Lifting Delete Relaxation Heuristics To Successor Generator Planning

Michael Katz
IBM Watson Health, Israel
katzm@il.ibm.com

Dany Moshkovich
IBM Watson Health, Israel
mdany@il.ibm.com

Erez Karpas
Technion, Israel
karpase@technion.ac.il

Abstract

The problem of deterministic planning, i.e., of finding a sequence of actions leading from a given initial state to a goal, is one of the most basic and well studied problems in artificial intelligence. Two of the best known approaches to deterministic planning are the black box approach, in which a programmer implements a successor generator, and the model-based approach, in which a user describes the problem symbolically, e.g., in PDDL. While the black box approach is usually easier for programmers who are not experts in AI to understand, it does not scale up without informative heuristics. We propose an approach that we baptize as semi-black box (SBB) that combines the strength of both. SBB is implemented as a set of Java classes, which a programmer can inherit from when implementing a successor generator. Using the known characteristics of these classes, we can then automatically derive heuristics for the problem. Our empirical evaluation shows that these heuristics allow the planner to scale up significantly better than the traditional black box approach.

Introduction

The field of artificial intelligence has spent considerable effort on the seemingly simple problem of deterministic planning. At a high level, this problem can be formulated as: given an initial state, a desired goal, and a set of possible (deterministic) actions, find a sequence of actions which leads from the initial state to a state satisfying the goal. One popular approach to solving deterministic planning problems is heuristic search. However, two very different ways of using heuristic search algorithms to solve deterministic planning problems have been pursued throughout the history of the field.

The first approach, which we will refer to as the "black box" approach, involves implementing a piece of software to represent the planning problem. While the details of deterministic planning problems can be quite complex, it is enough to implement a very simple interface consisting of three functions: GET-INIT-STATE(), which returns an object representing the initial state, GET-SUCCESSORS(s), which returns the successors of a given state s, and IS-GOAL?(s), which checks whether the given state s is a goal state. Standard forward search algorithms, such as breadth first search, depth first search, or depth-first iterative deepening (Korf 1985), can use these three functions to solve the planning problem. However, in order to solve the problem more quickly, it is possible to use a *heuristic evaluation function*, or heuristic for short, which estimates the distance from a given state to the goal. Heuristic search algorithms such as A^* (Hart, Nilsson, and Raphael 1968) and its variants can use such a heuristic to solve the problems more quickly. Of course, the developer now also has to implement the H(s) function, in order to allow heuristic search algorithms to be used.

The second approach is the model-based approach (Geffner 2010), wherein one uses some symbolic language, such as PDDL (Mcdermott et al. 1998), to describe the planning problem. This typically involves defining a set of state variables and describing the initial state, the goal, and action preconditions and effects in terms of these state variables. It is then possible to automatically derive the same three functions mentioned above, as well as a heuristic evaluation function from the problem description (for example (Bonet and Geffner 1999)). Thus, it is possible to use the same heuristic search algorithms to solve domain-independent planning problems.

However, a major challenge for using the model-based approach to solve planning problems of interest to real-world users is that the average software developer has little to no experience with modeling. This is further compounded by the fact that some aspects of real world problems can be very hard to model symbolically, as evidenced by approaches such as planning with semantic attachments (Dornhege et al. 2009; Hertle et al. 2012) and planning modulu theories (Gregory et al. 2012), which allow the modeler to plug in external code in a general programming language to deal with specific aspects of the problem.

Our main motivation in this paper is the desire to make solving deterministic planning problems accessible to software developers who are not necessarily experts in artificial intelligence. The need to solve deterministic planning problems occurs not infrequently in real life, yet we are not aware of any frameworks which are both accessible to non-experts, and provide reasonable performance "out of the box".

In this paper, we describe such a framework, which brings the benefits of the model based approach, namely automatically derived heuristics, into black box successor generator planning. The key insight behind our framework is that, while planning problems can vary in their details, there are some common underlying principles behind the vast majority of these problems. Our framework provides an implementation of these common principles, which is transparent to the model-based view, yet can still be used inside a "black box" implementation.

