition for being a monotonicity invariant. The definition gives rise to the algorithm shown in Fig. 12. Most of the actual work is in unifying operator parameters and quantified variables of universal conditions; the algorithm simplifies significantly in STRIPS domains. We do not want to discuss the algorithm in all detail, instead focusing on two points that require some explanation, namely the satisfiability and entailment tests that occur towards the end of algorithms *check-operator-too-heavy* and *check-operator-unbalanced*.

For the heaviness test, two add effects can only lead to an operator being too heavy if the operator is actually applicable (o'.precondition is true), both add effects apply (e.condition and e'.condition are true) and the add effects actually add propositions that were not true previously (e.atom and e'.atom are false). For the imbalance test, an add effect is unbalanced by default. However, it becomes balanced if whenever the operator is actually applicable (o'.precondition is true), the add effect triggers (e.condition is true) and actually adds something (e.atom is false), then something is

algorithm prove-invariant(V, Φ):
  for each schematic operator o with an add effect for some predicate in Φ:
    call check-operator-too-heavy(o, V, Φ).
    call check-operator-unbalanced(o, V, Φ).
  accept property as an invariant.

{ In the following, the *variables* of an operator include both its operator parameters and quantified variables of its effects. We assume that all variable names are unique, and that whenever a variable is renamed, the change is immediately reflected in program variables referring to effects of the operator. For example, if $e = \mathtt{at}(p, l)$ is an effect of operator o and p is renamed to v within o, then e becomes $\mathtt{at}(v, l)$. }

algorithm check-operator-too-heavy(o, V, Φ):
  Let o' be a copy of o.
  Duplicate all (non-trivially) quantified effects of o', naming quantified variables uniquely.
  for each pair (e, e') of add effects of o' that affect a predicate in Φ:
    if the variables of operator o' can be renamed so that
     (e.atom $\neq e'$.atom **and** covers(V, Φ, e.atom) **and** covers(V, Φ, e'.atom) **and**
     o'.precondition $\wedge$ e.condition $\wedge$ e'.condition $\wedge$ $\neg e$.atom $\wedge$ $\neg e'$.atom is satisfiable):
      reject property. { The operator is too heavy. }

algorithm check-operator-unbalanced(o, V, Φ):
  for each add effect e of o that affects a predicate in Φ:
    Let o' be a copy of o where the variables are renamed so that covers(V, Φ, e.atom) is true. Do not rename two variables to the same variable except when forced.
    for each delete effect e' of o' that affects a predicate in Φ:
      if the quantified variables of e' can be renamed in o' so that
       (covers(V, Φ, e'.atom) **and**
       o'.precondition $\wedge$ e.condition $\wedge$ $\neg e$.atom $\models$ e'.condition $\wedge$ e'.atom):
        continue with next add effect e. { This add effect is balanced. }
    reject property. { The operator is unbalanced. }

function covers(V, Φ, ψ):
  for each $\varphi \in \Phi$:
    if the counted variables in φ (those not in V) can be renamed so that $\varphi = \psi$:
      return true.
  return false.

Figure 12: Algorithm for proving that a monotonicity property $\langle V, \Phi \rangle$ is an invariant.

deleted at the same time, which means that the delete effect triggers (e'.condition is true) and deletes something that was previously true (e'.atom is true).

Coming back to the earlier LOGISTICS example, all three initial candidates are threatened by the same operator `unload-truck`, whose add effect $\mathtt{at}(x, l)$ is not balanced. Thus, as indicated before, none of (1)-(3) is an invariant.

There are a few subtleties about the algorithm which we want to point out briefly:

- We duplicate universal effects at the beginning of *check-operator-too-heavy* so that we can detect if two instantiations of the same universal effect can simultaneously increase the weight of some instance of the monotonicity property.

- Where Fig. 12 contains statements like "Let o' be a copy of o where variables are renamed so that...", the question arises whether such a renaming is uniquely determined, and what to do if it is not. Indeed, renamings are unique (and easy to compute) as long as all atoms of the property refer to *different* predicates, which is usually the case. However, the algorithm generalizes to properties with several occurrences of the same predicate, like $\forall x\, \mathtt{at}(x, y) + \mathtt{at}(y, x) \downarrow$. This requires that all possible (non-isomorphic) renamings must be considered for o' in algorithm *check-operator-unbalanced*. In our experience, invariants of this type are not very useful, although Fast Downward implements them correctly.

- We have noted before that we do not consider invariants involving derived predicates. This is because axioms correspond to operators that have a single add effect, but no delete effect. Monotonicity properties including derived predicates can thus never be balanced, except if the axiom body is unsatisfiable, which is not a very interesting case. Since we do not consider derived predicates within invariants, we can ignore axioms completely during invariant synthesis.

- Instead of using full-blown satisfiability and entailment tests, simpler tests are possible if they only err in the "conservative" direction. The IPC4 version of Fast Downward only employs simple structural entailment tests. This is due to scarcity of development time, not to conserve runtime, and we intend to extend the test to more complete logical reasoning.

One final subtlety concerns the semantics of PDDL operators with conflicting effects. Note that our balance test does not special-case the possibility that e.atom equals e'.atom, i. e. that the same atom is added and deleted. For the PDDL semantics that we adhere to, an operator which would add and delete the same atom would be invalid and thus inapplicable, not threatening any monotonicity properties. We call this the *consistent effect semantics*. However, under another commonly accepted semantics, the add effect would "win" in such a case. We call this the *add-after-delete semantics*. Using the *add-after-delete* semantics, we would need to add the condition e.atom $\neq e'$.atom before the entailment test in *check-operator-unbalanced*. We believe that there is no commonly agreed "correct" semantics for PDDL with regard to this issue: For some standard benchmarks, such as MPRIME, only the consistent effect semantics are reasonable, while for others, such as ROVERS, only the add-after-delete semantics make sense. Currently, Fast Downward usually follows the consistent effect semantics, but can be forced to partially switch to add-after-delete semantics by changing an internal variable. For future versions of the program, we plan to provide a command-line option to select the intended meaning.

algorithm refine-property(V, Φ):
Select some schematic operator o such that check-operator-unbalanced(o, V, Φ) fails.
for each atom φ' over variables from V and at most one other variable:
$\Phi' := \Phi \cup \{\varphi'\}$
Simplify Φ' by removing atoms that are covered by other atoms.
(These can never contribute to the weight of an instance of $\langle V, \Phi' \rangle$.)
Simplify Φ' by removing unused parameters.
if check-operator-unbalanced(o, V, Φ') does not fail:
Add $\langle V, \Phi' \rangle$ to the set of candidate properties.

Figure 13: Algorithm for refining an unbalanced monotonicity property $\langle V, \Phi \rangle$. The algorithm is not called if some operator o is too heavy for $\langle V, \Phi \rangle$.

4.3.3 REFINING FAILED PROPERTIES

As indicated in the overview of the invariant synthesis algorithm, we do not give up immediately if we cannot prove a given property to be an invariant. Instead, we try to *refine* it by adding atoms that can restore balance. In algorithmic terms, whenever we reject a monotonicity property $\langle V, \Phi \rangle$, we try to generate a set of new candidates of the form $\langle V, \Phi \cup \{\varphi'\} \rangle$.

Whether or not this is promising depends on the reason why the candidate was rejected. If it was rejected because an operator is too heavy, then no possible refinement can change this fact, and we give up on the property completely. If, however, it was rejected because of unbalanced operators, there is hope that we can deal with the flaw by adding new atoms that can match some delete effect of the threatening operator, balancing the unbalanced add effect.

This is all there is to the refinement of monotonicity properties; the algorithm is shown in Fig. 13. The actual implementation in Fast Downward does not generate all possible refining atoms φ' naïvely, but rather uses information from the set of delete effects of the threatening operator o and the failed call to *check-operator-unbalanced* to only create candidates for φ' for which there is a chance that the new balance check will succeed. Since this is conceptually straight-forward but somewhat tricky to implement, we do not go into more detail about this technique.

Instead, let us return to the LOGISTICS example. Recall that property (1), $\forall x\, \mathtt{at}(x, l) \downarrow$, is threatened by the operator `unload-truck`, whose add effect $\mathtt{at}(x, l)$ is unbalanced. The operator has only one delete effect, namely $\neg\mathtt{in}(x, t)$. Indeed, $\mathtt{in}(x, t)$ is a suitable candidate for φ' without further variable renaming, since the `unload-truck` operator is balanced with regard to the refined property $\forall x\, \mathtt{at}(x, l) + \mathtt{in}(x, t) \downarrow$. So we add this property to the set of candidates. At a later stage, it will be considered by *prove-invariant*, which will show that it is an invariant.

By contrast, the other two candidates cannot be suitably refined. In order to refine property (3), $\forall x, l\, \mathtt{at}(x, l) \downarrow$, to balance the `drive-truck` operator, we would need to add the atom $\mathtt{at}(x, l')$. However, this covers the original atom $\mathtt{at}(x, l)$, so that we need to remove it, and this in turn leads to parameter l being unused. Therefore, the refined invariant simplifies to $\forall x\, \mathtt{at}(x, l') \downarrow$, which is isomorphic to the previously considered candidate (1), and is hence not considered again. A similar argument shows that refinements of property (2) are quickly rejected.

LOGISTICS	$\forall x\ \texttt{at}(x,l) + \texttt{in}(x,t) \downarrow$
BLOCKSWORLD	$\forall \cdot\ \texttt{handempty}() + \texttt{holding}(b) \downarrow$
	$\forall b\ \texttt{holding}(b) + \texttt{clear}(b) + \texttt{on}(b',b) \downarrow$
	$\forall b\ \texttt{holding}(b) + \texttt{ontable}(b) + \texttt{on}(b,b') \downarrow$
GRID	$\forall \cdot\ \texttt{armempty}() + \texttt{holding}(k) \downarrow$
	$\forall \cdot\ \texttt{at-robot}(l) \downarrow$
	$\forall \cdot\ \texttt{open}(d) + \texttt{locked}(d) \downarrow$
	$\forall \cdot\ \texttt{locked}(d) \downarrow$
	$\forall d\ \texttt{open}(d) + \texttt{locked(d)} \downarrow$
	$\forall d\ \texttt{locked}(d) \downarrow$
	$\forall k\ \texttt{holding}(k) + \texttt{at(k,l)} \downarrow$

Figure 14: Invariants found in some standard benchmark domains.

4.3.4 EXAMPLES

This concludes our description of the invariant synthesis step. To give an impression of the kind of invariants found by the algorithm, Fig. 14 shows some of the results obtained on domains from the standard benchmark suite. The invariants found in the GRID domain are most interesting, as they include some monotonicity information that is *not* a mutex: The third (fourth) GRID invariant states that the total number of open and locked doors (the number of locked doors) never increases, and the sixth invariant states that a door which is not locked can never become locked.

4.4 Grounding

Having computed monotonicity invariants, we next want to obtain a grounded representation of the normalized PDDL task.

Definition 10 ***Grounded PDDL tasks***
*A **grounded PDDL task** is a PDDL task such that all literals occurring in the goal formula, axioms and operators are ground literals.*

Before performing the actual axiom and operator instantiation that yields the grounded representation, we try to determine which ground atoms of the PDDL task can actually become true. In a typical planning task, most ground atoms can never be true, either because they are not type-correct (e. g. $\texttt{at}(\texttt{vehicle1}, \texttt{vehicle2})$), or for more subtle reasons (e. g. $\texttt{at}(\texttt{vehicle1}, \texttt{loc1})$, where there is no path from the initial location of `vehicle1` to `loc1`). Instantiating operators or axioms in such a way that their preconditions or bodies are necessarily false in every reachable state would be wasteful.

Determining whether or not a given atom can ever be true is as difficult as planning itself, but an over-approximation of the set of reachable atoms can be computed efficiently based on the idea of *relaxed planning tasks* in the sense of HSP and FF (Bonet & Geffner, 2001; Hoffmann & Nebel, 2001). Instead of computing the set of reachable atoms of a PDDL task Π itself, we compute the reachable atoms of a relaxed planning task $\mathcal{R}(\Pi)$, which differs from Π as follows:

- Negative literals in axiom bodies, operator preconditions, effect conditions and goal condition are assumed to be always true.
- Delete effects of operators are ignored.

It is easy to see that the set of reachable atoms of $\mathcal{R}(\Pi)$ is a superset of the set of reachable atoms of Π, so any ground atoms not reachable in $\mathcal{R}(\Pi)$ need not be represented in the grounded version of Π.

The nice property of relaxed planning tasks is that computing their reachable atoms is conceptually simple. Nevertheless, this step is the most time-critical part of the whole translation component, because the set of reachable atoms can be huge in some of the benchmark domains, especially those with a comparatively simple logical structure like LOGISTICS or SATELLITE. Therefore, it is important for us to compute reachable atoms efficiently. This is what the *Horn exploration* algorithm is designed for. We will spend most of this section discussing this algorithm, finishing with a few remarks on operator instantiation.

4.4.1 OVERVIEW OF HORN EXPLORATION

The idea of Horn Exploration is to encode the atom reachability problem into a set of logical facts and rules, i. e. as a logic program, and to efficiently compute the ground atoms of the canonical model of that logic program, which is the set of ground atoms implied by the logic rules. The algorithm consists of three steps: Generating the logic program, translating it into a normal form, and computing its canonical model. Before going into detail for each of these steps, let us formally define what we mean by a logic program:

Definition 11 ***Positive logic programs***
Let $\mathcal{L}$ be a set of predicate, constant and variable symbols in first-order logic.

*A **positive Horn clause** over $\mathcal{L}$ is a formula of the form $\varphi_1 \wedge \cdots \wedge \varphi_k \rightarrow \psi$ ($k \geq 0$), where φ_i and ψ are (usually not ground) atoms over $\mathcal{L}$. It can be written as φ **:**$-\varphi_1, \dots, \varphi_k$. Using this notation, φ is called the **head** and $\varphi_1, \dots, \varphi_k$ is called the **body** of the clause.*

*A **positive logic program** (PLP) is a pair $\langle \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{F}$ is a set of ground atoms over $\mathcal{L}$ called the set of **facts** and $\mathcal{R}$ is a set of positive Horn clauses over $\mathcal{L}$ called **rules**.*

*The **canonical model** of a PLP $\langle \mathcal{F}, \mathcal{R} \rangle$ is the set of all ground atoms φ with $\mathcal{F} \cup \mathcal{R} \models \varphi$.*

Next, we show how to translate the reachability problem into a positive logic program. Afterwards, we demonstrate how to translate this logic program into a particularly simple form and how to compute the canonical model of the simplified logic program efficiently.

4.4.2 GENERATING THE LOGIC PROGRAM

Due to the fact that the PDDL task has been normalized, generating the logic program is conceptually easy. A ground atom is reachable in the relaxed task iff it is true in the initial state or there exists some axiom or operator of the relaxed task that can make it true. Therefore, the set of facts of the logic program is formed by the atoms in the initial state of the planning task, and the set of rules is derived from the axiom and operator definitions. Additionally, we introduce a rule for the goal of the planning task to detect solvability of the relaxed task; if it is unsolvable, the original task is unsolvable too, which we can report immediately and stop planner execution.

Recall from Section 4.2 that at this stage, all conditions occurring in the PDDL task are conjunctions of literals. For such conjunctions φ, we denote the conjunction of all *positive* literals of φ by φ^+. In the context of logic programs, we follow the PROLOG convention of using uppercase letters for first-order variables and lower-case letters for constants. The exploration rules for a normalized PDDL tasks are generated as follows:

- *Axioms:* For schematic axioms $a = \varphi \,\texttt{:-}\, \psi$ with $\psi^+ = \psi_1^+ \wedge \cdots \wedge \psi_m^+$ and free$(\varphi) \cup$ free$(\psi) = \{X_1, \dots, X_k\}$, we generate the *axiom applicability rule*

  $$a\texttt{-applicable}(X_1, \dots, X_k) \;\texttt{:-}\; \psi_1^+, \dots, \psi_m^+.$$

  and the *axiom effect rule*

  $$\varphi \;\texttt{:-}\; a\texttt{-applicable}(X_1, \dots, X_k).$$

- *Operators:* For schematic operators o with parameters $\{X_1, \dots, X_k\}$ and precondition φ with $\varphi^+ = \varphi_1^+ \wedge \cdots \wedge \varphi_m^+$, we generate the *operator applicability rule*

  $$o\texttt{-applicable}(X_1, \dots, X_k) \;\texttt{:-}\; \varphi_1^+, \dots, \varphi_m^+.$$

  and for each add effect e of o adding the atom ψ with quantified variables $\{Y_1, \dots, Y_l\}$ and effect condition φ with $\varphi^+ = \varphi_1^+ \wedge \cdots \wedge \varphi_m^+$, we generate the *effect trigger rule*

  $$e\texttt{-triggered}(X_1, \dots, X_k, Y_1, \dots, Y_l) \;\texttt{:-}\; o\texttt{-applicable}(X_1, \dots, X_k), \varphi_1^+, \dots, \varphi_m^+.$$

  and *effect rule*

  $$\psi \;\texttt{:-}\; e\texttt{-triggered}(X_1, \dots, X_k, Y_1, \dots, Y_l).$$

- *Goal rule:* For the goal φ with $\varphi^+ = \varphi_1^+ \wedge \cdots \wedge \varphi_m^+$, we generate the *goal rule*

  $$\texttt{goal-reachable}() \;\texttt{:-}\; \varphi_1^+, \dots, \varphi_m^+.$$

The correctness of these rules should be evident. The reader might wonder why we sometimes introduce new predicates that do not seem necessary. For example, axiom applicability rule and axiom effect rule could be combined into a single rule without introducing the auxiliary predicate a`-applicable`. The purpose of these predicates is to track which axioms and operators must be instantiated when grounding the PDDL task. For example, in the LOGISTICS domain, we will not generate a ground operator `(fly-airplane plane1 loc1 loc3)` if `loc3` is not an airport location, since in this case the canonical model of the logic program does not include the atom `fly-airplane-applicable(plane1, loc1, loc3)`. The operator applicability predicates serve the additional purpose of "factoring out" common subexpressions. Without them, all operator preconditions would need to be repeated in each effect trigger rule (or effect rule, if effect trigger rules were similarly eliminated).

4.4.3 TRANSLATING THE LOGIC PROGRAM TO NORMAL FORM

After the logic program has been generated, it is translated into the following normal form:

Definition 12 ***Normal form for positive logic programs***
A rule of a positive logic program is called ***variable-unique*** *if no variable occurs twice in any atom of the head or body. In other words, atoms like $P(X, Y, X)$ are forbidden because variable X occurs twice. (Repetitions of constants are allowed.)*

A variable-unique rule is called a ***projection rule*** *if it is of the form φ :- φ_1 with vars(φ) $\subseteq$ vars(φ_1). In other words, projection rules are unary rules where all variables in the head occur in the body.*

A variable-unique rule is called a ***join rule*** *if it is of the form φ :- φ_1, φ_2 with vars(φ_1) $\cup$ vars(φ_2) $=$ vars(φ) $\cup$ (vars(φ_1) $\cap$ vars(φ_2)). In other words, join rules are binary rules where all variables occurring in the head occur in the body, and all variables occurring in the body but not in the head occur in* both *atoms of the body.*

A positive logic program is ***in normal form*** *iff all rules are either projection rules or join rules.*

The names of the rule types in Definition 12 are reminiscent of the related database-theoretic operations from relational algebra: Projection rules correspond to the projection operator π and join rules correspond to the natural join operator $\bowtie$ (or strictly speaking, a combination of natural join and projection). We will now describe how to convert the positive logic program from the previous section into normal form.