Background

We now describe the two approaches we mention above in more detail. We begin by defining a deterministic planning problem over a state space, which is a tuple $\Pi = \langle S, A, s_0, S_G, f\rangle$, where S is a finite set of *states*, A is a finite set of *action labels*, $s_0 \in S$ is the *initial state*, $S_G \subseteq S$ is the set of *goal states*, and $f : S \times A \to S$ is the *transition function*, such that $f(s, a)$ is the state which applying action a in state s leads to. A *solution* to such a problem is a sequence of action labels $\pi = \langle a_0, a_1, \dots a_n\rangle$, such that $f(f(f(s_0, a_0), a_1), \dots a_n) \in S_G$ — that is, a sequence of action labels which leads from the initial state to some goal state, using the transition function f.

While deterministic planning over a state space provides a nice mathematical model, the question of how the state space is described has more than one answer. The "black box" approach uses a tuple $\Pi_{bb} = \langle s_0, succ, goal?\rangle$, where s_0 is the initial state, $succ : S \to 2^{A\times S}$ is a successor generator, and $goal? : S \to \{T, F\}$ is the goal test function. In order to obtain a "black box" description of state space planning problem Π, we use the same initial state, and define $succ(s) = \{\langle a, s'\rangle \mid f(s, a) = s'\}$, and $goal?(s) = \begin{cases} T & s \in S_G \\ F & otherwise \end{cases}$.

On the other hand, the model-based approach assumes that the state space, S, can be *factored*, and represented by a set of variables. Different mathematical formalisms for such models exist (Fikes and Nilsson 1971; Bäckström and Nebel 1995), but we will focus on describing PDDL (Mcdermott et al. 1998), which includes both a mathematical formalism and a syntax for writing text files describing a planning problem in this formalism.

For ease of presentation, we describe a limited subset of PDDL, which corresponds to STRIPS (Fikes and Nilsson 1971). A planning task in PDDL is described by a tuple $\Pi_{pddl} = \langle O, P, Op, s_0, G\rangle$, where O is a set of objects, P is a set of predicates, Op is a set of operator schemas, s_0 is the initial state, and G is the goal condition. Each predicate $p \in P$ has an arity $ar(p)$, and defines the set of boolean propositions $\{p(o_1 \dots o_n) \mid ar(p) = n, o_1 \dots o_n \in O\}$. We will denote the union of these propositions from all predicates by F. Then the set of states defined by Π_{pddl} is 2^F, the initial state s_0 is defined by a list of the propositions which are true in the initial state, and the goal condition G is a list of propositions which we want to be true in the end, that is $S_G = \{s \mid G \subseteq s\}$.

Each operator scheme $op \in Op$ has a set of named arguments, $args(op)$, as well as a list of preconditions, add effects, and delete effects. Each element in these lists is a predicate $p \in P$, with a list of arguments from $args(op)$ of size $ar(p)$. A grounded action a is obtained from op by applying a substitution $\theta : args(op) \to O$ to all preconditions, add effects, and delete effects, which results in a 3-tuple $\langle pre(a), add(a), del(a)\rangle$, such that $pre(a), add(a), del(a) \subseteq F$. We can finally describe the transition function f that is defined by Π_{pddl}, as

$$f(s, a) = \begin{cases} (s \setminus del(a)) \cup add(a) & pre(a) \subseteq s \\ s & otherwise \end{cases}.$$

Note that it is always possible to crate a PDDL description Π of a finite state space S, by defining a predicate of arity 0 for each state $s \in S$. However, the number of states that is described by this planning problem is exponential in $|S|$. Finding a *compact* PDDL description of a state space planning problem Π requires understanding the structure of Π, and is not always an easy task.

Additionally, PDDL is not always easy to deal with. For example, the occasional need to define actions with a very large number of parameters has been addressed by automatic domain transformations (Areces et al. 2014). PDDL also makes the "closed world assumption", that the only objects in the world are O. When this assumption does not hold, using PDDL planners is much more difficult (Talamadupula et al. 2010).

From Model-Based to Black Box

As previously mentioned, our objective is to provide developers who are not AI experts with off-the-shelf solvers to solve problems they are interested in. Modeling a problem in PDDL is often difficult for such non-experts. For example, many non-experts find it hard to understand why we can not define an action that moves agent A to location Y by *move*(A, Y), and why we instead need to define the action as *move*(A, X, Y). Thus, writing code to describe their problem (the "black box" approach) remains their only viable option. However, as the solver can not automatically derive a heuristic evaluation function using this approach, it is unlikely to scale.