First, we eliminate duplicate variable occurrences as follows: If any rule contains atoms with duplicate occurrences of the same variable X, we change one occurrence of X in any such atom into a new variable X' and add the atom $\texttt{equals}(X, X')$ to the body of the rule. We repeat until no further such transformations are possible. If we needed to introduce any `equals` atoms, we add the fact $\texttt{equals}(o, o)$ to the initial state for each object o of the planning task.

Second, for any variable X that occurs in the head but not in the body of a rule, we add the atom $\texttt{object}(X)$ to the rule body. (Remember from Section 4.2.1 that $\texttt{object}(o)$ is true for any object o of the planning task.)

Third, all rules with an empty body are converted into facts. Their heads must be ground atoms because all variables occurring in the head must occur in the (in this case, empty) body after the previous transformation.

After these transformations, all remaining unary rules are projection rules; we still need to normalize rules with two or more atoms in the body. As a first step towards this goal, we determine if the body of such a rule contains any variables that occur in no other atom of the rule, neither in the body nor in the head. If this is the case, such variables are projected away as follows: We are given the rule φ :- $\varphi_1, \dots, \varphi_k$, where vars($\varphi_i$) $= \{X_1, \dots, X_k\}$ contains variables not present in any of the other atoms, say $\{X_{j+1}, \dots, X_k\}$. Then we introduce a new predicate p and replace the original rule by the two rules φ :- $\varphi_1, \dots, \varphi_{i-1}, p(X_1, \dots, X_j), \varphi_{i+1}, \dots, \varphi_k$ and $p(X_1, \dots, X_j)$:- φ_i.

After this transformation, all binary rules are valid join rules. In the last normalization step, we split rules with $m > 2$ atoms in the body into $m-1$ join rules by applying the *greedy join algorithm*, illustrated in Fig. 15. The algorithm iteratively picks two atoms from the rule body and joins them, introducing a new predicate for the result of the join and replacing the two atoms in the rule body by an instance of that new predicate. This process is repeated until the body of the rule no longer contains more than two atoms.

The order in which atoms are joined is critical for the speed of evaluating the join rules. To see this, consider the rule $p(X)$:- $q(X), s(X, Y), t(Y)$. One possible decomposition into join rules is given by the rules $u(X)$:- $s(X, Y), t(Y)$ and $p(X)$:- $q(X), u(X)$. Another is given by the rules $v(X, Y)$:- $q(X), t(Y)$ and $p(X)$:- $s(X, Y), v(X, Y)$. We can expect that the canonical model of the first decomposition contains relatively few instances of the intermediate predicate u, maybe about as many as it contains instances of t. On the other hand, the canonical model of the second

algorithm greedy-join(*rule*):
    while $|\textit{rule}.\text{body}| > 2$:
        Choose $\varphi, \varphi' \in \textit{rule}.\text{body}$ such that $\varphi \neq \varphi'$ and join-cost$(\textit{rule}, \varphi, \varphi')$ is minimal.
        $X_1, \dots, X_k :=$ join-vars$(\textit{rule}, \varphi, \varphi')$.
        Generate a new predicate symbol p with arity k.
        Generate a new join rule $p(X_1, \dots, X_k) \,\mathbf{:-}\, \varphi, \varphi'$.
        $\textit{rule}.\text{body} := \textit{rule}.\text{body} \setminus \{\varphi, \varphi'\} \cup \{p(X_1, \dots, X_k)\}$

function join-vars(*rule*, φ, φ'):
    { Compute the relevant variables for the predicate generated by joining φ and φ'. }
    return $\text{vars}(\{\varphi, \varphi'\}) \cap \text{vars}(\{\textit{rule}.\text{head}\} \cup \textit{rule}.\text{body} \setminus \{\varphi, \varphi'\})$.

function join-cost(*rule*, φ, φ'):
    new-arity $:= |\text{join-vars}(\textit{rule}, \varphi, \varphi')|$
    max-old-arity $:= \max(|\text{vars}(\varphi)|, |\text{vars}(\varphi')|)$
    min-old-arity $:= \min(|\text{vars}(\varphi)|, |\text{vars}(\varphi')|)$
    return $(\text{new-arity} - \text{max-old-arity}, \text{new-arity} - \text{min-old-arity}, \text{new-arity})$.
    { Cost estimates are triples which are compared lexicographically.
      We prefer joins where "new arity" − "max old arity", i. e. the increase in arity, is small
      and consider the other criteria only in case of ties. }

Figure 15: The greedy join algorithm for decomposing a rule into join rules.

decomposition contains as many instances of the intermediate predicate v as the product of the number of instances of q and t, which can be much higher.

Since the performance of our algorithm for computing the canonical model of a logic program is closely related to the model size, we prefer to generate smaller intermediate results. The greedy join algorithm tries to achieve this goal by preferring to join atoms that contain many common variables and lead to intermediate predicates of low arity.

4.4.4 Computing the Canonical Model

Having translated the logic program into normal form, we are ready to compute the canonical model. We use a queue-based approach, distinguishing between reachable atoms that have already been processed, which means that the consequences of their being reachable have already been evaluated (*closed atoms*), and reachable atoms that still need to be processed (*open atoms*). Open atoms are those that are currently stored in the queue, while closed atoms are those that were enqueued once, but no longer are.

Our algorithm, shown in Fig. 16, stores open atoms in the *queue* variable, while both open and closed atoms are stored in the result variable *canonical-model*. Additionally, it uses the following data structures:

- *Rule matcher*: A rule matcher is an indexing structure that supports efficient unification queries on the bodies of logic programs. When given a ground atom a, the rule matcher determines all projection rules $\varphi \,\mathbf{:-}\, \varphi_1$ and join rules $\varphi \,\mathbf{:-}\, \varphi_1, \varphi_2$ such that φ_1 or φ_2 unifies

```
algorithm calculate-canonical-model(F, R):
    for each join rule r ∈ R:
        r.index_1 := make-empty-hashtable()
        r.index_2 := make-empty-hashtable()
    rule-matcher := build-rule-matcher(R)
    queue := make-queue(F)
    canonical-model := F
    { In the following, enqueuing a fact means adding it to queue and canonical-model
      if it is not yet an element of canonical-model. }
    while queue is not empty:
        current-fact := queue.pop()
        for each match m ∈ rule-matcher.match(current-fact):
            if m refers to φ_1 in a projection rule r = φ :- φ_1:
                Let α be the variable assignment for which α(φ_1) = current-fact.
                Enqueue α(φ).
            else if m refers to φ_1 in a join rule r = φ :- φ_1, φ_2:
                Let α be the variable assignment for which α(φ_1) = current-fact.
                key := α restricted to vars(φ_1) ∩ vars(φ_2)
                Add α to r.index_1[key].
                Enqueue (α ∪ β)(φ) for each β ∈ r.index_2[key].
            else if m refers to φ_2 in a join rule r = φ :- φ_1, φ_2:
                { Handled analogously to the previous case. }
```

Figure 16: Computing the canonical model of a positive logic program $\langle \mathcal{F}, \mathcal{R} \rangle$ in normal form.

with a, i. e. such that it is possible to substitute objects for variables in φ_1 or φ_2 in such a way that a is obtained. The rule matcher reports the matched rules and whether φ_1 or φ_2 was matched (if both unify with a, two matches are generated).

Note that matching ground atoms to the rules they can trigger is simple if the rules do not contain constants in the body. Unfortunately, some of the IPC4 benchmarks contain a huge number of operator schemas involving constants (most importantly, the STRIPS formulation of the AIRPORT domain), and an efficient indexing structure is important for those. Rule matchers are implemented as decision-tree like data structures very similar to *successor generators*, which are discussed in Section 5. Because of that similarity and because they are not central to the instantiation algorithm, we do not discuss rule matchers further.

- *Join rule indices:* Each join rule $r = \varphi \mathbin{:\!-} \varphi_1, \varphi_2$ maintains two hash tables $r.\text{index}_1$ and $r.\text{index}_2$ that map instantiations of the common variables of φ_1 and φ_2 to instantiations of the variables of φ_1 and φ_2, respectively.

  At any time (except during updates) and for any assignment *key* to the common variables of φ_1 and φ_2, $r.\text{index}_1[\textit{key}]$ contains those variable mappings $\alpha \supseteq \textit{key}$ for the variables of φ_1 for which $\alpha(\varphi_1)$ belongs to the closed set. Similarly, $r.\text{index}_2[\textit{key}]$ contains those variable mappings $\beta \supseteq \textit{key}$ for the variables of φ_2 for which $\beta(\varphi_2)$ belongs to the closed set.

  This information can be exploited for quickly determining all possible instantiations of φ_2 that match a given instantiation of φ_1, or vice versa, as is done in the algorithm. Note that the variable assignment $\alpha \cup \beta$ considered in the algorithm is indeed a function, since α and β agree on all variables for which they are both defined.

To motivate the soundness of *compute-canonical-model*, we state an important invariant which holds before and after each iteration of the **while** loop: *All non-closed atoms which can be derived in one step from the closed atoms using the rules of the logic program $\langle \mathcal{F}, \mathcal{R} \rangle$ are open atoms.* This implies that upon termination of the algorithm, when there are no more open atoms and hence *canonical-model* holds exactly the set of closed atoms, the model is closed under application of $\mathcal{R}$. Because it also contains all facts from $\mathcal{F}$ and only contains facts that can be derived from $\mathcal{F}$, it thus contains exactly the canonical model of $\langle \mathcal{F}, \mathcal{R} \rangle$.

The invariant is obviously true initially, since there are no closed atoms at the beginning of the algorithm. With our descriptions of the data structures of *compute-canonical-model*, the reader should have no trouble verifying that it remains true after each iteration of the **while** loop.

This concludes our discussion of the Horn exploration algorithm. One final word on performance: If we assume that the arity of predicates in the logic program is bounded by a constant, then all basic operations of *calculate-canonical-model* can be performed in constant time. The runtime of the algorithm then typically scales roughly linearly in the combined size of its input and output (the computed canonical model). However, runtime can be worse if there are many situations where the algorithm tries to enqueue an atom that is already part of the canonical model.

4.4.5 AXIOM AND OPERATOR INSTANTIATION

With the help of the canonical model, instantiating axioms and operators is very straight-forward. To compute the grounded representation, we scan through the set of ground atoms in the canonical model in the order in which they were generated.

When encountering an atom p over a predicate a-applicable where a is a schematic axiom, we generate a ground instance of a with the parameters substituted with the arguments of p.

When encountering an atom p over a predicate o-applicable where o is a schematic operator, we generate a ground instance of o without effects. Like in the case of axioms, the parameters of the operator are substituted with the arguments of p, and the precondition is instantiated accordingly.

Finally, when encountering an atom p over a predicate e-triggered where e is an effect of some operator o, we look up the set of already generated ground instances of schematic operator o to find a ground operator whose parameters match the corresponding arguments of atom p. We can always find such an instance because e-triggered can only be derived *after* o-applicable has been derived for the respective operator. Having found the ground operator, we attach the effect obtained by instantiating the variables in e with the respective arguments of p.

After a single pass through the canonical model, we have thus generated a grounded PDDL task which is equivalent to the normalized PDDL task we started with.

4.5 MPT generation

Together with the invariants synthesized earlier, the grounded PDDL task generated in the previous stage provides all the information we need for transforming the STRIPS task into a multi-valued planning representation.

Recall from Definition 2 that a multi-valued planning task (MPT) is given by a 5-tuple $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$ of variables $\mathcal{V}$, each with an associated finite domain, initial state s_0 and goal $s_\star$, axioms $\mathcal{A}$ and operators $\mathcal{O}$. We start by defining suitable variables and variable domains; everything else then more or less falls into place.

4.5.1 VARIABLE SELECTION

Each variable of the generated MPT corresponds to one or more (reachable) ground atoms of the STRIPS task. We start by enumerating the set $\mathcal{P}$ of all such atoms, partitioned into atoms $\mathcal{P}_f$ which are instances of *modifiable fluent predicates* or *derived predicates* and atoms $\mathcal{P}_c$ which are instances of *constant predicates* (cf. Section 4.3.1).

We want to represent as many ground atoms by a single state variable as possible. To achieve this, we first determine the set of *mutex groups* induced by the computed invariants. Mutex groups are computed in a straight-forward manner by instantiating the monotonicity invariants in all possible ways, checking for each if it has weight 1 in the initial state, and if so, which atoms from $\mathcal{P}_f$ it covers. The algorithm is shown in Fig. 17; the actual implementation in Fast Downward uses an indexing structure for efficiently determining the set of reachable atoms covered by a given invariant instance.

Normally, not every mutex group will correspond to an MPT state variable, since the same atom can be part of several mutex groups, but of course only needs to be encoded once. As an example of this phenomenon, consider Fig. 18, which shows the mutex groups of a BLOCKSWORLD task with four blocks. If, for example, we decide to encode mutex groups (1)–(4) with four multi-valued state variables, then we only need to encode one atom from each of the other groups, since all instance of the on and holding predicates are already represented. Therefore, the translator would first generate four state variables with domains consisting of seven values each, namely holding(x), clear(x), on(a, x), on(b, x), on(c, x), on(d, x) and the seventh option "none of the other six is true". (Of these seven values, two — block x being on top of itself and none of the six atoms being

algorithm compute-mutex-groups(*invariants*, $\mathcal{P}_f$, s_0):
  for each invariant $\mathcal{I} \in$ *invariants*:
    for each instance α of $\mathcal{I}$:
      if weight$(\alpha, s_0) = 1$:
        Create a mutex group containing all atoms in $\mathcal{P}_f$ covered by α.

Figure 17: Computing mutex groups from the set of monotonicity invariants *invariants*, the set of reachable atoms $\mathcal{P}_f$ and the initial state s_0.

(1) $\{\texttt{holding(a)}, \texttt{clear(a)}, \texttt{on(a, a)}, \texttt{on(b, a)}, \texttt{on(c, a)}, \texttt{on(d, a)}\}$
(2) $\{\texttt{holding(b)}, \texttt{clear(b)}, \texttt{on(a, b)}, \texttt{on(b, b)}, \texttt{on(c, b)}, \texttt{on(d, b)}\}$
(3) $\{\texttt{holding(c)}, \texttt{clear(c)}, \texttt{on(a, c)}, \texttt{on(b, c)}, \texttt{on(c, c)}, \texttt{on(d, c)}\}$
(4) $\{\texttt{holding(d)}, \texttt{clear(d)}, \texttt{on(a, d)}, \texttt{on(b, d)}, \texttt{on(c, d)}, \texttt{on(d, d)}\}$
(5) $\{\texttt{holding(a)}, \texttt{ontable(a)}, \texttt{on(a, a)}, \texttt{on(a, b)}, \texttt{on(a, c)}, \texttt{on(a, d)}\}$
(6) $\{\texttt{holding(b)}, \texttt{ontable(b)}, \texttt{on(b, a)}, \texttt{on(b, b)}, \texttt{on(b, c)}, \texttt{on(b, d)}\}$
(7) $\{\texttt{holding(c)}, \texttt{ontable(c)}, \texttt{on(c, a)}, \texttt{on(c, b)}, \texttt{on(c, c)}, \texttt{on(c, d)}\}$
(8) $\{\texttt{holding(d)}, \texttt{ontable(d)}, \texttt{on(d, a)}, \texttt{on(d, b)}, \texttt{on(d, c)}, \texttt{on(d, d)}\}$
(9) $\{\texttt{holding(a)}, \texttt{holding(b)}, \texttt{holding(c)}, \texttt{holding(d)}, \texttt{handempty()}\}$

Figure 18: Mutex groups for a BLOCKSWORLD task with four blocks. Some atoms, e. g. $\texttt{on(a, a)}$, are reachable in the relaxed task although they are never true in the "real" task.

true — are actually impossible.) Afterwards, it would encode the truth values of the remaining atoms $\texttt{ontable}(x)$ and $\texttt{armempty}()$ with binary state variables.

In this case, there was at least one atom in each mutex group that was unique to this particular group, so that the resulting encoding is not much better than an encoding which simply takes all mutex groups and introduces a state variable for each. However, in other cases, one group can be completely covered by others; examples of this can be found in the AIRPORT domain. In this case we would like to cover the set of reachable atoms with as few state variables as possible.

Unfortunately, set cover problems of this kind are **NP**-complete (Garey & Johnson, 1979, problem SP5), so we limit our covering efforts to the greedy algorithm shown in Fig. 19. Iteratively, we pick a mutex group P of maximal cardinality and introduce a new MPT state variable with domain $P \cup \{\perp\}$, where $\perp$ stands for "none of the elements of P is true". We then remove all covered elements from all other mutex groups, removing groups that no longer contain more than one element. This process is repeated until all mutex groups have been removed. At this stage, the remaining uncovered atoms p are represented by binary variables with domain $\{p, \perp\}$.

After execution of the algorithm, each reachable atom $p \in \mathcal{P}_f$ corresponds to exactly one MPT variable v such that $v = p$ is true in an MPT state iff p is true in the corresponding state of the PDDL task. We use the notation $\mathit{var}(p) = v$ to state this relation between p and v. With this information, we can now go about converting the rest of the PDDL task to the MPT representation.

algorithm choose-variables($\mathcal{P}_f$, *mutex-groups*):
  uncovered $:= \mathcal{P}_f$
  while *mutex-groups* $\neq \emptyset$:
    Pick a mutex group P of maximal cardinality.
    Create an MPT variable v with domain $\mathcal{D}_v = P \cup \{\perp\}$.
    uncovered := *uncovered* $\setminus P$.
    mutex-groups $:= \{ P' \setminus P \mid P' \in$ *mutex-groups* $\wedge |P' \setminus P| \geq 2 \}$
  Create an MPT variable v with domain $\{p, \perp\}$ for all remaining elements of *uncovered*.

Figure 19: Greedy algorithm for computing the MPT variables and variable domains.

4.5.2 CONVERTING THE INITIAL STATE

We start by converting the initial state, which is the easiest step. For each atom $p \in \mathcal{P}_f$ that is in the initial state, we set the initial value of $var(p)$ to p. MPT variables v for which there is no initial state atom p with $var(p) = p$ are initialized to $\perp$. Note that different initial state atoms p, p' must satisfy $var(p) \neq var(p')$, because p and p' could only be represented by the same MPT variable if they were mutually exclusive, which implies their not being in the initial state together. Therefore, the converted initial state is indeed well-defined.