In order to be able to combine both being able to programmatically specify the planning problem, and yet still be able to derive some heuristic guidance automatically, we propose a new framework, which we call *object oriented planning*. In this framework, the state of the planning problem is represented by a set of objects referred to as *entities*, each with their own internal state. The successor generator is defined by another set of classes, each of which represents a single operator, using two functions: IS-APPLICABLE(s,p)? which takes a state s and a list of parameters p, and checks if the action with parameters p is applicable in s, and APPLY(s,p) which returns the state resulting from applying the action with parameters p in state s. Note that while this is similar in spirit to PDDL, this is still a black box, since these functions are defined procedurally, not symbolically. The successor generator is implemented by calling IS-APPLICABLE? on the current state with all possible combinations of parameters, where each parameter can be any entity in the state. This allows the programmer to add and delete entities on the fly, a challenge with PDDL.

So far, we have described a framework which makes the "black box" approach slightly easier to use. However, the key idea behind our framework is that there is a small number of *stereotypes* of entities, which appear in many different planning domains. The developer can inherit from these stereotypes, saving some implementation effort. We also provide a number of *operator stereotypes* with known behavior. Since our framework understands these stereotypes, it can derive some heuristic guidance for these entities. The next section describes the stereotypes available in our prototype implementation.

Our current prototype implementation focuses on transportation domains, and supports two major stereotypes: *temporal*, which describes an entity with a clock, and *mobile*, which inherits the clock from *temporal* and can also be in one of several locations. Our framework also provides a *place* stereotype, which represents an immobile entity, and a *roadmap* interface, which allows the programmer to define the time and distance to travel directly between any two *places*, or to specify that they are not directly connected. For a mobile entity, we support specifying a set of temporally extended goal locations, constraining the allowed behavior of the entity.

Additionally, the framework provides *operator stereotypes*, which correspond to common operations on the entity stereotypes: *move*, which moves a mobile entity from one place to another, *load* which changes the location of a mobile entity to inside a mobile entity, and *unload*, which changes the location of a mobile entity from inside a mobile entity to the location of the entity. In the latter two cases, the clocks of all involved entities increase to reflect earliest applicability. As our framework is implemented in an object oriented language (specifically, in Java), the developer can inherit from the entity and action stereotypes, and implement the desired *additional* behavior, on top of what the framework provides.

Operators with the *move* stereotype take a *mobile* entity and a destination *place* as parameters, and update the location of the *mobile* entity to the destination, as well as incrementing the internal clock by the duration it takes to travel. Each such operator is associated with an instance of *roadmap* to use for obtaining the travel time. Thus, *walk* and *drive* can both inherit from *move*, be defined over the same set of *places*, but have different *roadmaps* defining different travel times, costs, and even connectivity. The cost of the operator is a linear combination of the travel time and travel distance, where the weights are specified by the programmer.

Operators with the *load* stereotype take two *mobile* entities, which must be in the same place, and change the location of the second to inside the first, and increment both their clocks to the maximum among their clocks plus the action duration. This represents having to meet up in the same place at the same time.

Operators with the *unload* stereotype take a *mobile* entity, which has been loaded inside another mobile entity, and unloads it, setting its location to the location of the external entity, and updates the clocks of both entities to the maximum among their clocks plus the action duration.

```
public class Vehicle extends MobileEntity {
  private int vehicleCurrentCapacity;
  private final int maximalCapacity;
  public Vehicle(String entityId, long time,
      long timeBound, Place location,
      RoutingRequest constraints,
      int maxCapacity) {
    super(entityId, time, timeBound,
                     location, constraints);
    maximalCapacity = maxCapacity;
    vehicleCurrentCapacity = -1;
  }
}

public class Participant
                      extends MobileEntity {
  private final String availableVehicleID;
  public Participant(String entityId,
              long time, long timeBound,
              Place location,
              RoutingRequest constraints,
              String vehicleID) {
    super(entityId, time, timeBound,
                     location, constraints);
    this.availableVehicleID = vehicleID;
  }
}
```

Figure 1: Commuter pooling domain Vehicle and Participant entities implementation example.

Examples

We now demonstrate the advantages of the Semi-Black Box approach on two concrete examples.

Commuter Pooling Domain

Our first example demonstrates the simplicity of modeling with the Semi-Black Box approach. We model a *commuter pooling* planning problem, where co-workers share rides on their way to work and back home.