4.5.3 CONVERTING OPERATOR EFFECTS

Translating the state changes incurred by operator effects requires some care. For add effects setting an atom p to true, conversion is easy: Such an effect is always translated to an MPT effect setting $\mathrm{var}(p)$ to p, because we know p to be true after operator application if the effect fires.

However, for delete effects setting an atom p to false, the correct translation is not as clear. We cannot simply set $\mathrm{var}(p)$ to $\perp$ ("none of the variables represented by $\mathrm{var}(p)$ is true") unconditionally, because this is not always correct: It could be the case that another effect of the same operator triggers simultaneously and adds another atom represented by the same variable, or that p *was not true* when the operator was applied, but some other atom represented by $\mathrm{var}(p)$ was.

Therefore, the correct translation is to set $\mathrm{var}(p)$ to $\perp$ only if we know that p was previously true and that no effect adding an atom represented by $\mathrm{var}(p)$ triggers simultaneously, and not to do anything if this is not the case. If the other effects of the operator that add atoms represented by $\mathrm{var}(p)$ have effect conditions $\chi_1, \ldots, \chi_k$, then this is achieved by adding $p \wedge \neg\chi_1 \wedge \cdots \wedge \neg\chi_k$ to the effect condition of the delete effect.

If some of the formulas χ_i are proper conjunctions, this results in an effect condition which is not a conjunction of literals. In this case, we introduce a new derived variable v_i that evaluates to true whenever $\neg\chi_i$ is true, and use v_i in the effect condition instead.

All things considered, this conversion of delete effects looks very complicated, and indeed in most cases easier translations are possible. For this purpose, we detect two common special cases, with which we deal differently:

- If we see that whenever the delete effect triggers, some add effect affecting the same variable must trigger as well, because it has the same or a more general effect condition, then we do not need to represent the delete effect in the MPT at all. The add effect will take care of the value change of its affected variable.

- On the other hand, if we see that no add effect affecting the same variable can trigger at the same time, because no such effect exists or each of their effect conditions is inconsistent with the condition of the delete effect, then we can convert the delete effect to an effect setting var(p) to $\perp$. If p is not already part of the operator precondition or effect condition, we must add it to the effect condition to make sure that var(p) is only cleared if it was previously set to p.

In most cases, translating delete effects is straight-forward because the two simpler cases are by far more common than the general case. In particular, for operators without conditional effects, one of the special cases always applies.

4.5.4 CONVERTING CONDITIONS

The third major translation step is the conversion of grounded conditions of the PDDL task, which occur in the goal, in operator preconditions and effect conditions and in axiom bodies.

To translate a grounded condition, we first check if it contains any atoms not in $\mathcal{P}_f$. These have constant truth values, so that the condition can be simplified accordingly. If this leads to a constant false condition, we react accordingly (for the goal, we report that the problem is unsolvable; for axiom bodies, operator preconditions or effect conditions, we remove the axiom, operator or effect).

Having considered trivially false conditions, we translate each positive literal p in the condition to the pairing $\mathit{var}(p) = p$. Translating negative literals $\neg p$ is slightly more tricky. Recall the BLOCKSWORLD example discussed earlier, where we generated the MPT state variable v with $\mathcal{D}_v = \{\mathtt{holding}(\mathtt{a}), \mathtt{clear}(\mathtt{a}), \mathtt{on}(\mathtt{a},\mathtt{a}), \mathtt{on}(\mathtt{b},\mathtt{a}), \mathtt{on}(\mathtt{c},\mathtt{a}), \mathtt{on}(\mathtt{d},\mathtt{a}), \perp\}$, and consider a condition including the atom $\neg\mathtt{on}(\mathtt{c},\mathtt{a})$. If the condition also contains some positive literal concerning variable v, for example the atom $\mathtt{clear}(\mathtt{a})$, then we do not need to encode $\neg\mathtt{on}(\mathtt{c},\mathtt{a})$ at all, because it is implied by the other literal. However, otherwise there is no simple way to represent $\neg\mathtt{on}(\mathtt{c},\mathtt{a})$ as an MPT condition. We would need to write something like $v \neq \mathtt{on}(\mathtt{c},\mathtt{a})$, but conditions of this form are not supported by the representation.

Therefore, in situations like this, similar to what we did when translating difficult effect conditions that arise for complicated delete effects, we introduce a new derived variable $\mathtt{not}\text{-}p$ with domain $\{\top, \perp\}$ and generate an axiom $(\mathtt{v} = d) \rightarrow (\mathtt{not}\text{-}p := \top)$ for each value $d \in \mathcal{D}_v \setminus \{p\}$. The pairing $\mathtt{not}\text{-}p = \top$ can then serve as a translation of the literal $\neg p$.

If we wanted to avoid introducing new axioms, we could further normalize the PDDL task so that no negative literals occur in conditions. There are well-known translation methods to achieve such a normal form, but for our purposes, our method has the advantage that no new non-derived state variables are introduced, keeping the memory requirements of search states small.

4.5.5 COMPUTING AXIOM LAYERS

As a final translation step, we must compute the axiom layers for the MPT representation so that the semantics match with those of stratified logic programs (cf. Definitions 1 and 2).

This is done as follows: Whenever the body of an axiom a includes the condition $v = \perp$ for some derived variable v, then all axioms a' with affected variable v must be evaluated before a, i. e. we introduce an ordering constraint $a' \prec a$. If the axiom definitions of the original PDDL task corresponded to a stratifiable logic program, then the graph containing all such ordering constraints will be acyclic. Thus, we can use a topological search algorithm to assign the individual axioms to axiom layers: The first axiom layer contains all axioms without predecessors in the graph, the second axiom layer contains all axioms whose predecessors belong to the first layer, and so on, until layers are assigned to all axioms.

4.5.6 GENERATING THE OUTPUT

Having partitioned the axioms into layers, we have finished translating the PDDL task. Before generating output, the translator applies a few post-processing techniques to simplify the generated task where possible.

Most importantly, if there are two axioms with the same head, $a = (\mathit{cond} \to v := d)$ and $a' = (\mathit{cond}' \to v := d)$ with $\mathit{cond} \subset \mathit{cond}'$, then a is triggered whenever a' is triggered, so a' is unnecessary. In such a case, which occurs frequently in domains where axioms encode transitive closures, we say that a *dominates* a' and only keep a. Similarly, we do not keep several copies of the same axiom which only differ in the order in which the conditions are listed.

Once post-processing is completed, the generated MPT is written to disk. For this purpose, Fast Downward uses a simple text format suitable for easy parsing by the other components of the planner, but too cryptic for human consumption. We are considering adding an option to the translator that generates output in a more human-readable format in the future. That way, users of the planner could get a better impression of how the input task is encoded and other researchers interested in working with multi-valued inputs would have an easier job using Fast Downward's translation component as a preprocessor.

4.6 Statistics for the translation component

Before moving on to the knowledge compilation component, let us mention some experimental data about the performance of the translation component and the size of the generated MPTs. We present more comprehensive results on the complete Fast Downward planner in Section 7. Here, we just want to provide some anecdotal information concerning the size that grounded MPTs can reach in practice and the time that is typically required for translation.

We ran our Python implementation of the translation component on a 3.066 GHz Intel Xeon CPU with a memory limit of 2 GB and managed to translate all 1442 propositional benchmarks of the first four international planning competitions (cf. Section 7 for a detailed description of the benchmark suite) without exhausting the memory limit or our patience. However, the largest PSR instance, namely the task PSR-LARGE #50, could only just be translated within the memory bound, consuming more than 1.9 GB of RAM before finishing after 32:40 minutes. However, with 61 842 relevant state variables, almost all of them being derived variables, this task is far from being solvable with current domain-independent planning technology anyway. For comparison, at IPC4, Fast Downward could only solve the PSR-LARGE instances up to #31. The largest of these instances comprises 5 807 relevant MPT state variables and needed 39 seconds for translation. Of the other competitors, the best system could solve 11 instances, of which the largest comprises 527 state variables. MPT translation took 4 seconds for this instance.

Fig. 20 provides an overview of the runtime and output characteristics for the translator. To get an impression of the size and translation cost of a "typical" task, the mean values, which are heavily influenced by some very large PSR tasks, are misleading. The comparatively high standard deviations show that the distributions are highly irregular, so the percentile information is probably more meaningful. To get an impression of what very large planning tasks look like, Fig. 21 provides information about the "largest" five input tasks according to each of the four measures *translation*

Measure	Mean	Std. Dev.	25% Perc.	Median	75% Perc.	Max
Translation time	12.35 s	82.52 s	0.39 s	1.19 s	4.28 s	1 960.0 s
Size of output	672 KB	3 253 KB	23 KB	93 KB	326 KB	81 748 KB
No. state variables	565	3 176	25	55	240	61 842
No. operators/axioms	7 046	38 308	289	1 009	3 469	989 250

Figure 20: Some statistics on the performance and output characteristics of the translation component. Results are given for the 1442 propositional benchmarks from the fully automated tracks of the first four international planning competitions.

Task	Translation time	Size of output	No. state variables	No. operators/axioms
PSR-LARGE #50	1 960.00 s	15 558 KB	61 842	116 413
SATELLITE #33	1 355.50 s	81 748 KB	1 375	989 250
PSR-LARGE #48	1 145.80 s	11 835 KB	48 210	88 772
PSR-LARGE #46	920.45 s	10 305 KB	40 357	76 991
SATELLITE #32	634.80 s	52 032 KB	1 115	638 665
PSR-LARGE #44	624.64 s	8 145 KB	32 174	61 689
PSR-LARGE #49	512.22 s	8 076 KB	36 790	68 548
SATELLITE #36	328.56 s	34 758 KB	1 091	428 109
SATELLITE #31	262.02 s	29 997 KB	865	368 990
SATELLITE #35	108.22 s	27 672 KB	1 073	342 193

Figure 21: Statistics for the five largest planning tasks in each of the four categories. There are fewer than 20 entries because some tasks are among the top five in several categories. Entries are sorted by decreasing translation time.

time, *size of translation output*, *no. of state variables in translated problems* and *no. of operators and axioms in translated problems*.

To estimate the potential speedup that could be obtained by reimplementing the translator in C++, we have compared the runtime of the "old" C++-based translator to the "new" translator on those 566 benchmarks that the old translator can handle. We only considered harder tasks, those for which either program required at least one second for translation, thus obtaining 254 data points. In all cases, the C++ translator computed its result faster. In 240 out of 254 cases, the observed speed ratio was between 6 and 12. The different speedups observed are shown in Fig. 22. Since we are comparing *different algorithms* as well as different programming languages, this data should be taken with a grain of salt. Nevertheless, we believe that reimplementing our translation algorithm in a lower-level language could easily lead to a speedup of about 10.

5. Knowledge compilation

The purpose of the knowledge compilation component is to set the stage for the search algorithms by compiling the critical information about the planning task into a number of data structures for

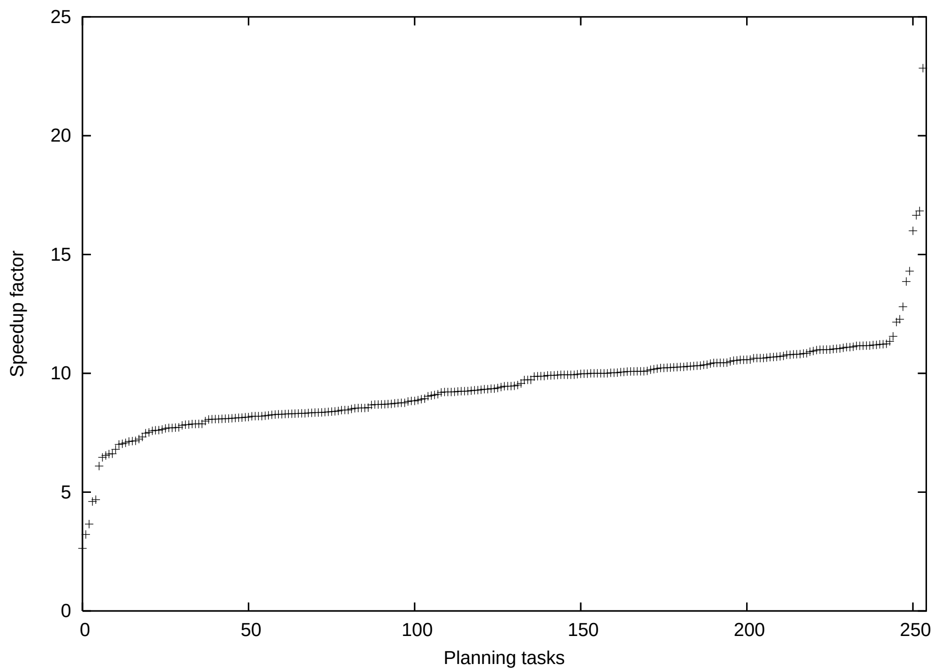

Figure 22: Observed difference in speed between the “old” and “new” translator. Each data point corresponds to one planning task. Data points are sorted by speedup. There was no visible correlation between size of input task and observed speedup, meaning that the asymptotical runtime of both implementations appears to be similar for the domains considered.

efficient access. In other contexts, computations of this kind are often called *preprocessing*, and indeed the implementation of the knowledge compilation component in the IPC4 version of Fast Downward is a program called `preprocess`. However, "preprocessing" is such a nondescript term that it can mean basically anything, so we decided to use a term that puts a stronger emphasis on the role of this module: To rephrase the critical information about the planning task in such a way that it is directly useful to the search algorithms. Of the three major building blocks of Fast Downward (translation, knowledge compilation, search), it is the least time-critical part, always requiring less time than translation and being dominated by search for all but the most trivial tasks.

Knowledge compilation comprises three items. First and foremost, we compute the *domain transition graph* of each state variable. The domain transition graph for a state variable encodes under what circumstances that variable can change its value, i. e. from which values in the domain there are transitions to which other values, which operators or axioms are responsible for the transition, and which conditions on other state variables are associated with the transition. Domain transition graphs are described in Section 5.1. They are a central concept for the computation of the CG heuristic, described in Section 6.1.

Second, we compute the *causal graph* of the planning task. Where domain transition graphs encode dependencies between values for a given state variable, the causal graph encodes dependencies between different state variables. For example, if a given location in a planning task can be unlocked by means of a key that can be carried by the agent, then the variable representation the lock state of the location is dependent on the variable that represents whether or not the key is being carried. This dependency is encoded as an arc in the causal graph. Like domain transition graphs, causal graphs are a central concept for the computation of the CG heuristic, giving it its name. The CG heuristic requires causal graphs to be acyclic. For this reason, the knowledge compilation component also generates an acyclic subgraph of the real causal graph when cycles occur. This amounts to a relaxation of the planning task where some operator preconditions are ignored. In addition to their usefulness for the CG heuristic, causal graphs are also a key concept of the *focused iterative-widening search* algorithm introduced in Section 6.4. We discuss causal graphs in Section 5.2.

Third, we compute two data structures that are useful for any forward-searching algorithm for MPTs, called *successor generator* and *axiom evaluator*. The successor generator computes the set of applicable operators and successor states of a given world state, while the axiom evaluator computes the values of the derived variables for a given reduced state. Both data structures are designed to do their job as quickly as possible, which is especially important for the focused iterative-widening search algorithm, which spends no time computing heuristic values of states and thus requires basic search operations like expanding a search node to be implemented efficiently. Successor generators and axiom evaluators are discussed in Section 5.3.

5.1 Domain transition graphs

The domain transition graph of a state variable is a representation of the ways in which the variable can change its value, and of the conditions that must be satisfied for such value changes to be allowed. Domain transition graphs were introduced by Jonsson and Bäckström in the context of SAS^+ planning (1998a). Our formalization of domain transition graphs generalizes the original definition to planning tasks involving axioms and conditional effects.

Definition 13 ***Domain transition graphs***
Let $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$ be a multi-valued planning task, and let $v \in \mathcal{V}$ be a state variable of Π.

The ***domain transition graph*** *of v, in symbols DTG(v), is the labeled directed graph with vertex set $\mathcal{D}_v$ containing the following arcs:*

- *For each axiom $cond \to v := d' \in \mathcal{A}$ such that cond contains some condition $v = d$, an arc from d to d' labeled with $cond \setminus \{v = d\}$.*
- *For each axiom $cond \to v := d' \in \mathcal{A}$ such that cond does not contain the condition $v = d$ for any $d \in \mathcal{D}_v$, an arc from each $d \in \mathcal{D}_v \setminus \{d'\}$ to d' labeled with cond.*
- *For each effect $cond \to v := d'$ of an operator o with precondition pre such that $pre \cup cond$ contains some condition $v = d$, an arc from d to d' labeled with $pre \cup cond \setminus \{v = d\}$.*
- *For each effect $cond \to v := d'$ of an operator o with precondition pre such that $pre \cup cond$ does not contain the condition $v = d$ for any $d \in \mathcal{D}_v$, an arc from each $d \in \mathcal{D}_v \setminus \{d'\}$ to d' labeled with $pre \cup cond$.*

Arcs of domain transition graphs are called ***transitions****, their labels are referred to as the* ***conditions*** *of the transition.*

Domain transition graphs can be weighted, in which case each transition has an associated non-negative integer weight. Unless stated otherwise, we assume that all transitions derived from operators have weight 1 and all transitions derived from axioms have weight 0.

The definition is somewhat lengthy, but its informal content is easy to grasp: The domain transition graph for v contains a transition from d to d' if there exists some operator or axiom that can change the value of v from d to d'. Such a transition is labeled with the conditions on *other* state variables that must be true if the transition shall be applied. Multiple transitions between the same values using different conditions are allowed and occur frequently.

We have already seen domain transition graphs in the introductory section (Figs. 3 and 4), although these were only introduced informally and did not show the arc labels usually associated with transitions. Fig. 23 shows some examples from a simple task in the GRID domain, featuring a 3×2 grid with a single initially locked location in the center of the upper row, unlockable by a single key. In the MPT encoding of the task, there are three state variables: variable r with $\mathcal{D}_r = \{ (x, y) \mid x \in \{1, 2, 3\},\ y \in \{1, 2\} \}$ encodes the location of the robot, variable k with $\mathcal{D}_k = \mathcal{D}_r \cup \{ \mathit{carried} \}$ encodes the state of the key, and variable d with $\mathcal{D}_d = \{ \mathit{closed}, \mathit{open} \}$ encodes the state of the initially locked grid location.