A commuter pooling planning problem is defined by a set of participants P, which are *mobile* entities, as well as their home locations L and a work location w which are *places*. Some participants have a vehicle with limited capacity available, which is also a *mobile* entity. Figure 1 describes how these are implementated in our Semi-Black Box framework, and shows how Vehicle and Participant are implemented as classes which inherit from MobileEntity.

Each participant $p \in P$ is initially at her home location $home(p) \in L$, which she can leave no earlier than $hd(p)$, and arrive to the work location no later than $wa(p)$. On her way home, she can leave her work location no earlier than $wd(p)$, and arrive back home no later than $ha(p)$. These are implemented as temporally extended goals on p.

To model the *roadmap*, we implement the ILocationService interface, which specifies travel time and cost between different places. For each pair of locations, l_1, l_2, a duration and distance of moving from l_1 to l_2 are given by $\mathcal{T}(l_1, l_2)$ and $\mathcal{D}(l_1, l_2)$, respectively.

```
public class Drive extends Move {
  public Drive(
      ILocationService locationService){
    super("DRIVE_ACTION", Vehicle.class,
            Place.class, locationService,
            1, 0, 0);
  }

  @Override
  public boolean isApplicable(IState state,
          IEntity[] params) {
    Vehicle v = (Vehicle)params[0];
    return
      (v.getVehicleCurrentCapacity()>-1)
      && super.isApplicable(state,params);
  }
}
```

Figure 2: Implementation of Drive action in commuter pooling domain.

```
public class Board extends Load {

  public Board(){
    super("BOARD_ACTION", Participant.class,
          Vehicle.class, 1);
  }

  @Override
  public boolean isApplicable(IState state,
                      IEntity[] params){
    if (!super.isApplicable(state,params))
      return false;

    Participant p = (Participant)params[0];
    Vehicle v = (Vehicle) params[1];
    int capacity =
             v.getVehicleCurrentCapacity();

    if (capacity == -1)
      return p.getAvailableVehicleID()
                 .equals(v.getEntityId());

    if (capacity < v.getMaximalCapacity())
      return !p.getAvailableVehicleID()
                 .equals(v.getEntityId());

    return false;
  }

  @Override
  public void apply(IState state,
                          IEntity[] params){
    super.apply(state, params);

    Vehicle v = (Vehicle)params[1];
    int capacity =
           v.getVehicleCurrentCapacity();
    v.setVehicleCurrentCapacity(capacity+1);
  }
}
```

Figure 3: Implementation of Board action in commuter pooling domain.

Each participant's vehicle is initially located at that participant's home location. Each participant with an available vehicle can board/disembark that vehicle as a driver and any vehicle as a rider, as long as its full capacity is not reached. Vehicles with boarded drivers can drive between two connected locations. Figure 2 shows the implementation of the drive action, which inherits from Move. Note that the isApplicable method is overriden, and an extra check for checking if there is a driver in the car (v.getVehicleCurrentCapacity() > -1) is added.

Figure 3 shows the implementation of the board action, which checks if the vehicle is full or not by comparing current occupancy to the passenger capacity. In PDDL , this would have required having named slots for each seat. We omit the description of the other actions for the sake of brevity.

Evolution Domain

Our second example demonstrates that the Semi-Black Box approach is more expressive than PDDL. Our objective here is to create an organism that will merge the qualities of several organisms, a common task in evolutionary biology.

An Evolution planning task is defined by a set of organisms, who are either male or female. Each of them is initially at some location, and can move between locations. Two organisms of an opposite sex can reproduce, given that they are at the same location.

Given a subset of organisms G, the goal is to obtain a new organism, whose predecessors contain all organisms in G. Note that this planning problem involves creating an unknown number of new entities, and is therefore beyond the ability of PDDL to express.

Planning with Semi Black Box Representations

Having described our representation framework, we must now describe how we can solve problems formulated in this representation. We have already described how we can implement a successor generator and a goal test, and therefore we can use any uninformed search algorithm, such as BFS, DFS, ID-DFS (Korf 1985), *etc.* to solve the problem. However, uninformed search will not scale to large problem sizes.

In order to be able to scale up, we must make use of the extra information we have available — the model-based *portion* of the representation. Since we already have some known operator stereotypes, we exploit our knowledge of how these affect some aspects of the entities they are applied to, which also have known stereotypes. We do this by deriving a heuristic evaluation function, which estimates the distance from a given state to the goal. This allows us to use informed search algorithms, such as GBFS or weighted A* and solve larger problems.