If all operators of an MPT are unary (i. e. only have a single effect) and we leave aside axioms for a moment, then there is a strong correspondence between the state space of an MPT and its domain transition graphs. Since vertices in domain transition graphs correspond to values of state variables, a given state is represented by selecting one vertex in each domain transition graph, called the *active vertex* of that state variable. Applying an operator means changing the active vertex of some state variable by performing a transition in the corresponding domain transition graph. Whether or not such a transition is allowed depends on its condition, which is checked against the active vertices of the other domain transition graphs. We will use the GRID example to motivate the importance of this correspondence. Consider an initial state where the robot is at location $(1, 1)$, the key is at location $(3, 2)$, and the door is locked. We represent this by placing pebbles on the appropriate

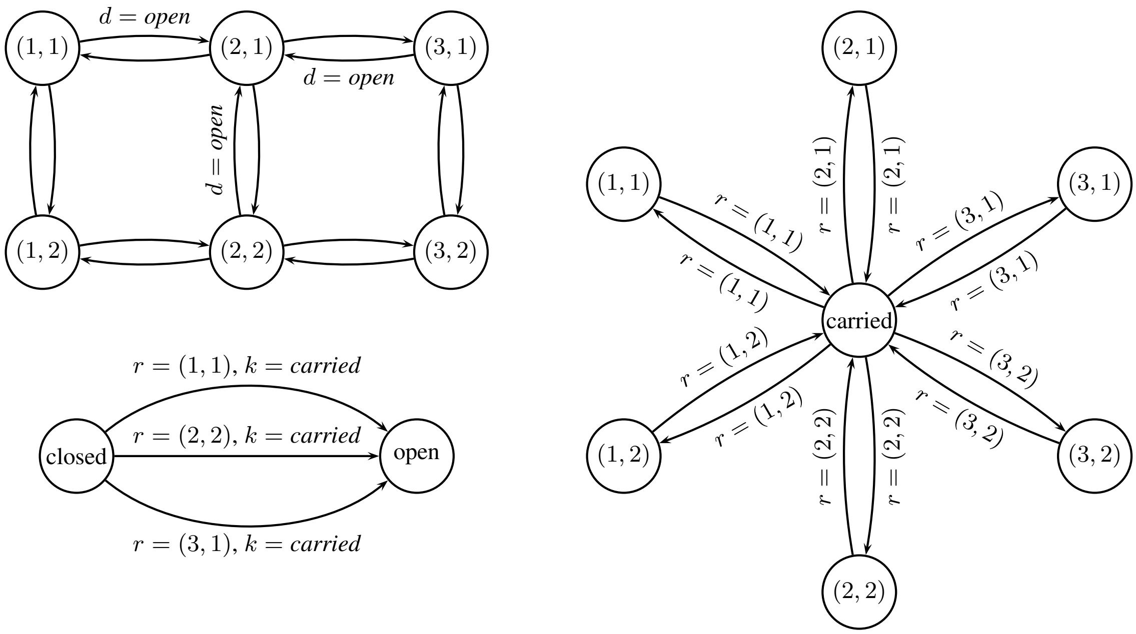

Figure 23: Domain transition graphs of a GRID task. Top left: *DTG*(r) (robot); right: *DTG*(k) (key); bottom left: *DTG*(d) (door).

vertices of the three domain transition graphs. We want to move the pebble in the domain transition graph of the key to location $(2, 1)$. This can be done by moving the robot pebble to vertex $(1, 2)$, then $(2, 2)$, then $(3, 2)$, moving the key pebble to the vertex *carried*, moving the robot pebble back to vertex $(2, 2)$, moving the door pebble to *open*, moving the robot pebble to vertex $(2, 1)$ and finally moving the key pebble to vertex $(2, 1)$.

The example shows how plan execution can be viewed as simultaneous traversal of domain transition graphs (cf. Domshlak & Dinitz, 2001). This is an important notion for Fast Downward because the CG heuristic computes its heuristic estimates by solving subproblems of the planning task by looking for paths in domain transition graphs in basically the way we just described.

As mentioned before, this view of MPT planning is only completely accurate for unary tasks without axioms, for which the domain transition graphs are indeed a complete representation of the state space. For non-unary operators, we would need to "link" certain transitions in different domain transition graphs which belong to the same operator. These could then only be executed together. For axioms, we would need to mark certain transitions as "mandatory", requiring that they be taken whenever possible. (This is only intended as a rough analogy and leaves out details like layered axioms.)

In our previous work (Helmert, 2004), we have successfully applied this view of planning to STRIPS tasks. Extending the notion to plans with conditional effects provides no challenges because domain transition graphs always consider planning operators one effect at a time, in which case effect condition can just be seen as part of the operator precondition. However, axioms provide a challenge that is easily overlooked. If we want to change the value of a fluent from d to d', the

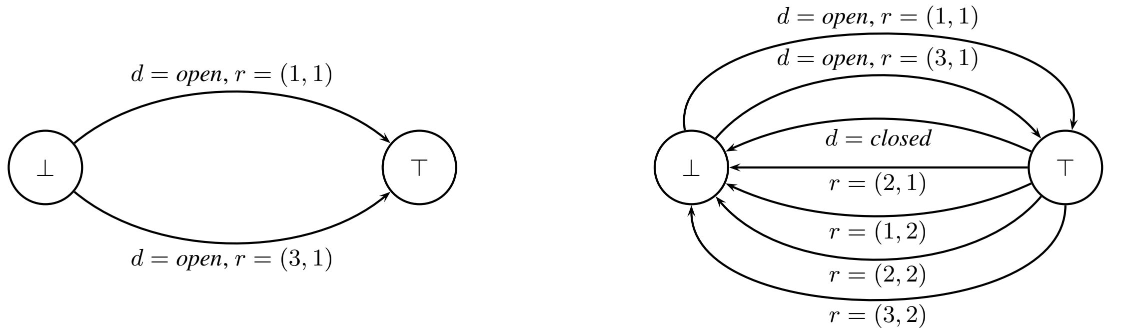

Figure 24: Domain transition graphs for the *freezing* variable in the GRID task, normal (left) and extended (right). Note that only the extended graph shows how to change state from "freezing" ($\top$) to "not freezing" ($\perp$).

domain transition graph contains all the important information; just find a path from d to d' and try to find out how the associated conditions can be achieved. Consider the same problem for a derived state variable. Let us assume that unlocking the location in the GRID example leads to a draught, causing the robot to freeze if it enters a horizontally adjacent location. We could encode this with a new derived variable f (for *freezing*) with domain $\mathcal{D}_f = \{\top, \perp\}$, defined by the axioms $d = \mathit{open}, r = (1, 1) \rightarrow f := \top$ and $d = \mathit{open}, r = (3, 1) \rightarrow f := \top$. The domain transition graph $\mathit{DTG}(f)$ is depicted in Fig. 24 (left).

The problem with that domain transition graph is that it does not tell us how we can change the state of variable f from $\top$ to $\perp$. In general, in MPTs derived from STRIPS tasks where derived predicates occur negatively in any condition, the domain transition graph does not contain sufficient information for changing the value of a derived variable from "true" to "false". Derived variables never assume the value $\perp$ due to a *derivation* of that value; because of negation by failure semantics, they only assume that value *by default* if *no other value* can be derived. If we want to reason about ways of setting the value of a derived variable to $\perp$, we will need to make that information explicit.

In logical notation, whether or not a derived variable assumes a given value by triggering an axiom at a given layer is determined by a formula in disjunctive normal form, with one conjunct for each axiom setting the value. For example, our axioms $d = \mathit{open}, r = (1, 1) \rightarrow f := \top$ and $d = \mathit{open}, r = (3, 1) \rightarrow f := \top$ correspond to the DNF formula $(d = \mathit{open}) \wedge (r = (1, 1)) \vee (d = \mathit{open}) \wedge (r = (3, 1))$. If we want to know when these rules do *not* trigger, we must negate this formula, leading to the CNF formula $(d \neq \mathit{open}) \vee (r \neq (1, 1)) \wedge (d \neq \mathit{open}) \vee (r \neq (3, 1))$. To be able to encode this information in the domain transition graph, we need to replace the inequalities with equalities and translate the formula back to DNF. Since such transformations can increase the formula size dramatically, we apply simplifications along the way, removing duplicated and dominated conjuncts. The result in this case is the DNF formula $(d = \mathit{closed}) \vee (r = (2, 1)) \vee (r = (1, 2)) \vee (r = (2, 2)) \vee (r = (3, 2))$.

A domain transition graph for a derived variable which has been enriched to contain the possible ways of causing the variable to assume the value $\perp$ is called an *extended domain transition graph*, shown for the example in Fig. 24 (right). Since computing the extended domain transition graph can be costly and is not always necessary, the knowledge compilation component scans the conditions of the planning task (axioms, operator preconditions and effect conditions, goal) for occurrences of pairings of the type $v = \bot$ for derived variables v. Extended domain transition graphs are only computed for those derived variables for which they are required.

Note that negative occurrences of derived variables can cascade: If u, v and w are derived variables with domain $\{\top, \bot\}$ and the condition $v = \bot$ is present in some operator precondition, and moreover v is defined by the axiom $u = \top, w = \top \rightarrow v := \top$, then v assumes the value $\bot$ whenever u or w do, so we would require extended domain transition graphs for u and w as well.

On the other hand, multiple layers of negation by failure can cancel each other out: If derived variable v only occurs in conditions of the form $v = \bot$ but never in positive form and is defined by the axiom $u = \bot, w = \bot \rightarrow v := \top$, then we do not necessarily require extended domain transition graphs for u and w.

In general, whether or not we need extended domain transition graphs for a derived variable is computed using the following rules:

- If v is a derived predicate for which the condition $v = d$ for $d \neq \bot$ appears in an operator precondition, effect condition or in the goal, then v *is used positively*.
- If v is a derived predicate for which the condition $v = \bot$ appears in an operator precondition, effect condition or in the goal, then v *is used negatively*.
- If v is a derived predicate for which the condition $v = d$ for $d \neq \bot$ appears in the body of an axiom whose head is used positively (negatively), then v is used positively (negatively).
- If v is a derived predicate for which the condition $v = \bot$ appears in the body of an axiom whose head is used positively (negatively), then v is used negatively (positively).

The knowledge compilation component computes extended domain transition graphs for all derived predicates which are used negatively and (standard) domain transition graphs for all other state variables. Normal domain transition graphs are computed by going through the set of axioms and the set of operator effects following Definition 13, which is reasonably straight-forward; the computation of extended domain transition graphs has been outlined above. Therefore, the algorithmic aspects of this topic should not require further discussion.

5.2 Causal graphs

Causal graphs have been introduced informally in the introduction. Here is our formal definition:

Definition 14 ***Causal graphs***
*Let Π be a multi-valued planning task with variables set $\mathcal{V}$. The **causal graph** of Π, in symbols $CG(\Pi)$, is the directed graph with vertex set $\mathcal{V}$ containing an arc (v, v') iff $v \neq v'$ and one of the following conditions is true:*

- *The domain transition graph of v' has a transition with some condition on v.*
- *The set of affected variables in the effect list of some operator includes both v and v'.*

*In the first case, we say that an arc is induced by a **transition condition**, in the second case we say that it is induced by **co-occurring effects**. Of course, these are not mutually exclusive; the same causal graph arc can be generated for both reasons.*

Informally, the causal graph contains an arc from a source variable to a target variable if changes in the value of the target variable can depend on the value of the source variable. Such arcs are included also if this dependency is of the form of an *effect* on the source variable. This agrees with the definition of *dependency graphs* by Jonsson and Bäckström (1998b), although these authors distinguish between the two different ways in which an arc in the graph can be introduced by using labeled arcs.

Whether or not co-occurring effects should induce arcs in the causal graph depends on the intended semantics: If they are not included, the set of causal graph ancestors *anc*(v) of a variable v are precisely those variables which are relevant if our goal is to change the value of v. Plans for this goal can be computed without considering any variables outside *anc*(v), by eliminating all variables outside *anc*(v) from the planning task and simplifying axioms and operators accordingly. We call this the *achievability definition* of causal graphs, because causal graphs encode what variables are important for achieving a given assignment to a state variable.

However, with the achievability definition, a planner that only considers *anc*(v) in generating an action sequence that achieves a given valuation for v may modify variables outside of *anc*(v), i. e. the generated plans have side effects which could destroy previously achieved goals or otherwise have a negative impact on overall planning. Therefore, we prefer our definition, which we call the *separability definition* of causal graphs.

5.2.1 ACYCLIC CAUSAL GRAPHS

Following the separability definition of causal graphs, solving a subproblem over variables *anc*(v) is always possible without changing any values outside of *anc*(v). This leads us to the following observation:

Observation 15 ***Acyclic causal graphs and strongly connected domain transition graphs***
Let Π be an MPT such that CG(Π) is acyclic, all domain transition graphs are strongly connected, there are no derived variables, and no trivially false conditions occur in operators or goals. Then Π has a solution.

By *trivially false* conditions, we mean conditions of the kind $\{v = d, v = d'\}$ for $d \neq d'$. Note the similarity of Observation 15 to the results of Williams and Nayak (1997) on planning in domains with unary operators, acyclic causal graphs and reversible transitions. Under the separability definition of causal graphs, acyclic causal graphs imply unariness of operators because operators with several effects introduce causal cycles. Moreover, strong connectedness of domain transition graphs is closely related to Williams' and Nayak's reversibility property.

The truth of the observation can easily be seen inductively: If the planning task has only one state variable and the domain transition graph is strongly connected, then any state (of the one variable) can be transformed into any other state by applying graph search techniques. If the planning task has several state variables and the causal graph is acyclic, we pick a sink of the causal graph, i. e. a variable v without outgoing arcs, and check if a goal is defined for that variable. If not, we remove the variable from the task, thus reducing the problem to one with fewer state variables, solved recursively. If yes, we search for a path from $s_0(v)$ to $s_\star(v)$ in the domain transition graph of v, which is guaranteed to exist because the graph is strongly connected. This yields a "high-level plan" for setting v to $s_\star(v)$ that can be fleshed out by recursively inserting the plans for setting the variables of the predecessors of v in the causal graph to the values required for the transitions that

algorithm solve-easy-MPT($\mathcal{V}$, s_0, $s_\star$, $\mathcal{O}$):
  if $s_\star = \emptyset$:
    return $\langle\rangle$.
  Let $v \in \mathcal{V}$ be a variable that does not occur in any operator precondition or effect condition.
  { Such a variable always exists if *CG*(Π) is acyclic. }
  $\mathcal{V}' := \mathcal{V} \setminus \{v\}$.
  $\mathcal{O}' := \{\ o \in \mathcal{O} \mid o$ does not affect $v\ \}$.
  plan $:= \langle\rangle$
  if $s_\star(v)$ is defined:
    Let $t_1, \dots, t_k$ be a path of transitions in *DTG*(v) leading from $s_0(v)$ to $s_\star(v)$.
    for each $t \in \{t_1, \dots, t_k\}$:
      Let *cond* and o be the condition and operator associated with t.
      Let s'_0 be the state reached after executing *plan*, restricted to $\mathcal{V}'$.
      Extend *plan* by solve-easy-MPT$(\mathcal{V}', s'_0, \mathit{cond}, \mathcal{O}')$.
      Extend *plan* by o.
  Let s'_0 be the state reached after executing *plan*, restricted to $\mathcal{V}'$.
  $s'_\star := s_\star$ restricted to $\mathcal{V}'$.
  Extend *plan* by solve-easy-MPT$(\mathcal{V}', s'_0, s'_\star, \mathcal{O}')$.
  return *plan*

Figure 25: Planning algorithm for MPTs with acyclic causal graphs and strongly connected domain transition graph.

form the high-level plan. Once the desired value of v has been set, v can be eliminated from the planning task and the remaining problem can be solved recursively.

The algorithm is shown in Fig. 25. Although it is backtrack-free, it can require exponential time to execute because the generated plans can be exponentially long. This is unavoidable; even for MPTs that satisfy the conditions of Observation 15, shortest plans can be exponentially long. An example of such a planning domain is given in the proof of Theorem 4.4 in the article by Bäckström and Nebel (1995).

5.2.2 GENERATING AND PRUNING CAUSAL GRAPHS

The knowledge compilation component begins its causal graph processing by generating the "full" causal graph according to the above definition.

One consequence of the separability definition of causal graphs is that all state variables which are not ancestors of variables mentioned in the goal are completely irrelevant. Therefore, having computed the graph, the next step is computing the causal graph ancestors of all variables in the goal. Any state variables which are not found to be goal ancestors are eliminated from the planning task and causal graph, associated operators and axioms removed.

Afterwards, we compute a *pruned causal graph*, a subgraph of the causal graph which contains all vertices but fewer arcs, so that all cycles are eliminated. The pruned causal graph is later used by the CG heuristic, described in Section 6.1, to solve simplified MPTs with a divide-and-conquer approach similar to the *solve-easy-MPT* algorithm. Divide-and-conquer approaches easily get lost in infinite loops when applied to planning tasks with cycles in the causal graph, so cycles must be broken. We try do this in such a fashion that "important" causal dependencies are retained whenever possible.

More specifically, we apply the following algorithm. First, we compute the strongly connected components of the causal graph. Cycles only occur within strongly connected components, so each component can be dealt with separately. Second, for each connected component, we compute a total order $\prec$ on the vertices, retaining only those arcs (v, v') for which $v \prec v'$. If $v \prec v'$, we say that v' has a *higher level* than v. The total order is computed in the following way:

1. We assign a weight to each arc in the causal graph. The weight of an arc is $n + m$ if it is induced by n axioms and m operators. The lower the cumulated weight of the incoming arcs of a vertex, the fewer conditions are ignored by assigning a low level to this vertex.

2. We then pick a vertex v with minimal cumulated weight of incoming arcs and select it for the lowest level, i. e. we set $v \prec v'$ for all other vertices v' in the strongly connected component.

3. Since v has been dealt with, we remove the vertex and its incident arcs from consideration for the rest of the ordering algorithm.

4. The remaining problem is solved by iteratively applying the same technique to ordering the other vertices until only a single vertex remains.

The reader will notice that the pruning choices within a strongly connected component are performed by a greedy algorithm. We could also try to find sets of arcs of minimal total weight such that eliminating these arcs results in an acyclic graph. However, this is an **NP**-equivalent problem, even in the case of unweighted graphs (Garey & Johnson, 1979, problem GT8).

Having computed the causal graph and pruned causal graph, both are written to the output file of the knowledge compilation component. Domain transition graphs (extended, where necessary) are also written to output at this stage, except for those variables which are not causal ancestors of the goal variables and are hence irrelevant. For each relevant variable v, two versions of the domain transition graph are generated. The first, *normal* version contains all transition conditions, while in the second, *pruned* version a condition on a variable v' is ignored if the arc (v', v) is *not* present in the pruned causal graph. Both normal and pruned domain transition graphs are simplified before being written to the output by removing *dominated transitions*: If t and t' are transitions between the same two values of a variable, and the condition of t is a proper subset of the condition of t', then transition t is easier to apply than t', so that we remove t'. Similarly, if there are several transitions with identical conditions, we only keep one of them.