Our framework provides operator stereotypes with known behavior, and entity stereotypes with known properties. Therefore, we derive a heuristic estimate of the distance to the goal by first "projecting" the problem onto its known aspects (that is, the known properties of entities and known behavior of operators), and then deriving a heuristic estimate for this projection. Note that this projection is not a true abstraction in the formal sense of the word, as an operator with a known stereotype can modify its inherited behavior in arbitrary ways. However, that would constitute poor software engineering, and our purpose here is to provide a useful tool for software developers. Furthermore, even if the programmer did do this, it would only lead to inaccurate heuristic estimates, but will never affect the correctness of the plan that is returned.

We illustrate this point for our prototype implementation on mobile entities, and provide a PDDL-like description of this projection. The objects in our PDDL description are the set of entities and locations. The predicates we use are:

- Each mobile entity E can be in location L (at(E, L))
- Each mobile entity E can be inside another entity E' (in(E, E'))
- Each temporal entity (including mobile ones) has a clock with value T (time(E, T))
- For each mobile entity's temporally extended goal locations G, we need to indicate whether it was satisfied or not (satisfied(E, G)).

Finally, we can describe the effects of *move*, *load*, and *unload* using the above predicates.

While one might think it is possible to use any of the existing heuristics from the model-based planning community, there is a subtle issue here — unlike in PDDL, it is possible to add or delete entities on the fly in our framework. Therefore, if we ground the projection according to the initial state, as is commonly done in model-based planning, we might end up deriving a heuristic for the wrong problem as soon as some entity is added or deleted. Another issue is that with *temporal* entities, we can not ground their clocks, as the domain of the variable is all non-negative integers, and is thus unbounded. Therefore, we opt for computing a heuristic estimate over a lifted representation.

Devising meaningful estimates from lifted representation poses a challenge to the planning community (Ridder and Fox 2014). Our current implementation is a lifted variant of the $h_{\mathrm{FF}}(\Pi^C)$ heuristic (Keyder, Hoffmann, and Haslum 2014; Hoffmann and Fickert 2015). Given a set of sets of facts C, $h_{\mathrm{FF}}(\Pi^C)$ finds a semi-relaxed plan, in which delete effect interactions between the fluents in each set $X \in C$ are preserved. In our framework, these sets of fluents correspond to $\{\text{AT}(E) \cup \text{IN}(E) \cup \text{TIME}(E) \cup \text{SATISFIED}(E) \mid E \text{ is a moblie entity}\}$, where

- $\text{AT}(E) = \{\text{at}(E, L) \mid L \text{ is a location}\}$,
- $\text{IN}(E) = \{\text{in}(E, E') \mid E' \text{ is an entity}\}$,
- $\text{TIME}(E) = \{\text{time}(E, T) \mid T \text{ is a clock value}\}$, and
- $\text{SATISFIED}(E) = \{\text{satisfied}(E, G) \mid G \text{ is a temporally extended goal location}\}$.

Naturally, these sets are quite large, and impractical to be exploited in the grounded setting. In our framework, however, these sets correspond exactly to the possible values of mobile entities.

Specifically, we construct a variant of the relaxed planning graph, which is a layered graph, describing the relaxed action application from a given state. The layers are added until a fixpoint is reached, that is no new relaxed entity is added. During the construction of the graph, a successor generator is used to create concrete grounded instances of the *move*, *load*, and *unload* actions and add these instances to the graph. Additionally, if we can achieve some entity E being at some location L at two different times, we only keep the earliest. Thus, the overall procedure is guaranteed to terminate, since the aforementioned actions change mobile entities, with location having a finite number of possible values. Since each layer adds at least one such modified mobile entity to the graph, the overall bound on the number of layers is polynomial in the number of these values.

Once the relaxed planning graph is constructed, the last layer is checked to consists of a representative with all temporally extended goals on locations achieved for each mobile entity of the evaluated state. If that does not hold, an infinity value is returned. Otherwise, similarly to h_{FF}, the heuristic is computed using *best supporters* from either $h_{\max}$ or h_{add} heuristics on the nodes of the constructed layered graph[1] (Bonet and Geffner 2001; Keyder and Geffner 2008).

In order to speed up the heuristic computation, we introduced a simple dead end detection check, validating that each mobile entity can reach each location it is explicitly constrained to visit within the defined temporal bounds in a relaxed fashion. We note that our implementation of the heuristic function is rather naive and can be significantly sped up by introducing sophisticated data structures, etc.

Related work

We are not the first to identify the difficulty of using symbolic languages such as PDDL to model some interesting, useful planning problems. Functional STRIPS (Geffner 2000; Francés and Geffner 2015) introduces function symbols which can be nested, and thus allows us to have objects without explicit names — something that PDDL does not support.

Planning with semantic attachments (Dornhege et al. 2009; Hertle et al. 2012) and planning modulu theories (Gregory et al. 2012) both allow the user to combine symbolic models with more expressive modules (or theories), which are implemented as external function calls to a generic programming language. These external calls are tied into the symbolic model via an interface involving a set of predicates of the symbolic model.

None of the approaches described above alleviate the need for symbolic modeling. In fact, they force the user to think of a good abstraction for the external modules, which will serve as the interface. Our approach, on the other hand, frees the user from the need for symbolic modelling, except where she specifically chooses to do so.

[1]We currently do not implement the preferred operators feature that proved to be extremely helpful in the model-based planning, leaving it for the future work.

	Plan cost				Quality				Total time				
	LAMA	FF	SBB	BB	LAMA	FF	SBB	BB	optic	LAMA	FF	SBB	BB
02_0	1108	1108	1108	1108	1.00	1.00	1.00	1.00	0.1	0.2	0.1	0.1	0.1
02_1	1108	1108	1108	1108	1.00	1.00	1.00	1.00	0.0	0.1	0.1	0.1	0.0
04_0	2176		1136		0.52	0.00	1.00	0.00	8.3	1322.5		280.4	
04_1	1136	1136	1136		1.00	1.00	1.00	0.00	0.3	1203.7	737.8	29.3	
04_2	1140	1140	1140	1140	1.00	1.00	1.00	1.00	22.2	93.2	25.9	1.5	2.0
04_3	1136	1136	1136	1136	1.00	1.00	1.00	1.00	0.2	4.8	1.1	0.3	0.6
06_0	5324		4332		0.81	0.00	1.00	0.00		50.3		3.5	
06_1	5364		5374		1.00	0.00	1.00	0.00		22.2		5.3	
06_2	4304		3270		0.76	0.00	1.00	0.00	437.4	35.1		1.1	
06_3	3264	3304	2244		0.69	0.68	1.00	0.00		22.6	627.0	471.7	
06_4	2244	2244	2244	2244	1.00	1.00	1.00	1.00		54.7	304.6	6.7	25.9
08_0	7472		5426		0.73	0.00	1.00	0.00		151.3		39.0	
08_1	6412		6402		1.00	0.00	1.00	0.00		208.6		6.7	
08_6			∞		0.00	0.00	1.00	0.00				239.9	
10_0	9560				1.00	0.00	0.00	0.00		244.6			
10_2	8560				1.00	0.00	0.00	0.00		368.4			
Sum					13.51	6.68	14.00	5.00					

Table 1: Empirical Results on Commuter Pooling Domain.

task	03_3	04_3	04_4	06_3	06_4	08_3	08_4	10_3
cost	250	250	340	240	320	220	320	220
time	0.564	0.311	0.511	1.27	1.224	0.482	6.88	2.368

Table 2: Empirical Results for SBB on Evolution Domain.

Empirical evaluation

In order to empirically evaluate the effectiveness of solving complex problems with the semi-black box approach, we implemented the approach in Java, together with the greedy best-first search, and the lazy weighted A^* search. The comparison was performed on 25 generated problems of an increasing size of the commuter pooling domain. The results are depicted in Table 1, showing the instances where at least one of the planners was able to find a solution. We used a 2GB memory bound and 30 minutes time bound on a single core of an Intel(R) Core(TM) i7 2.5 GHz machine.

Our approach (SBB in Table 1) performs an iterative search with found solution cost passed as an upper bound to the next iteration, similarly to the LAMA planner (Richter and Westphal 2010). We start with a greedy best first search, and then weighted A^* with decreasing weights 5, 3, 2, and 1, continuing with weight 1 until no solution is found. First, we compare our approach to a state-of-the-art temporal planner *optic* (Benton, Coles, and Coles 2012). Second, we compare to the pure black box approach (BB in Table 1) — BFS without the automatically derived heuristic.