5.2.3 CAUSAL GRAPH EXAMPLES

To give some impression of the types of causal graphs typically found in the standard benchmarks and the effects of pruning, we show some examples of increasing graph complexity. Fig. 26 depicts an example task from the LOGISTICS domain, featuring two trucks, two airplanes and two packages. As can be seen, the graph is acyclic, so it requires no pruning for the CG heuristic. Since LOGISTICS tasks also feature strongly connected domain transition graphs, they can even be solved by *solve-easy-MPT*.

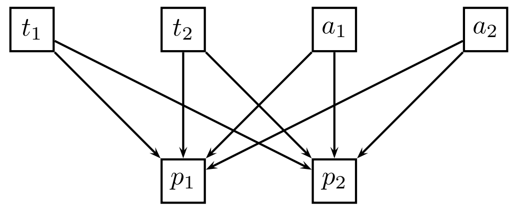

Figure 26: Causal graph of a LOGISTICS task. State variables t_i and a_i encode the locations of trucks and airplanes, state variables p_i the locations of packages.

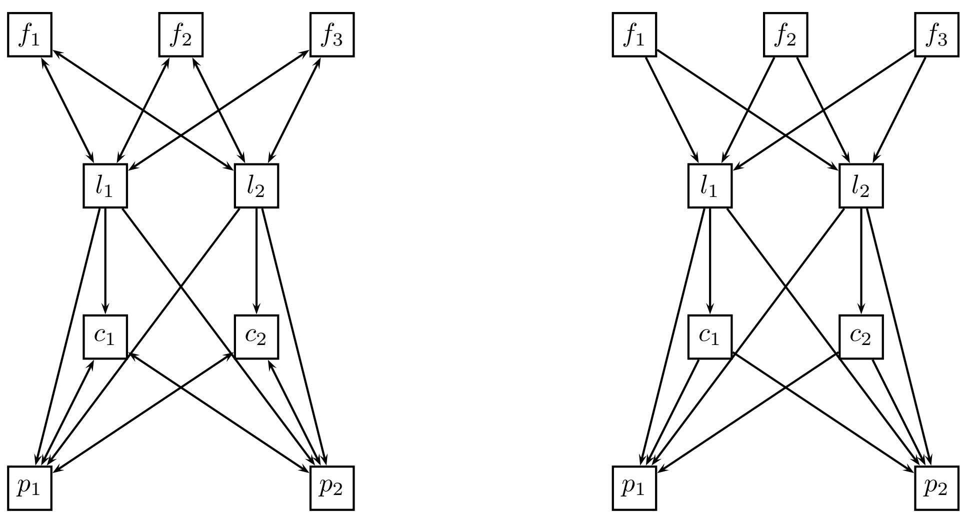

Figure 27: Causal graph of a MYSTERY task (left) and of a relaxed version of the task (right). State variables f_i encode the fuel at a location, state variables l_i and c_i encode the locations and remaining capacities of trucks, and state variables p_i encode the locations of packages.

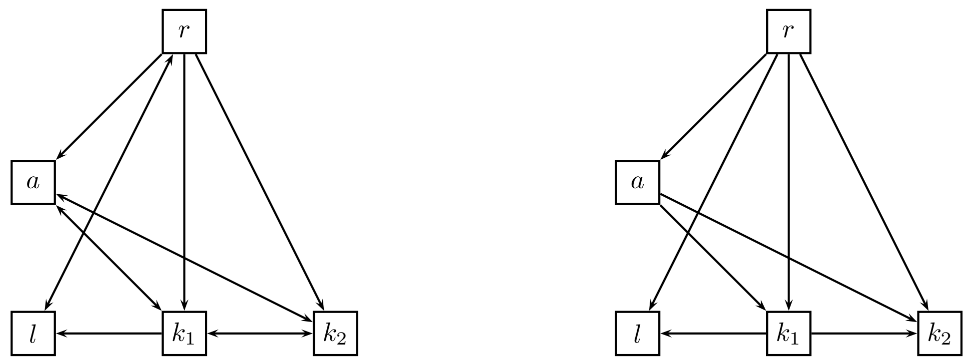

Figure 28: Causal graph of a GRID task (left) and of a relaxed version of the task (right). State variable r encodes the location of the robot, a encodes the status of the robot arm (empty or carrying a key), l encodes the status of the locked location (locked or open), and k_1 and k_2 encode the locations of the two keys.

The next figure, Fig. 27, shows an example from the MYSTERY domain with three locations, two trucks and two packages. The causal graph contains a number of cycles, but these are mostly local. By pruning arcs from vertices l_i to f_j, we ignore the fact that we must move trucks to certain locations if we want to use up fuel at that location. As using up fuel is not a very useful thing to do, this is not a big loss in information. By pruning arcs from vertices p_i to c_j, we ignore the fact that vehicles can only increase or decrease their current capacity by unloading or loading packages. Compared to heuristics based on ignoring delete effects, this is not a great loss in information, since ignoring delete effects in the MYSTERY domain almost amounts to ignoring capacity and fuel constraints altogether. By pruning these arcs, we can eliminate all cycles in the causal graph, so the MYSTERY domain can be considered fairly well-behaved.

A worse case is shown in Fig. 28, which shows an example from the GRID domain with an arbitrary number of locations, of which a single one is locked. There are two keys, one of which can unlock the locked location. Eliminating cycles here requires a few minor relaxations regarding the status of the robot arm (empty or non-empty), but also one major simplification, namely the elimination of the arc from l to r representing the fact that the robot can only enter the locked location if it has been unlocked.

As a (nearly) worst-case example, consider a task in the BLOCKSWORLD domain (no figure). A typical MPT encoding uses one state variable h for encoding whether or not the hand is empty and two state variables per block in the task: For the i-th block, t_i encodes whether or not the block is lying on the table, and b_i encodes which block is lying on top of it, or if it is clear or being held by the arm. In the causal graph of such a task, variable h has ingoing arcs from and outgoing arcs to all other state variables, and all state variables b_i are connected with each other in both directions. Only the state variables t_i have a slightly simpler connection structure, being only connected to h and t_i for the same value of i. Any relaxation of the problem that eliminates cycles from the causal graph loses a large amount of information, and it is not surprising that the DEPOT domain, which includes a BLOCKSWORLD subproblem, is the one for which the precursor of Fast Downward fared worst (Helmert, 2004). Still, it should be pointed out that planners that ignore delete effects have similar problems with BLOCKSWORLD-like domains, as the comparison of FF with the CG-based planner in the same article shows.

5.3 Successor generators and axiom evaluators

In addition to good heuristic guidance, a forward searching planning system needs efficient methods for generating successor states if it is to be applied to the benchmark suite from the international planning competitions. For some domains, our CG heuristic or other popular methods like the FF heuristic provide excellent goal estimates, yet still planning can be too time-consuming because of very long plans and vast branching factors.

The variant of best-first search implemented in Fast Downward does not compute the heuristic estimate for each state that is generated. Essentially, only closed nodes have their heuristic evaluation computed, while computation is deferred for nodes on the search frontier. For domains with strong heuristic guidance and large branching factors, the number of nodes on the frontier can by far dominate the number of nodes in the closed set. As a case in point, consider the problem instance SATELLITE #29. For solving this task, the default configuration of Fast Downward only computes heuristic estimates for 67 597 world states while adding 107 233 381 states to the frontier. Clearly, determining the set of applicable operators quickly is of critical importance in such a scenario.

Given that in some SATELLITE tasks, there are almost 1 000 000 ground operators (cf. Fig. 21), we must avoid an approach that goes through the set of ground operators one after the other to check their applicability. Similarly, in the biggest PSR tasks, more than 100 000 axioms must be evaluated for each state to compute the values of the derived variables.

Thus, to find plans in these domains, we need to make these computations efficient. For these purposes, Fast Downward uses two data structures called *successor generators* and *axiom evaluators*.

5.3.1 SUCCESSOR GENERATORS

Successor generators are recursive data structures very similar to decision trees. The internal nodes have associated conditions, which can be likened to the decisions in a decision tree, and the leaves have associated operator lists which can be likened to a set of classified samples of a decision tree leaf. They are formally defined as follows:

Definition 16 ***Successor generators***
A ***successor generator*** *for an MPT* $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$ *is a tree consisting of* ***selector nodes*** *and* ***generator nodes****.*

A selector node is an internal node of the tree. It has an associated variable $v \in \mathcal{V}$ *called the* ***selection variable****. Moreover, it has* $|\mathcal{D}_v| + 1$ *children accessed via labeled edges, one edge labeled* $v = d$ *for each value* $d \in \mathcal{D}_v$*, and one edge labeled* $\top$*. The latter edge is called the* ***don't care edge*** *of the selector.*

A generator node is a leaf node of the tree. It has an associated set of operators from $\mathcal{O}$ *called the set of* ***generated operators****.*

Each operator $o \in \mathcal{O}$ *must occur in exactly one generator node, and the set of edge labels leading from the root to that node (excluding don't care edges) must equal the precondition of* o*.*

Given a successor generator for Π and a state s of Π, we can compute the set of applicable operators in s by traversing the successor generator as follows, starting from the root:

- At a selector node with selection variable v, follow the edge $v = s(v)$ and the don't care edge.
- At a generator node, report the generated operators as applicable.

Building a suitable successor generator for a planning task is not very difficult. We apply a top-down algorithm that considers the variables of the planning task in an arbitrary order $v_1 \prec v_2 \prec \cdots \prec v_n$. At the root node, we choose v_1 as selection variable and classify the set of operators according to their preconditions with respect to v_1. Operators with a precondition $v_1 = d$ will be represented in the child of the root accessed by the edge with that label, while operators without preconditions on v_1 will be represented in the child of the root accessed by the don't care edge. In the children of the root, we choose v_2 as selection variable, in the grandchildren v_3, and so on. There is one exception to this rule to avoid creating unnecessary selection nodes: If no operator in a certain branch of the tree has a condition on v_i, then v_i is not considered as a selection variable in this branch. The construction of a branch ends when all variables have been considered, at which stage a generator node is created with the operators associated with that branch.

We believe that neither the algorithm for building a successor generator nor the algorithm for traversing it to generate the set of applicable operators requires further explanation and move on to axiom evaluators.

5.3.2 AXIOM EVALUATORS

Axiom evaluators are a simple data structure used for efficiently implementing the well-known *marking algorithm* for propositional Horn logic, extended and modified for the layered logic programs that correspond to the axioms of an MPT.

They consist of two parts: an indexing data structure which maps a given variable/value pairing and a given axiom layer to the set of axioms in that layer in whose body the pairing appears, and a set of counters, one for each axiom, counting the number of conditions of the axiom that have not yet been derived.

Within Fast Downward, axioms are evaluated in two steps. First, all derived variables are set to their default value $\perp$. Second, algorithm *evaluate-axiom-layer* (shown in Fig. 29) is executed for each axiom layer in sequence to determine the final values of the derived variables. We assume that the reader is familiar enough with the marking algorithm not to require much explanation, so we only point out that the test whether or not an axiom is ready to trigger is implemented by means of a queue in which axioms are put as soon as their counter reaches 0.

The actual implementation of *evaluate-axiom-layer* in Fast Downward is slightly more efficient than indicated by the pseudocode, since it avoids looking at each variable for each layer in the initialization step of the algorithm. However, this is a minor technical detail, so we turn to the final piece of Fast Downward's architecture, the search component.

6. Search

Unlike the translation and knowledge compilation components, for which there is only a single mode of execution, the search component of Fast Downward can perform its work in various alternative ways which are selected by command-line parameters. There are three basic search algorithms and three different heuristic estimators available:

```
algorithm evaluate-axiom-layer(s, A_i):
    for each axiom a ∈ A_i:
        a.counter := |a.cond|
    for each variable v:
        for each axiom a ∈ A_i with a condition v = s(v) in the body:
            a.counter := a.counter − 1
    while there exists an axiom a ∈ A_i with a.counter = 0 that was not yet considered:
        Let a be such an axiom.
        ⟨v, d⟩ := a.head
        if s(v) ≠ d:
            s(v) := d
            for each axiom a ∈ A_i with a condition v = d in the body:
                a.counter := a.counter − 1
```

Figure 29: Computing the values in the derived variables of a given planning state.

1. *Greedy best-first search:* This is the standard textbook algorithm (Russell & Norvig, 2003), adapted slightly to reduce the negative influence of wide branching. We have also extended the algorithm so that it can deal with *preferred operators*, similar to FF's helpful actions (Hoffmann & Nebel, 2001). We discuss greedy best-first search in Section 6.3. Fast Downward uses this algorithm together with the CG heuristic, discussed in Section 6.1.

2. *Multi-heuristic best-first search:* A variation of greedy best-first search that evaluates search states using multiple heuristic estimators, maintaining separate open lists for each. Like our implementation of greedy best-first search, it supports the use of *preferred operators*. Multi-heuristic best-first search is discussed in Section 6.3.3. Fast Downward uses this algorithm together with the CG and FF heuristics, discussed in Sections 6.1 and 6.2. A third heuristic estimator, the HSP h_{add} heuristic, is implemented but currently not used by any configuration; we do not discuss it.

3. *Focused iterative-widening search:* This is a simple search algorithm that does not use heuristic estimators, instead limiting the vast set of search possibilities by focusing on a limited operator set derived from the causal graph. It is an experimental search algorithm; in future versions of Fast Downward, we will likely replace it with a more robust search scheme based on similar ideas. Focused iterative-widening search is discussed in Section 6.4.

For the two heuristic search algorithms, the user may also choose which strategy to use for the selection of *preferred operators*. There are five options available:

1. Do not use preferred operators.

2. Use the *helpful transitions* of the CG heuristic as preferred operators.

3. Use the *helpful actions* of the FF heuristic as preferred operators.

4. Use helpful transitions as preferred operators, falling back to helpful actions if there are no helpful transitions in the current search state.

5. Use both helpful transitions and helpful actions as preferred operators.

Each of these five options can be combined with any of the two heuristic search algorithms, so that there is a total of eleven possible settings for the search component, ten using one of the heuristic algorithms and one using focused iterative-widening search.

In addition to these basic settings, the search component can be configured to execute several alternative configurations in parallel by making use of an internal scheduler. Both IPC4 configurations of Fast Downward made use of this feature by running one configuration of the heuristic search algorithms in parallel with focused iterative-widening search. As its heuristic search algorithm, the configuration "Fast Downward" employed greedy best-first search with helpful transitions, falling back to helpful actions when necessary (option 4.). The configuration "Fast Diagonally Downward" employed multi-heuristic best-first search using helpful transitions and helpful actions as preferred operators (option 5.).

6.1 The CG heuristic

The *causal graph heuristic*, or *CG heuristic* for short, is the centerpiece of Fast Downward's heuristic search engine. It estimates the cost of reaching the goal from a given search state by solving a number of subproblems of the planning task which are derived by looking at small "windows" of the (pruned) causal graph. In this section, we do not intend to provide a complete motivation why it is defined the way it is, what its strengths and weaknesses are, or how it relates to other heuristics in the literature. Neither do we discuss some of the more theoretical aspects. For addressing such questions, we refer to the article in which the heuristic was first introduced (Helmert, 2004) and to an extended version of that article which we are currently preparing. All we want to provide here is a basic understanding of the heuristic.

6.1.1 Conceptual view of the CG heuristic

For each state variable v and each pair of values $d, d' \in \mathcal{D}_v$, the CG heuristic computes a heuristic estimate $\mathit{cost}_v(d, d')$ for the cost of changing the value of v from d to d', assuming that all other state variables carry the same values as in the current state. (This is a simplification. Cost estimates are not computed for state variables v or values d for which they are never required. We ignore this fact when discussing the heuristic on the conceptual level.) The heuristic estimate of a given state s is the sum over the costs $\mathit{cost}_v(s(v), s_\star(v))$ for all variables v for which a goal condition $s_\star(v)$ is defined.

Conceptually, cost estimates are computed one variable after the other, by traversing the (pruned) causal graph in a bottom-up fashion. By bottom-up, we mean that we start with the variables that have no predecessors in the causal graphs; we call this order of computation "bottom-up" because we consider variables that can change their state of their own accord *low-level*, while variables whose state transitions require the help of other variables have more complex transition semantics and are thus considered *high-level*. Note that in our figures depicting causal graphs, *high-level* variables are typically displayed near the *bottom*.

For variables without predecessors in the causal graph, $\mathit{cost}_v(d, d')$ simply equals the cost of a shortest path from d to d' in the (pruned) domain transition graph $\mathit{DTG}(v)$. For other variables, cost estimates are also computed by graph search in the domain transition graph. However, the conditions of transitions must be taken into account during path planning, so that in addition to

algorithm compute-costs-bottom-up(Π, s):
  for each variable v of Π, traversing the pruned causal graph in bottom-up order:
    Let $\mathcal{V}'$ be the set of immediate predecessors of v in the pruned causal graph.
    for each pair of values $(d, d') \in \mathcal{D}_v \times \mathcal{D}_v$:
      Generate a planning task $\Pi_{v,d,d'}$ with the following properties:
       — *Variables:* $\mathcal{V}'$.
       — *Initial state:* $v = d$ and $v' = s(v')$ for all $v' \in \mathcal{V}'$.
       — *Goal:* $v = d'$.
       — *Axioms and operators:*
        1. Those corresponding to transitions in the pruned DTG of v.
        2. For all variables $v' \in \mathcal{V}'$ and values $e, e' \in \mathcal{D}_{v'}$, an operator
         with precondition $v' = e$, effect $v' = e'$ and cost $cost'_v(e, e')$.
         { Note that all variables $v' \in \mathcal{V}'$ have been considered previously,
          so that their cost values are known. }
      Set $cost_{v'}(d, d')$ to the cost of a plan π that solves $\Pi_{v,d,d'}$.

Figure 30: The *compute-costs-bottom-up* algorithm, a high-level description of the CG heuristic.

counting the number of transitions required to reach the destination value, we also consider the costs for achieving the value changes of the other variables necessary to set up the transition conditions.

The important point here is that in computing the values $cost_v(d, d')$, we completely consider all interactions of the state variable v with its predecessors in the causal graph. If changing the value from d to d' requires several steps and each of these steps has an associated condition on a variable v', then we realize that v' must assume the values required by those conditions *in sequence*. For example, if v represents a package in a transportation task that must be moved from A to B by means of a vehicle located at C, then we recognize that the vehicle must first move from C to A and then from A to B in order to drop the package at B. This is very different to the way HSP- or FF-based heuristics work on such examples. However, we only consider interactions with the *immediate* predecessors of v in the causal graph. Interactions that occur via several graph layers are not captured by the heuristic estimator.

In essence, we compute $cost_v(d, d')$ by solving a particular subproblem of the MPT, induced by the variable v and its predecessors in the pruned causal graph. For this subproblem, we assume that v is initially set to d, we want v to assume the value d', and all other state variables carry the same value as in the current state. We call this planning problem the *local subproblem for* v*,* d *and* d', or the *local subproblem for* v *and* d if we leave the target value d' open.

For a formalization of these intuitive notions of how the *cost* estimates are generated, consider the pseudo-code in Fig. 30. It does not reflect the way the heuristic values are actually computed within Fast Downward; the algorithm in the figure would be far too expensive to evaluate for each search state. However, it computes the same cost values as Fast Downward does, provided that the algorithm generating the plans π in the last line of the algorithm is the same one as the one used for the "real" cost estimator.

6.1.2 COMPUTATION OF THE CG HEURISTIC

The actual computation of the CG heuristic traverses the causal graph in a top-down direction, starting from the goal variables, rather than bottom-up, starting from variables without causal predecessors. In fact, this top-down traversal of the causal graph is the reason for Fast Downward's name.

Computing cost estimates in a top-down traversal implies that while the algorithm is computing plans for local subproblems of a given variable, it typically does not yet know the costs for changing the state of its causal predecessors. The algorithm *compute-costs* addresses this by evaluating the cost values of dependent variables through recursive invocations of itself.

For a given variable-value pairing $v = d$, we always compute the costs $cost_v(d, d')$ for all values of $d' \in \mathcal{D}_v$ at the same time, similar to the way Dijkstra's algorithm computes the shortest path not from a single source to a single destination vertex, but from a single source to all possible destination vertices. Computing the costs for all values of d' is not (much) more expensive than computing only one of these values, and once all cost values have been determined, we can cache them and re-use them if they are needed again later during other parts of the computation of the heuristic value for the current state.

In fact, the similarity to shortest path problems is not superficial but runs quite deeply. If we ignore the recursive calls for computing cost values of dependent variables, *compute-costs* is basically an implementation of Dijkstra's algorithm for the single-source shortest path problem on domain transition graphs. The only difference to the "regular" algorithm lies in the fact that we do not know the cost for using an arc in advance. Transitions of derived variables have a base cost of 0 and transitions of fluents have a base cost of 1, but in addition to the base cost, we must pay the cost for achieving the conditions associated with a transition. However, the cost for achieving a given condition $v' = e'$ depends on the current value e of that state variable at the time the transition is taken. Thus, we can only compute the real cost for a transition once we know the values of the dependent state variables in that situation.

Of course, there are many different ways of taking transitions through causal graphs, all potentially leading to different values for the dependent state variables. When we introduced the CG heuristic, we showed that deciding plan existence for the local subproblems is **NP**-complete (Helmert, 2004), so we are content with an approach that does not lead to a complete planning algorithm, as long as it works well for the subproblems we face in practice.

The approach we have chosen is to achieve each value of state variable v in the local subproblem for v and d as quickly as possible, following a greedy policy. In the context of the Dijkstra algorithm, this means that we start by finding the cheapest possible plan to make a transition from d to some other value d'. Once we have found the cheapest possible plan $\pi_{d'}$, we commit to it, annotating the vertex d' of the domain transition graph with the local state obtained by applying plan $\pi_{d'}$ to the current state. In the next step, we look for the cheapest possible plan to achieve another value d'', by either considering transitions that start from the initial value d, or by considering transitions that continue the plan $\pi_{d'}$ by moving to a neighbor of d'. This process is iterated until all vertices of the domain transition graph have been reached or no further progress is possible.

Our implementation follows Dijkstra's algorithm (Fig. 31). We have implemented the priority queue as a vector of buckets for maximal speed and use a cache to avoid generating the same $cost_v(d, d')$ value twice for the same state. In addition to this, we use a global cache that is shared throughout the whole planning process so that we need to compute the values $cost_v(d, d')$ for vari-

```
algorithm compute-costs(Π, s, v, d):
    Let V′ be the set of immediate predecessors of v in the pruned causal graph of Π.
    Let DTG be the pruned domain transition graph of v.
    cost_v(d, d) := 0
    cost_v(d, d′) := ∞ for all d′ ∈ D_v \ {d}
    local-state_d := s restricted to V′
    unreached := D_v
    while unreached contains a value d′ ∈ D_v with cost_v(d, d′) < ∞:
        Choose such a value d′ ∈ unreached minimizing cost_v(d, d′).
        unreached := unreached \ {d′}
        for each transition t in DTG leading from d′ to some d″ ∈ unreached:
            transition-cost := 0 if v is a derived variable; 1 if v is a fluent
            for each pair v′ = e′ in the condition of t:
                e := local-state_d′(v′)
                call compute-costs(Π, s, v′, e).
                transition-cost := transition-cost + cost_v′(e, e′)
            if cost_v(d, d′) + transition-cost < cost_v(d, d″):
                cost_v(d, d″) := cost_v(d, d′) + transition-cost
                local-state_d″ := local-state_d′
                for each pair v′ = e′ in the condition of t:
                    local-state_d″(v′) := e′
```

Figure 31: Fast Downward's implementation of the CG heuristic: the *compute-costs* algorithm for computing the estimates $cost_v(d, d')$ for all values $d' \in \mathcal{D}_v$ in a state s of an MPT Π.

ables v with few ancestors in the pruned causal graph only once. (Note that $cost_v(d, d')$ only depends on the current values of the ancestors of v.)

Apart from these technical considerations, Fig. 31 gives an accurate account of Fast Downward's implementation of the CG heuristic. For more details, including complexity considerations and a worked-out example, we refer to the original description of the algorithm (Helmert, 2004).

6.1.3 STATES WITH INFINITE HEURISTIC VALUE

We noted that Fast Downward uses an incomplete planning algorithm for determining solutions to local planning problems. Therefore, there can be states s such that $cost_v(s(v), s_\star(v)) = \infty$ even though the goal condition $v = s_\star(v)$ can still be reached. This means that we cannot trust infinite values returned by the CG heuristic. In our experience, states with infinite heuristic evaluation from which it is still possible to reach the goal are rare, so we indeed treat such states as *dead-ends*.

If it turns out that *all* states at the search frontier are dead-ends, we cannot make further progress with the CG heuristic. In this case, we use a sound dead-end detection routine to verify the heuristic assessment. If it turns out that all frontier states are indeed dead-ends, then we report the problem as unsolvable. Otherwise, the current implementation of Fast Downward gives up, reporting that it cannot solve the task. Of course, it would be possible to switch to an alternative search strategy in such a case, e. g. using the FF heuristic, but since we have not observed such failure in practice, we have not implemented such emergency measures.

The dead-end detection routine has been originally developed for STRIPS-like tasks. However, extending it to full MPTs is easy; in fact, no changes to the core algorithm are required, as it works at the level of domain transition graphs and is still sound when applied to tasks with conditional effects and axioms. Since it is not a central aspect of Fast Downward, we do not discuss it here, referring to our earlier work instead (Helmert, 2004).

6.1.4 HELPFUL TRANSITIONS

Inspired by Hoffmann's very successful use of *helpful actions* within the FF planner (Hoffmann & Nebel, 2001), we have extended our algorithm for computing the CG heuristic so that in addition to the heuristic estimate, it also generates a set of applicable operators considered useful for steering search towards the goal.

To compute helpful actions in FF, Hoffmann's algorithm generates a plan for the relaxed planning task defined by the current search state and considers those operators *helpful* which belong to the relaxed plan and are applicable in the current state.

Our approach follows a similar idea. After computing the heuristic estimate $cost_v(s(v), s_\star(v))$ for a variable v for which a goal condition is defined, we look into the domain transition graph of v to trace the path of transitions leading from $s(v)$ to $s_\star(v)$ that gave rise to the cost estimate. In particular, we consider the first transition on this path, starting at $s(v)$. If this transition corresponds to an applicable operator, we consider that operator a *helpful transition* and continue to check the next goal. If the transition does not correspond to an applicable operator because it has associated conditions of the form $v' = e'$ which are not currently satisfied, then we recursively look for helpful transitions in the domain transition graph of each such variable v', checking the path that was generated during the computation of $cost_{v'}(s(v'), e')$.

The recursive process continues until we have found all helpful transitions. Unlike the case for FF, where helpful actions can be found for all non-goal states, we might not find any helpful transition at all. It may be the case that a transition does not correspond to an applicable operator even though it has no associated conditions; this can happen when there are some operator preconditions not represented in the domain transition graph due to cycles in the causal graph. Even so, we have found helpful transitions to be a useful tool in guiding our best-first search algorithms.

6.2 The FF heuristic

The *FF heuristic* is named after Hoffmann's planning algorithm of the same name, in the context of which it was originally introduced (Hoffmann & Nebel, 2001). It is based on the notion of *relaxed planning tasks* that ignore *negative interactions*. In the context of MPTs, ignoring negative interactions means that we assume that each state variable can hold several values simultaneously. An operator effect or axiom that sets a variable v to a value d in the original task corresponds to an effect or axiom that *adds* the value d to the range of values assumed by v in the relaxed task. A condition $v = d$ in the original task corresponds to a condition requiring d to be an element of the set of values currently assumed by v in the relaxed task.

It is easy to see that applying some operator can never render a solvable relaxed planning task unsolvable. It can only lead to more operators being applicable and more goals being true, if it has any significant effect at all. For this reason, relaxed planning tasks can be solved efficiently, even though optimal solutions are still **NP**-hard to compute (Bylander, 1994). A plan for the relaxation of some planning task is called a *relaxed plan* for that task.

The FF heuristic estimates the goal distance of a world state by generating a relaxed plan for the task of reaching the goal from that world state. The number of operators in that plan is then used as the heuristic estimate. Our implementation of the FF heuristic does not necessarily generate the same, or even an equally long, relaxed plan as FF.In our experiments, this did not turn out to be problematic, as both implementations appear to be equally informative.

While the FF heuristic was originally introduced for ADL domains, extending it to tasks involving derived predicates is straight-forward. One possible extension is to simply assume that each derived predicate is initially set to its default value $\perp$ and treat axioms as relaxed operators of cost 0. In a a slightly more complicated, but also more accurate approach, derived variables are initialized to their actual value in a given world state, allowing the relaxed planner to achieve the value $\perp$ (or other values) by applying the transitions of the extended domain transition graph of the derived variable. We have followed the second approach.

In addition to heuristic estimates, the FF heuristic can also be exploited for restricting or biasing the choice of operators to apply in a given world state s. The set of *helpful actions* of s consists of all those operators of the relaxed plan computed for s that are applicable in that state. As mentioned in the introduction to this section, Fast Downward can be configured to treat helpful actions as preferred operators.

There is a wealth of work on the FF heuristic in the literature, so we do not discuss it further. For a more thorough treatment, we refer to the work by Hoffmann (2001, 2002).

6.3 Best-first search in Fast Downward

Fast Downward uses *greedy best-first search with a closed list* as its default search algorithm. We assume that the reader is familiar with the algorithm and refer to the literature (Russell & Norvig, 2003) for details. Our implementation of greedy best-first search differs from the textbook algorithm in two ways, described in the following.

6.3.1 PREFERRED OPERATORS

To make use of helpful transitions computed by the CG heuristic or helpful actions computed by the FF heuristic, our implementation of greedy best-first search supports the use of so-called *preferred operators*.

The set of preferred operators of a given state is a subset of the set of applicable operators for that state. Which operators are considered preferred depends on the settings for the search component, as discussed earlier. The intuition behind preferred operators is that a randomly picked successor state is more likely to be closer to the goal if it is generated by a preferred operator, in which case we call it a *preferred successor*. Preferred successors should be considered before non-preferred ones on average.

Our search algorithm implements this preference by maintaining two separate open lists, one containing all successors of a state and one containing preferred successors exclusively. The search algorithm alternates between expanding a regular successor and a preferred successor. On even iterations it will consider the one open list, on odd iterations the other. No matter which open list a state is taken from, all its successors are placed in the first open list, and the preferred successors are additionally placed in the second open list. (Of course we could limit the first open list to only contain non-preferred successors; however, typically the total number of successors is vast and the number of preferred successors is tiny. Therefore, it is cheaper to add all successors to the first open list and detect duplicates upon expansion than scan through the list of successors determining for each element whether or not it is preferred.)

Since the number of preferred successors is smaller than the total number of successors, this means that preferred successors are typically expanded much earlier than others. This is especially important in domains where heuristic guidance is weak and a lot of time is spent exploring plateaus. When faced with plateaus, Fast Downward's open lists operate in a first-in-first-out fashion. (In other words: For a constant heuristic function, our search algorithm behaves like breadth-first search.) Preferred operators typically offer much better chances of escaping from plateaus since they lead to significantly lower effective branching factors.

6.3.2 DEFERRED HEURISTIC EVALUATION

Upon expanding a state s, the textbook version of greedy best-first search computes the heuristic evaluation of all successor states of s and sorts them into the open list accordingly. This can be wasteful if s has many successors and heuristic evaluations are costly, two conditions that are often true for heuristic search approaches to planning. This is where our second modification comes into play.

If a successor with a better heuristic estimate than s is generated early and leads to a promising path towards the goal, we would like to avoid generating the other successors. Let us assume that s has 1000 successors, and that s', the 10th successor of s being generated, has a better heuristic estimate than s. Furthermore, let us assume that the goal can be reached from s' on a path with non-increasing heuristic estimates. Then we would like to avoid computing heuristic values for the 990 later successors of s altogether, focusing on s' instead.

Deferred heuristic evaluation achieves this by *not* computing heuristic estimates for the successors of an expanded state s immediately. Instead, the successors of s are placed in the open list together with the heuristic estimate *of state* s, and their own heuristic estimates are only computed when and if they are expanded, at which time it is used for sorting *their* successors into the open list, and so on. In general, each state is sorted into the open list according to the heuristic evaluation of its parent, with the initial state being an exception. In fact, we do not need to put the successor state itself into the open list, since we do not require its representation before we want to evaluate its heuristic estimate. Instead, we save memory by storing only a reference to the parent state and the operator transforming the parent state into the successor state in the open list.

It might not be clear how this approach can lead to significant savings in time, since deferred evaluation also means that information is only available later. However, the savings become apparent when considering deferred evaluation together with preferred operators: If an improving successor s' of a state s is reached by a preferred operator, it is likely that it will be expanded (via the second open list) long before all other successors — or even most siblings — of s. In the situation described above, where there exists a non-increasing path from s' to the goal, heuristic evaluations will never be computed for most successors of s.

6.3.3 MULTI-HEURISTIC BEST-FIRST SEARCH

As an alternative to greedy best-first search, we have also implemented an extended algorithm called *multi-heuristic best-first search*. This algorithm differs from greedy best-first search in its use of multiple heuristic estimators; it forms the foundation of the version of the planner that was entered as *Fast Diagonally Downward* at IPC4.

The idea behind using multiple heuristic estimators is that different heuristic estimators have different weaknesses. It may be the case that a given heuristic is sufficient for directing the search towards the goal except for one part of the plan, where it gets stuck on a plateau. Another heuristic might have similar characteristics, but get stuck in another part of the search space.

Various ways of combining heuristics have been proposed in the literature, most obviously adding together or taking the maximum of the heuristic estimates. We believe that it is often beneficial *not* to combine the different heuristic estimates into a single numerical value. Instead, we propose maintaining a separate open list for each heuristic estimator, which is sorted according to the respective heuristic. The search algorithm alternates between expanding a state from each open list. Whenever a state is expanded, estimates are calculated according to *each* heuristic, and the successors are put into each open list.

When Fast Downward is configured to use multi-heuristic best-first search, it computes estimates both for the CG heuristic and FF heuristic, maintaining two open lists. Of course, the approach can be combined with the use of preferred operators; in this case, the search algorithm maintains four open lists, as each heuristic distinguishes between normal and preferred successors.

6.4 Focused iterative-widening search

The *focused iterative-widening search* algorithm is the most experimental piece of Fast Downward's search arsenal. In its present form, the algorithm is unsuitable for many planning domains, especially those containing comparatively few different goals. Yet we think that it might contain the nucleus for a successful approach to domain-independent planning which is very different to most current methods, so we include it for completeness and as a source of inspiration.

The algorithm is intended as a first step towards developing search techniques that emphasize the idea of using heuristic criteria locally, for limiting the set of operators to apply, rather than globally, for choosing which states to expand from a global set of open states. We made first experiments in this direction after observing the large boost in performance that can be obtained by using preferred

algorithm reach-one-goal(Π, v, d, *cond*):
  for each $\vartheta \in \{0, 1, \dots, \textit{max-threshold}\}$:
    Let $\mathcal{O}_\theta$ be the set of operators of Π whose modification distance wrt v is at most ϑ.
    Assign the cost c to each operator $o \in \mathcal{O}_\theta$ with modification distance c wrt v.
    Call the *uniform-cost-search* algorithm with a closed list, using the operator set O_θ,
      to find a state satisfying $\{v = d\} \cup \textit{cond}$.
    Continue if uniform-cost search fails.

Figure 32: The *reach-one-goal* procedure for reaching a state with $v = d$. The value *max-threshold* is equal to the maximal modification distance of any operator with respect to v.

operators in heuristic search. The algorithm performed surprisingly well in some of the standard domains, while performing badly in most others.

As the name suggests, the algorithm *focuses* the search by concentrating on one goal at a time, and by restricting its attention to operators which are supposedly important for reaching that goal:

Definition 17 ***Modification distances***
Let Π *be an MPT, let* o *be an operator of* Π*, and let* v *be a variable of* Π.

The ***modification distance*** *of* o *with respect to* v *is defined as the minimum, over all variables* v' *that occur as affected variables in the effect list of* o*, of the distance from* v' *to* v *in* $CG(\Pi)$.

For example, operators that modify v directly have a modification distance of 0 with respect to v, operators that modify variables which occur in preconditions of operators modifying v have a modification distance of 1, and so on. We assume that in order to change the value of a variable, operators with a low modification distance with respect to that variable are most useful.

Fig. 32 shows the *reach-one-goal* procedure for achieving a single goal of an MPT. For the time being, assume that the *cond* parameter is always $\emptyset$. The procedure makes use of the assumption that high modification distance implies low usefulness in two ways. First, operators with high modification distance with respect to the goal variable are considered to have a higher associated cost, and are hence applied less frequently. Second, operators whose modification distance is beyond a certain threshold are forbidden completely. Instead of choosing a threshold a priori, the algorithm first tries to find a solution with the lowest possible threshold of 0, increasing the threshold by 1 whenever the previous search has failed. The *uniform-cost-search* algorithm mentioned in Fig. 32 is the standard textbook method (Russell & Norvig, 2003).