The commuter pooling domain corresponds to a *temporally simple* fragment of temporal planning, and thus can be mapped to STRIPS in linear time (Cushing et al. 2007). Therefore, we also compare to two classical planners, manually adjusting the time granularity and manually removing (unrecognized by the preprocessor) unreachable time values, to allow for successful grounding of reasonable size tasks. We used the Fast Downward planning framework (Helmert 2006) with two configurations: an iterative search with the FF heuristic (Hoffmann and Nebel 2001) without preferred operators, which is the closest configuration to our solution method (FF in Table 1), and the state-of-the-art LAMA planner (Richter and Westphal 2010).

The leftmost part of Table 1 shows the best found plan cost for four of the approaches that aim at optimizing plan cost. The middle part shows the best obtained solution quality, which is a standard IPC score allocating a number between 0 and 1 to each run, where 1 is given to a planner that found the best solution for that task, and 0 stands for not being able to solve the task within the given bounds. The rightmost part shows the total run time until the best solution was found.

As these results show, SBB outperforms all other planners on IPC score. Comparing to the second best performer, LAMA, LAMA solves two instances that SBB did not, while SBB solves one instance that LAMA did not (proving that it is infeasible). On instances that they both solve, SBB is typically much faster, except for a single instance. A comparison to the most similar technique, FF, shows that SBB is much better, indicating that there is some value in the lifted heuristic computation. Finally, comparing to BB shows the automatically derived heuristic is essential.

In addition, to test the feasibility of our approach for solving tasks outside the PDDL fragment, we performed an evaluation of the Evolution domain. The results are depicted in Table 2. The tasks are named x_y, where x is the number of initially existing organizms, and y is the size of G, the subset of organizms that should be among the predecessors of the target organizm. The results clearly show that our approach is able to cope with sufficiently large instances.

Discussion and future work

We introduce a framework that brings the benefits of the model based approach into black box successor generator planning by allowing annotating planning problem entities and actions with certain predefined stereotypes. By that, we take a major step toward making solving deterministic planning problems accessible to software developers who are not necessarily experts in artificial intelligence.

For future work, we intend to extend our framework by both introducing and exploiting additional stereotypes, and by introducing additional search enhancements, such as additional automatically derived heuristics (landmarks, abstractions) and search boosting techniques, such as preferred operators.

References

Areces, C.; Bustos, F.; Dominguez, M. A.; and Hoffmann, J. 2014. Optimizing planning domains by automatic action schema splitting. In *Proc. ICAPS 2014*.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:625–656.

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal planning with preferences and time-dependent continuous costs. In *Proc. ICAPS 2012*.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proc. ECP 1999*, 360–372.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is temporal planning really temporal? In *Proc. IJCAI 2007*, 1852–1859.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Proc. ICAPS 2009*.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3/4):189–208.

Francés, G., and Geffner, H. 2015. Modeling and computation in planning: Better heuristics from more expressive languages. In *Proc. ICAPS 2015*.

Geffner, H. 2000. Functional strips: A more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*. Springer US. 187–209.

Geffner, H. 2010. The model-based approach to autonomous behavior: A personal view. In *Proc. AAAI 2010*.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Proc. ICAPS 2012*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics* 4(2):100–107.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)* 26:191–246.

Hertle, A.; Dornhege, C.; Keller, T.; and Nebel, B. 2012. Planning with semantic attachments: An object-oriented view. In *Proc. ECAI 2012*, 402–407.

Hoffmann, J., and Fickert, M. 2015. Explicit conjunctions without compilation: Computing $h^{ff}(pi^{c})$ in polynomial time. In *Proc. ICAPS 2015*, 115–119.

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

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proc. ECAI 2008*, 588–592.

Keyder, E. R.; Hoffmann, J.; and Haslum, P. 2014. Improving delete relaxation heuristics through explicitly represented conjunctions. *J. Artif. Intell. Res. (JAIR)* 50:487–533.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Mcdermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)* 39:127–177.

Ridder, B., and Fox, M. 2014. Heuristic evaluation based on lifted relaxed planning graphs. In *Proc. ICAPS 2014*.

Talamadupula, K.; Benton, J.; Schermerhorn, P. W.; Kambhampati, S.; and Scheutz, M. 2010. Integrating a closed world planner with an open world robot: A case study. In *Proc. AAAI 2010*.