The focused iterative-widening search algorithm is based on the *reach-one-goal* method; the idea is to achieve the goals of the planning task one after the other, by using the *reach-one-goal* algorithm as the core subroutine for reaching individual goals. Since it is not obvious what a good order of achieving the goals would be, one invocation of *reach-one-goal* is started for each goal in parallel. Since each one-goal solver focuses on the (supposedly) relevant operators for reaching that particular goal, there is hope that the number of states considered before a goal is reached is small. Once one of the one-goal solvers reaches its goal, the resulting plan is reported and all sub-searches are stopped. The overall search algorithm commits to that part of the plan and considers the situation in which the first goal has been reached its new initial state.

algorithm reach-one-goal(Π, v, d, *cond*):
  for each $\vartheta \in \{0, 1, \dots, \textit{max-threshold}\}$:
    Let $\mathcal{O}_\theta$ be the set of operators of Π whose modification distance wrt v is at most ϑ and which do not change the value of any state variable occurring in *cond*.
    Assign the cost c to each operator $o \in \mathcal{O}_\theta$ with modification distance c wrt v.
    Call the *uniform-cost-search* algorithm with a closed list, using the operator set O_θ, to find a state satisfying $\{v = d\} \cup \textit{cond}$.
    Continue if uniform-cost search fails.
  for each $\vartheta \in \{0, 1, \dots, \textit{max-threshold}\}$:
    Let $\mathcal{O}_\theta$ be the set of operators of Π whose modification distance wrt v is at most ϑ.
    Assign the cost c to each operator $o \in \mathcal{O}_\theta$ with modification distance c wrt v.
    Call the *uniform-cost-search* algorithm with a closed list, using the operator set O_θ, to find a state satisfying $\{v = d\} \cup \textit{cond}$.
    Continue if uniform-cost search fails.

Figure 33: The *reach-one-goal* procedure for reaching a state with $v = d$ (corrected).

Starting from this situation, we try to satisfy the second goal, by again starting parallel invocations of *reach-one-goal* for each possible second goal. Of course, this can lead to a situation where the search algorithm oscillates between goals, first achieving goal a, then abandoning it in favor of goal b, without any sign of making real progress. Therefore, we demand that *reach-one-goal* achieves the second goal *in addition* to the one we reached first, by setting the *cond* argument accordingly. Once two goals have been reached, the sub-searches are again stopped, sub-searches for the third goal are started, and so on, until all goals have been reached.

There is one final twist that we have not yet mentioned and that is not shown in Fig. 32: To motivate *reach-one-goal* not to needlessly wander away from already reached goals, we try to forbid the algorithm to apply any operator that undoes any of the previously achieved goals in *cond*. Only if this restriction leads to failure do we allow the procedure to temporarily undo the previously achieved goals. The complete procedure is shown in Fig. 33.

7. Experiments

To evaluate the performance of Fast Downward, and specifically the differences between the various configurations of the search component, we have performed a number of experiments on the set of benchmarks from the previous international planning competitions.

7.1 Benchmark set

The benchmark set we used consisted of all propositional planning tasks from the fully automated tracks of the first four international planning competitions hosted at AIPS 1998, AIPS 2000, AIPS 2002 and ICAPS 2004. The set of benchmark domains is shown in Fig. 34. Altogether, the benchmark suite comprises 1442 tasks. (The numbers in Fig. 34 add up to 1462, but the 20 SATELLITE instances that were introduced for IPC3 were also part of the benchmark set of IPC4, so we only count them once.)

We distinguish between three classes of domains:

Competition	Domain	Class	Number of tasks
IPC 1 (AIPS 1998)	ASSEMBLY	ADL	30
	GRID	STRIPS	5
	GRIPPER	STRIPS	20
	LOGISTICS	STRIPS	35
	MOVIE	STRIPS	30
	MYSTERY	STRIPS	30
	MPRIME	STRIPS	35
IPC 2 (AIPS 2000)	BLOCKSWORLD	STRIPS	35
	FREECELL	STRIPS	60
	LOGISTICS	STRIPS	28
	MICONIC-STRIPS	STRIPS	150
	MICONIC-SIMPLEADL	ADL	150
	MICONIC-FULLADL	ADL	150
	SCHEDULE	ADL	150
IPC 3 (AIPS 2002)	DEPOT	STRIPS	22
	DRIVERLOG	STRIPS	20
	FREECELL	STRIPS	20
	ROVERS	STRIPS	20
	SATELLITE	STRIPS	20
	ZENOTRAVEL	STRIPS	20
IPC 4 (ICAPS 2004)	AIRPORT	STRIPS	50
	PROMELA-OPTICALTELEGRAPH	PDDL2.2	48
	PROMELA-PHILOSOPHERS	PDDL2.2	48
	PIPESWORLD-NOTANKAGE	STRIPS	50
	PIPESWORLD-TANKAGE	STRIPS	50
	PSR-SMALL	STRIPS	50
	PSR-MIDDLE	PDDL2.2	50
	PSR-LARGE	PDDL2.2	50
	SATELLITE	STRIPS	36

Figure 34: Planning domains of the first four international planning competitions.

- *STRIPS domains:* These domains do not feature derived predicates or conditional effects, and all conditions appearing in goal and operators are conjunctions of positive literals.
- *ADL domains:* These domains make use of conditional effects and/or more general conditions in goals, operators and effect conditions, but do not require derived predicates.
- *PDDL2.2 domains:* These domains use the full range of PDDL2.2 features, including those features present in ADL domains and derived predicates.

At IPC4, some domains were presented in different *formulations*, meaning that the same real-world task was encoded in several different ways. Participants were asked to only work on one formulation per domain, being able to choose their preferred formulation for a given domain freely. For example, the AIRPORT domain was available in a STRIPS formulation and an ADL formulation.

However, the organizers did not strictly follow the rule of considering different encodings of the same real-world task different *formulations*, rather than different domains proper. Namely, for the PSR-MIDDLE and PROMELA domains, encodings with and without axioms were available, and these were considered as different domains on the grounds that the encodings without axioms were much larger and hence likely more difficult to solve. We apply the formulation vs. encoding view more strictly and thus only consider one PSR-MIDDLE domain and one domain for each of the two PROMELA variants, PROMELA-PHILOSOPHERS and PROMELA-OPTICALTELEGRAPH.

Our preferred choice of encoding for the IPC4 domains is reflected in Fig. 34 in so far as the *class* column shows the class of domain of the formulation we chose for Fast Downward. In addition to what is mentioned in the figure, the AIRPORT domain is also available with an ADL encoding, both PROMELA domains are also available with STRIPS and ADL encodings, and PSR-MIDDLE is also available with an ADL encoding.

Of the IPC1 benchmark set, all tasks are solvable except for 11 MYSTERY instances. Of the IPC2 benchmark set, all tasks are solvable except in the MICONIC-FULLADL domain, which contains at least 7 unsolvable instances. All IPC3 benchmarks are solvable. For IPC4, we have not checked all instances of the PIPESWORLD-TANKAGE and PSR-LARGE domains, but we assume that all are solvable.

If run in any of the heuristic search modes, Fast Downward proves the unsolvability of the unsolvable MYSTERY and MICONIC-FULLADL tasks by using the dead-end detection routine described in our earlier article on the CG heuristic (Helmert, 2004), or in some cases in the MICONIC-FULLADL domain by exhaustively searching all states with a finite FF heuristic, if run in multi-heuristic best-first search mode. Of course, if an unsolvable task is proved unsolvable by the planner, we report this as a "successfully solved" instance in the experimental results.

7.2 Experimental setup

As discussed in Section 6, there are eleven possible configurations of Fast Downward's search component. However, not all of them seem to be equally reasonable. For example, if we use FF's helpful actions, it would seem wasteful not to use the FF heuristic estimate, since these two are calculated together. Therefore, for the greedy best-first search setup, we exclude configurations where helpful actions are always computed. For the multi-heuristic best-first search setup, we exclude configurations where only one type of preferred operators is considered, but not the other, since this would seem to be a very arbitrary choice. This leaves us with six different configurations of the planner:

1. **G**: Use greedy best-first search without preferred operators.
2. **G + P**: Use greedy best-first search with helpful transitions as preferred operators.
3. **G + P$^+$**: Use greedy best-first search with helpful transitions as preferred operators. Use helpful actions as preferred operators in states with no helpful transitions.
4. **M**: Use multi-heuristic best-first-search without preferred operators.
5. **M + P**: Use multi-heuristic best-first search with helpful transitions and helpful actions as preferred operators.
6. **F**: Use focused iterative-widening search.

We ran each benchmark through the translation and knowledge compilation modules. For those domains where it was possible (STRIPS domains not involving constants or types), we used the old C++ translator for speed reasons. Having thus prepared the input for the search component, we evaluated each of the six configurations above on each of the 1442 benchmark tasks. We used a computer with a 3.066 GHz Intel Xeon CPU — the same machine that was used for IPC4 — and set a memory limit of 1 GB and a timeout of 300 seconds.

Our experiments are mainly intended for comparing the relative performance of the different configurations of Fast Downward on different domains. However, we also provide results for other planning systems as reference points. Because of the large amount of data, we do not present run-time results for individual planning tasks. Instead, we report the following information:

- Tables showing the number of tasks *not solved* by each planner within the 300 second timeout. Here, we present individual results for each domain.
- Graphs showing the number of tasks solved in a given time by each planner. Here, we do not present separate results for each domain, because this would require too many graphs. Instead, we partition the benchmark domains into three groups and generate graphs for each group.

We do not discuss plan lengths; our observations in that regard are similar to those made for our first implementation of the CG heuristic (Helmert, 2004). Detailed experimental results, including individual runtime and plan length results for each benchmark task, will be published on the Fast Downward web site once it is available.

7.3 Translation and knowledge compilation vs. search

In the following presentation of results, we only consider a task solved if the *total* processing time, including translation, knowledge compilation and search, is below 300 seconds. However, from our experimental data, we can also gather which tasks can be solved with a timeout of 300 seconds for the *search* component alone, allowing the other components to use an arbitrary amount of resources.

It turns out that this only makes a difference in five cases, most of which could have been solved in a total time below 310 seconds (Fig. 35). Only in one of those five cases, a SATELLITE instance of exorbitant size, did search take less time than the other two components added together. These results show that the search component is the only time-critical part of Fast Downward in practice. Therefore, we do not report performance results for the individual components beyond what we stated in Section 4.6, and instead focus on performance of the planner as a whole.

Domain	Task	Configuration	Preprocessing	Search
FREECELL(IPC2)	`probfreecell-10-1`	**M + P**	9.30 s	298.64 s
GRID	`prob05`	**M**	10.04 s	291.01 s
MPRIME	`prob14`	**M**	22.38 s	291.67 s
PSR-LARGE	`p30-s179-n30-l3-f30`	**G + P**	43.43 s	265.29 s
SATELLITE(IPC4)	`p33-HC-pfile13`	**M + P**	180.74 s	169.09 s

Figure 35: Tasks which could be solved by some configuration of Fast Downward with a search timeout of 300 seconds, but not with a total processing timeout of 300 seconds. The column “preprocessing” shows the total time for translation and knowledge compilation.

7.4 STRIPS domains from IPC1-3

First, we present results for the STRIPS domains of the first three planning competitions. We consider the IPC1-3 domains separately from the IPC4 domains because for the latter we can compare to the planners that participated in IPC4. For the former, we cannot: Many of the IPC4 planning systems are not yet available for public download, so we cannot generate experimental data for these. Considering STRIPS domains separately from ADL domains makes sense because it allows us to compare our results to those obtained for our first implementation of the CG heuristic (Helmert, 2004), which did not support ADL.

Domain	#Tasks	G	G+P	G+P$^+$	M	M+P	F	Any	CG	FF	LPG
BLOCKSWORLD	35	0	0	0	0	0	17	0	0	4	0
DEPOT	22	12	13	13	12	8	11	7	14	3	0
DRIVERLOG	20	2	0	0	1	0	1	0	3	5	0
FREECELL (IPC2)	60	4	4	12	11	12	40	3	2	3	55
FREECELL (IPC3)	20	0	0	5	1	2	14	0	0	2	19
GRID	5	1	2	1	1	0	4	0	1	0	1
GRIPPER	20	0	0	0	0	0	0	0	0	0	0
LOGISTICS (IPC1)	35	1	0	0	4	0	26	0	0	0	4
LOGISTICS (IPC2)	28	0	0	0	0	0	0	0	0	0	0
MICONIC-STRIPS	150	0	0	0	0	0	0	0	0	0	0
MOVIE	30	0	0	0	0	0	0	0	0	0	0
MYSTERY	30	1	2	1	0	0	13	0	1	12	15
MPRIME	35	0	0	0	2	0	14	0	1	3	7
ROVERS	20	2	0	0	0	0	2	0	3	0	0
SATELLITE (IPC3)	20	1	0	0	0	0	6	0	0	0	0
ZENOTRAVEL	20	0	0	0	0	0	0	0	0	0	0
Total	550	24	21	32	32	22	148	10	25	32	101

Figure 36: Number of unsolved tasks for the STRIPS domains from IPC1, IPC2 and IPC3.

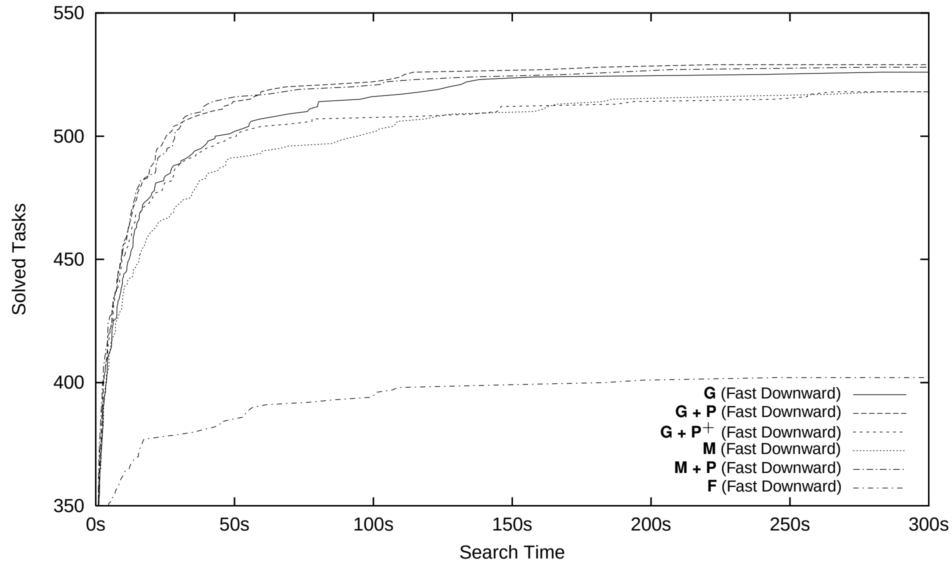

Figure 37: Number of tasks solved vs. runtime for the STRIPS domains from IPC1, IPC2 and IPC3. This graph shows the results for the various configurations of Fast Downward.

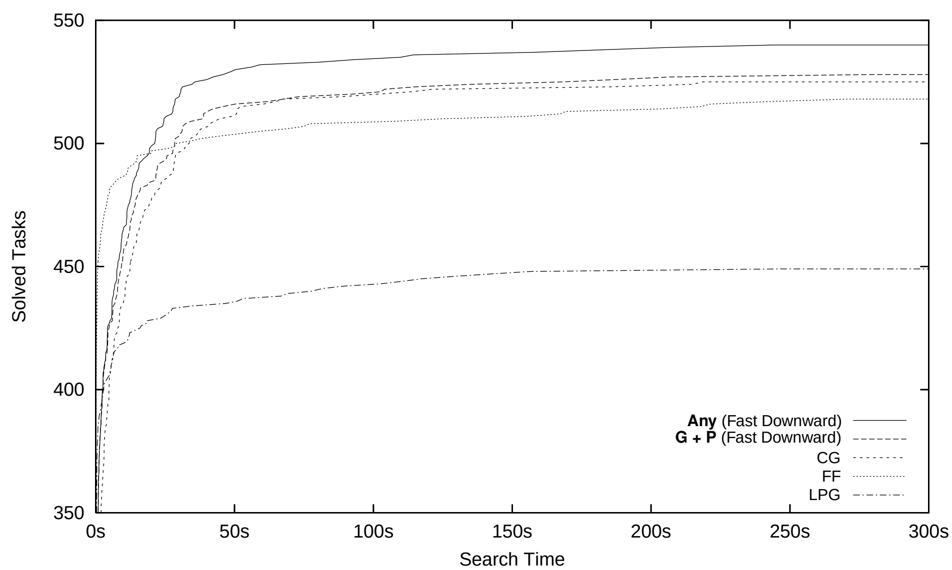

Figure 38: Number of tasks solved vs. runtime for the STRIPS domains from IPC1, IPC2 and IPC3. This graph shows the results for CG, FF and LPG and the hypothetical "Any" planner that always chooses the best configuration of Fast Downward. The result for greedy best-first search with helpful transitions is repeated for ease of comparison with Fig. 37.

Fig. 36 shows the number of unsolved tasks for each of these domains. Figs. 37 and 38 show the number of tasks solved by each planner within a given time bound between 0 and 300 seconds. Apart from the six configurations of Fast Downward we are interested in, we include four other planning systems.

Under the heading "Any", we include results for a hypothetical meta-planner that guesses the best configuration of Fast Downward for each input task and then executes Fast Downward with that setting. Under the heading "CG", we report the results for our first implementation of the CG heuristic. This should give the reader an impression how much the various modifications to the basic best-first search procedure affect the performance of the overall system. Finally, "FF" and "LPG" refer to the well-known planners (Hoffmann & Nebel, 2001; Gerevini et al., 2003) which won the fully-automated tracks of IPC2 and IPC3. They were chosen for comparison on this benchmark set because they showed the best performance by far of all publicly available planners we experimented with. For LPG, which uses a randomized search strategy, we attempted each task five times and report the median result.

The results show excellent performance of Fast Downward on this set of benchmarks. Compared to CG, which was already shown to solve more tasks than FF and LPG on this benchmark set (Helmert, 2004), we get another slight improvement for half of the configurations of Fast Downward. One of the configurations, multi-heuristic best-first search using preferred operators, solves all benchmarks in all domains except DEPOT and FREECELL. Even more important, the number of tasks solved by none of the Fast Downward configurations is as small as 10. Note that the planning competitions typically allowed a planner to spend 30 minutes on each task; under these time constraints, we could allocate five minutes to each of the six configurations of Fast Downward, getting results which are at least as good as those reported for the "Any" planner. Results might even be better under a cleverer allocation scheme.

Even the configuration using focused iterative-widening search performs comparatively well on these benchmarks, although it cannot compete with the other planners. Not surprisingly, this version of the planner had most problems in domains with many dead-ends (FREECELL, MYSTERY, MPRIME) or where goal ordering is very important (BLOCKSWORLD, DEPOT). It also fared comparatively badly in domains with very large instances (LOGISTICS (IPC1), SATELLITE). The reader should keep in mind that FF and LPG are excellent planning systems; of all the other planners we experimented with, including all those that were awarded prizes at the first three planning competitions, none solved more benchmarks from this group than focused iterative-widening search.

7.5 ADL domains from IPC1-3

Second, we present results for the ADL domains of the first three planning competitions. This is a much smaller group than the previous, including only four domains. This time, we cannot compare against CG or LPG, since neither CG nor the publicly available version of LPG supports ADL domains. Therefore, we compare to FF exclusively. Again, we report the number of unsolved tasks in each domain (Fig. 39) and present graphs showing how quickly the tasks are being solved (Figs. 40 and 41).

These results do not look as good as for the first group of domains. Results in both MICONIC domains are good, even improving on those of FF. However, greedy best-first search performs very badly on the ASSEMBLY domain, and all configurations perform badly in the SCHEDULE domain. Currently, we have no good explanation why the behavior for the SCHEDULE tasks is so bad —

Domain	#Tasks	G	G+P	G+P$^+$	M	M+P	F	Any	FF
ASSEMBLY	30	28	27	25	3	0	30	0	0
MICONIC-SIMPLEADL	150	0	0	0	0	0	0	0	0
MICONIC-FULLADL	150	8	7	8	8	7	90	5	12
SCHEDULE	150	134	93	93	132	28	113	25	0
Total	480	170	127	126	143	35	233	30	12

Figure 39: Number of unsolved tasks for the ADL domains from IPC1, IPC2 and IPC3.

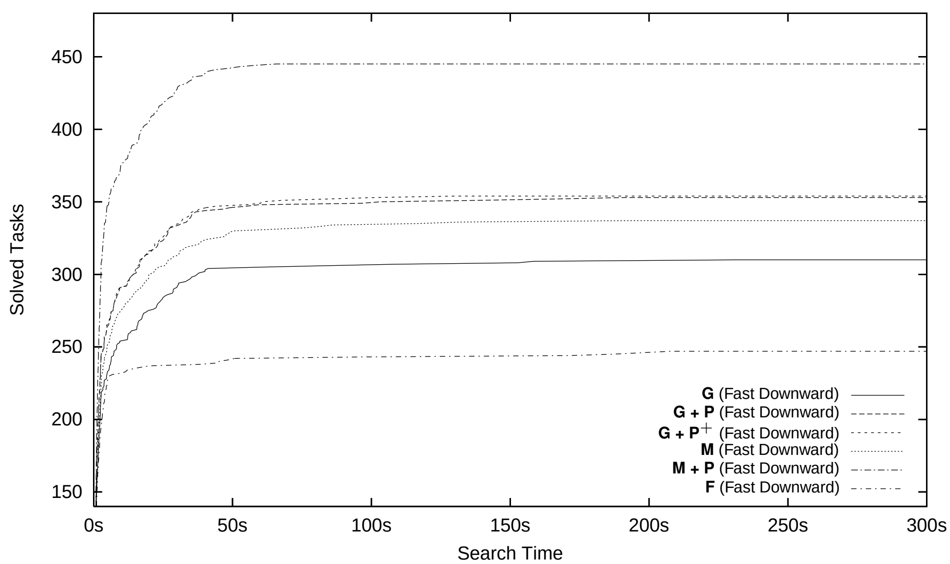

Figure 40: Number of tasks solved vs. runtime for the ADL domains from IPC1, IPC2 and IPC3. This graph shows the results for the various configurations of Fast Downward.

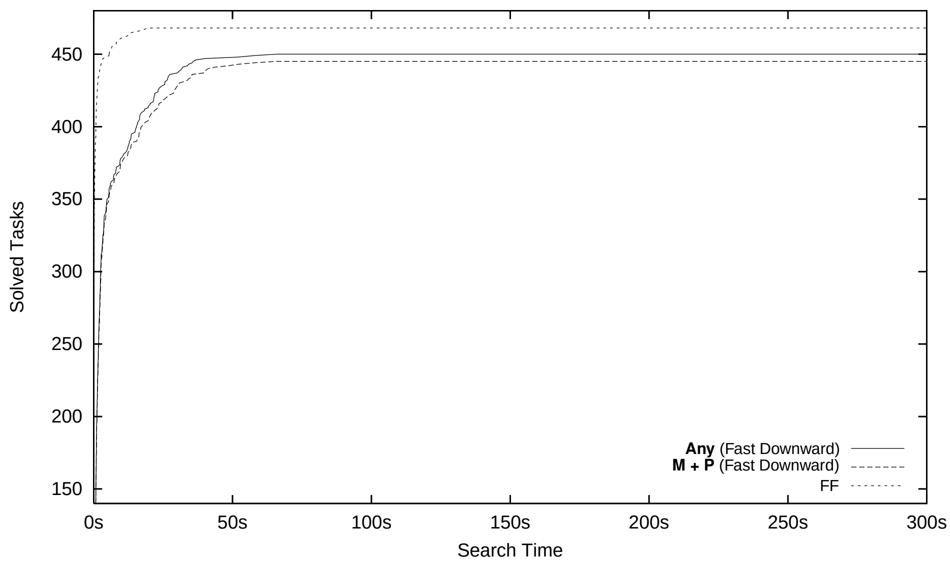

Figure 41: Number of tasks solved vs. runtime for the ADL domains from IPC1, IPC2 and IPC3. This graph shows the results for FF and the hypothetical "Any" planner that always chooses the best configuration of Fast Downward. The result for multi-heuristic best-first search with preferred operators is repeated for ease of comparison with Fig. 40.

this domain has a comparatively simple structure and should not pose such a great challenge to the planner. We hope to investigate this phenomenon further in the future.

7.6 Domains from IPC4

Third and finally, we present results for the IPC4 domains. Here, we do not compare to FF: for these benchmarks, FF is not as performant as the best planners from the competition. Besides, several of the IPC4 competitors are extensions of FF or hybrids using FF as part of a bigger system, so FF-based planning is well-represented even if we limit our attention to the IPC4 planners. For this comparison, we chose the four most successful other competition participants, namely LPG-TD, SGPlan, Macro-FF and YAHSP (cf. the results in Edelkamp & Hoffmann, 2004).

Domain	#Tasks	G	G+P	G+P$^+$	M	M+P	F	Any
AIRPORT	50	28	30	17	18	14	0	0
PROMELA-OPTICALTELEGRAPH	48	48	47	48	47	46	13	13
PROMELA-PHILOSOPHERS	48	0	0	0	16	0	21	0
PIPESWORLD-NOTANKAGE	50	25	25	23	14	7	10	7
PIPESWORLD-TANKAGE	50	36	36	36	34	17	34	14
PSR-SMALL	50	0	0	0	0	0	1	0
PSR-MIDDLE	50	0	0	0	0	0	22	0
PSR-LARGE	50	22	20	22	23	22	39	20
SATELLITE (IPC4)	36	8	0	0	8	3	22	0
Total	432	167	158	146	160	105	162	54

Domain	FDD	FD	LPG-TD	Macro-FF	SGPlan	YAHSP
AIRPORT	0	0	7	30	6	17
PROMELA-OPTICALTELEGRAPH	22	22	43	31	29	36
PROMELA-PHILOSOPHERS	0	0	1	36	0	19
PIPESWORLD-NOTANKAGE	7	11	10	12	0	0
PIPESWORLD-TANKAGE	19	34	29	29	20	13
PSR-SMALL	0	0	2	50	6	3
PSR-MIDDLE	0	0	0	19	4	50
PSR-LARGE	22	22	50	50	39	50
SATELLITE (IPC4)	3	0	1	0	6	0
Total	73	89	143	257	110	188

Figure 42: Number of unsolved tasks for the IPC4 domains. Results for the various configurations of Fast Downward are listed in the upper part, results for the competition participants in the lower part. “FD” and “FDD” denote the IPC4 versions of Fast Downward that were entered under the names “Fast Downward” and “Fast Diagonally Downward” (cf. Section 6).

Like for the previous two experiments, we report the number of unsolved tasks in each domain (Fig. 42) and present graphs showing how quickly the tasks are being solved (Figs. 43 and 44).

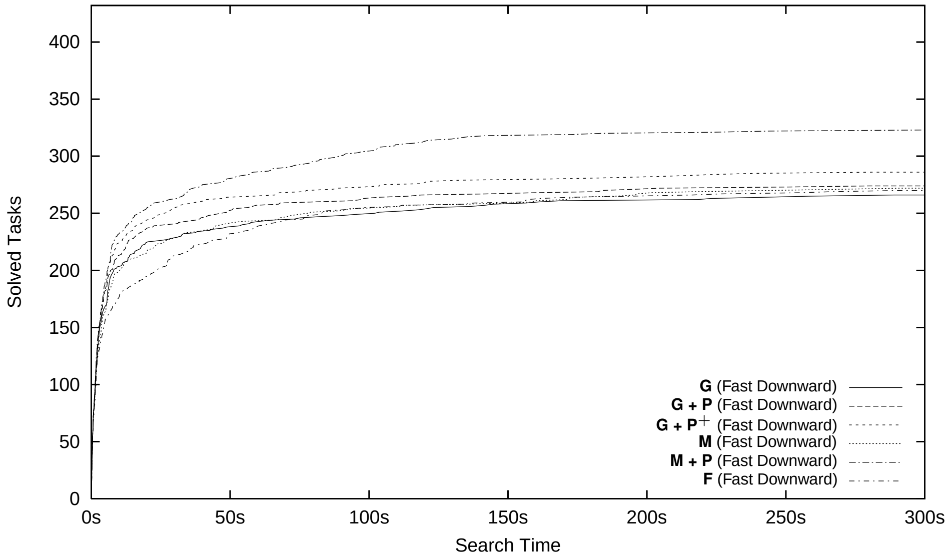

Figure 43: Number of tasks solved vs. runtime for the IPC4 domains. This graph shows the results for the various configurations of Fast Downward.

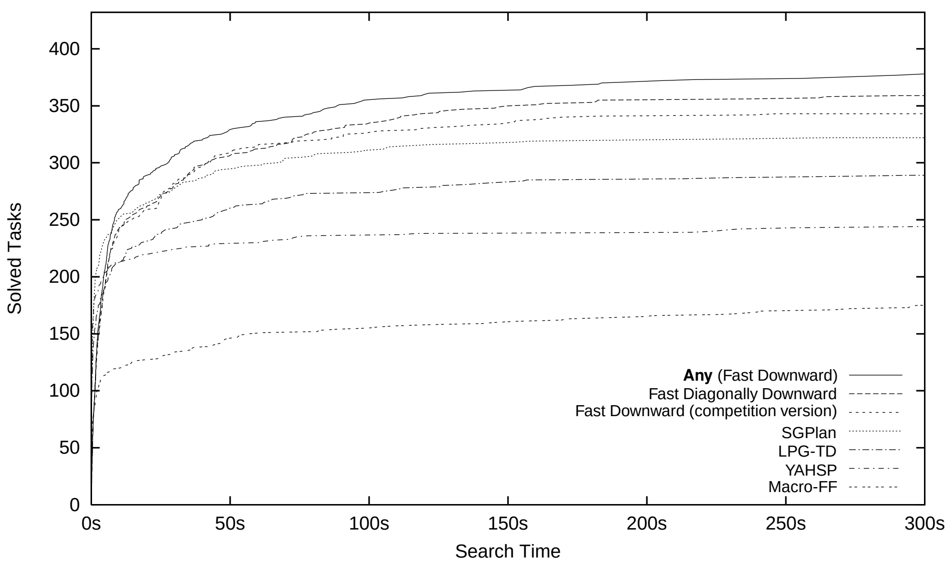

Figure 44: Number of tasks solved vs. runtime for the IPC4 domains. This graph shows the results for the hypothetical "Any" planner that always chooses the best configuration of Fast Downward, the competition configurations of Fast Downward and the best four other participants.

The results speak for themselves: Fast Downward is competitive with everyone else across domains, and better than all others in some. Only in the PIPESWORLD domains is any of the other planners noticeably better than the two competition versions of Fast Downward. This is the case for YAHSP in both PIPESWORLD domains and for SGPlan in PIPESWORLD-NOTANKAGE.

Interestingly, focused iterative-widening search performs very well on some of the benchmarks from this suite. We assume that this is related to the fact that in many of the tasks of the IPC4 suite, there is a large number of goals that can be pursued individually. Comparing the configuration **G** to **G + P**$^+$ and especially **M** to **M + P**, we see that using preferred operators is very useful for these benchmarks; even more so than in the two previous experiments. As a final remark, we point out that if we implemented the "Any" meta-planner by calling the six Fast Downward configurations in a round-robin fashion, we would obtain a planning system that could solve all but 54 of the IPC4 benchmarks within a $6 \cdot 5 = 30$ minute timeout. This is almost on par with the top performer of IPC4, Fast Diagonally Downward, which solved all but 52 of the IPC4 benchmarks under the same timeout during the competition. Thus, this is a benchmark set for which exploring different planner configurations definitely pays off.

8. Summary and discussion

We have tried to give a complete account of the Fast Downward planning system, including its motivation, architecture, algorithmic foundations and empirical behavior.

As a motivating starting point, we explained that planning tasks often exhibit a simpler structure if expressed with *multi-valued state variables*, rather than the traditional propositional representations. We then introduced *Fast Downward*, a planning system based on the idea of converting tasks into a multi-valued formalism and exploiting the causal information underlying such encodings.

We demonstrated how Fast Downward's *translation* component manages to automatically transform a STRIPS task into an equivalent multi-valued planning task with a nicer causal structure. Synthesizing monotonicity invariants is a critical point for merging several propositional variables into a single multi-valued one. We also explained the inner workings of Fast Downward's *knowledge compilation* component, demonstrating in depth what kind of knowledge the planner extracts from the problem representation, discussing *causal graphs* and *domain transition graphs*. During our discussion of Fast Downward's *search* component, we introduced its heuristic search algorithms, which can usefully integrate the information of two heuristic estimators, namely the *CG heuristic* and *FF heuristic*, and utilize preference information about operators. We also introduced Fast Downward's experimental *focused iterative-widening search* algorithm, which is based on the idea of pruning the set of operators to only consider those successor states which are likely to lead towards a specific goal.

Finally, we demonstrated Fast Downward's good performance across the whole range of propositional benchmarks from the previous planning competitions.

For the future, there are several aspects of Fast Downward that we are interested in working on. First, we intend to experiment with other search techniques along the lines of focused iterative-widening search, which emphasize heuristic forward pruning rather than heuristic state evaluation. Second, we would like to come up with an efficient heuristic for multi-valued planning tasks which does not require pruning cycles of the causal graph. Initial experiments in that direction have shown that it is difficult to achieve this goal without losing the performance of Fast Downward's heuristic estimator, but maybe better heuristic accuracy can outweigh worse per-state performance in many cases. Third, we want to investigate in how far the performance of the planner could be improved by encoding some domains differently. In some cases, merging a set of state variables which are closely interrelated into a single state variable whose domain is the product of the domains of the original state variables might be beneficial. Also, we want to test if hand-tailored encodings lead to better performance than automatically derived ones, and if so, how large the performance gap is. Fourth and finally, we would like to evaluate the behavior of the CG heuristic in specific planning domains both empirically and theoretically in the spirit of Hoffmann's work on the FF heuristic (Hoffmann, 2001, 2002) and analyze the commonalities and differences with other well-known heuristics. Hopefully, this will give some indication when we can expect good performance from the CG heuristic and when it is advisable to look for other approaches.

Acknowledgments

The author wishes to thank Silvia Richter, the other member of the Fast Downward team at the 4th International Planning Competition, for her part in implementing the planner and for valuable advice before, throughout, and after the competition. She also deserves thanks for helping out with the experiments and for proof-reading earlier drafts of this article and providing important feedback.

The STRIPS-to-MPT translator of the first version of the planner was partly based on MIPS 1.0, which was jointly implemented by the author and Stefan Edelkamp.

This work was partly supported by the German Research Council (DFG) within the Graduate Programme "Mathematical Logic and Applications" and as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

References

Bacchus, F., & Yang, Q. (1994). Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, *71*(1), 43–100.

Bäckström, C., & Nebel, B. (1995). Complexity results for SAS$^+$ planning. *Computational Intelligence*, *11*(4), 625–655.

Bonet, B., & Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1), 5–33.

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

Domshlak, C., & Brafman, R. I. (2002). Structure and complexity in planning with unary operators. In Ghallab, M., Hertzberg, J., & Traverso, P. (Eds.), *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, pp. 34–43. AAAI Press.

Domshlak, C., & Dinitz, Y. (2001). Multi-agent off-line coordination: Structure and complexity. In Cesta, A., & Borrajo, D. (Eds.), *Pre-proceedings of the Sixth European Conference on Planning (ECP'01)*, pp. 277–288, Toledo, Spain. Morgan Kaufmann.

Edelkamp, S., & Helmert, M. (1999). Exhibiting knowledge in planning problems to minimize state encoding length. In Fox, M., & Biundo, S. (Eds.), *Recent Advances in AI Planning. 5th European Conference on Planning (ECP'99)*, Vol. 1809 of *Lecture Notes in Artificial Intelligence*, pp. 135–147, New York. Springer-Verlag.

Edelkamp, S., & Hoffmann, J. (2003). PDDL2.2: The language for the classical part of the 4th international planning competition. Tech. rep. 195, Albert-Ludwigs-Universitt Freiburg, Institut fr Informatik.

Edelkamp, S., & Hoffmann, J. (2004). The classical part of IPC4: An overview. Submitted to JAIR.

Fox, M., & Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, *9*, 367–421.

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

Gerevini, A., Saetti, A., & Serina, I. (2003). Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, *20*, 239–290.

Gerevini, A., & Schubert, L. (1998). Inferring state constraints for domain-independent planning. In Rich, C., & Mostow, J. (Eds.), *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pp. 905–912. AAAI Press.

Helmert, M. (2004). A planning heuristic based on causal graph analysis. In Zilberstein, S., Koenig, S., & Koehler, J. (Eds.), *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, Whistler, Canada.

Hoffmann, J. (2001). Local search topology in planning benchmarks: An empirical analysis. In Nebel, B. (Ed.), *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pp. 453–458. Morgan Kaufmann.

Hoffmann, J. (2002). Local search topology in planning benchmarks: A theoretical analysis. In Ghallab, M., Hertzberg, J., & Traverso, P. (Eds.), *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, pp. 92–100. AAAI Press.

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

Jonsson, P., & Bäckström, C. (1998a). State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, *100*(1–2), 125–176.

Jonsson, P., & Bäckström, C. (1998b). Tractable plan existence does not imply tractable plan generation. *Annals of Mathematics and Artificial Intelligence*, *22*(3), 281–296.

Knoblock, C. A. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, *68*(2), 243–302.

Rintanen, J. (1998). A planning algorithm not based on directional search. In Cohn, A. G., Schubert, L., & Shapiro, S. C. (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pp. 617–624. Morgan Kaufmann.

Rintanen, J. (2000). An iterative algorithm for synthesizing invariants. In Kautz, H., & Porter, B. (Eds.), *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pp. 806–811. AAAI Press.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence — A Modern Approach*. Prentice Hall.

van den Briel, M., Vossen, T., & Kambhampati, S. (2005). Reviving integer programming approaches for AI planning: A branch-and-cut framework. Accepted for publication at ICAPS 2005.

Williams, B. C., & Nayak, P. P. (1997). A reactive planner for a model-based executive. In Pollack, M. E. (Ed.), *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, pp. 1178–1195. Morgan Kaufmann.