The Effect of Domain Modeling on Performance of Planning Algorithms

Roman Barták, Jindřich Vodrážka
Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 25, 118 00 Praha 1, Czech Republic

Abstract

It is known that a planning domain model influences performance of planners, but so far there have been only limited studies how exactly models influence planners' performance. There are also no guidelines how to design efficient planning models. This paper studies several modeling techniques, namely factored and structured representations of states, heuristics, and control knowledge and their influence on performance of planning algorithms when iterative deepening and branch-and-bound search procedures are used. The Picat planning module is used to demonstrate the techniques on classical planning domains from International Planning Competitions.

Automated planning is a core area of Artificial Intelligence with decades of research history. Despite of big progress in planning techniques, there has not been principal change of the modeling approach since the STRIPS planning model (Fikes and Nilsson 2002) that originated at the Shakey project (Nilsson 1984). This original idea how to represent a planning domain is implemented in the Planning Domain Definition Language (PDDL) (McDermott 1998) introduced for International Planning Competitions (IPCs) and is used as a de-facto standard modeling language by the planning community. There exist some extensions of PDDL towards uncertainty and time, there exist timeline-based approaches used mainly in space applications, and there exist extensions towards including control structures, namely Hierarchical Task Networks (Nau et al. 2003) and control rules (Bacchus and Kabanza 2000; Kvarnström and Magnusson 2003). Other formal modeling approaches such as various action languages (Gelfond and Lifschitz 1998) are less widespread in planning as they focus on expressivity and semantics rather than on planner's performance. Still, automated planning techniques are rarely used in practice, for example in robotics and computer games, by people that are not experts in planning despite the effort to separate problem modeling from the actual solving (planning) algorithm. We believe that one of the reasons could be that there are actually no guidelines how to design efficient planning domain models. Though it is clear that the domain model has a crucial importance for planner' performance, the research focus is still much higher on planning algorithms than on domain modeling.

Some papers (Howe and Dahlman 2002; Vallati et al. 2015) showed that even small syntactic changes such as different order of operators and predicate definitions influence performance of planners. Other papers (Zhou and Dovier 2013; Barták and Zhou 2014; Zhou et al. 2015) showed that a different modeling approach used in the Picat programming language (Picat 2015) that embeds ad-hoc state representation, heuristics, and control structures can easily beat domain-independent planners and be on par with domain-dependent planners (Barták et al. 2015) while preserving the size of models comparable to PDDL models. Still, it was not clear how these modeling techniques contribute to performance of planners, how they interact between themselves, and how they relate to the search algorithm.

In this paper we attempt to address the problem of understanding how domain modeling influences performance of planning algorithms. In contrast to (Vallati et al. 2015) we focus on semantic techniques, namely representation of states (factored vs structured), heuristics, and control knowledge. Because of this, we decided to exploit the `planner` module of the Picat language that provides a full power of a programming language. The idea is taking the modeling techniques from (Barták et al. 2015) and showing how they contribute to planner's performance using existing domains from IPCs.

Background on (Picat) Planning

Automated planning deals with the problem of finding a sequence of actions called a plan that changes the given state of the world to a state satisfying a certain goal condition. Hence this so called classical planning corresponds to the problem of finding a path in a directed graph, where nodes describe states of the world and arcs correspond to state transitions via actions. Formally, let $\gamma(s, a)$ describe the state after applying action a to state s, if a is applicable to s (otherwise the function is undefined). Then the planning task is to find a sequence of actions $\langle a_1, a_2, \ldots, a_n\rangle$ called a *plan* such that, s_0 is the initial state, for each $i \in \{1, \ldots, n\}$, a_i is applicable to the state s_{i-1}, $s_i = \gamma(s_{i-1}, a_i)$, and s_n is a goal state. For solving cost-optimization problems, a non-negative cost is assigned to each action and the task is to find a plan with the smallest cost. The major difference from classical path-finding is that the state space for planning problems is enormous and does not fit in memory. Hence a compact representation of states and actions (and state transitions) is necessary. This representation is called a *domain model*.

Since the time of Shakey The Robot (Nilsson 1984) the *factored representation* of states is used, which has been reflected later in the design of the Planning Domain Definition Language (PDDL) (McDermott 1998). Thanks to International Planning Competitions (IPCs), PDDL is today a defacto standard modeling language for describing planning domains and problems. In this representation a state consists of a vector of attribute values and actions are changing values of certain variables (action effect) while requiring values of some attribute variables as preconditions.

Even more advanced is a *structured representation*, where a state includes objects possibly with attributes as well as relationships between the objects (Russell and Norvig 2009). Models based on first-order logic are close to this representation, but we are not aware about any widely-used planning domain modeling language based on structured representation that leads to efficient planners. The Picat `planner` module supports both factored and structured representations of states – the state is represented by any term – which was one of the reasons for selecting the Picat in this study.

The detailed description of Picat domain models can be found in (Picat 2015; Barták et al. 2015). Briefly speaking the planning domain model in Picat is expressed as a set of rules describing the transition function γ in the form

```
action(+State,-NextState,-Action,-Cost),
    precondition,
    [control_knowledge]
?=>
    description_of_next_state,
    action_cost_calculation,
    [heuristic_and_deadend_verification].
```

The Picat planner uses two search approaches to find optimal plans. Both of them are based on depth-first search with tabling and they correspond to classical forward planning. They start in the initial state, select the first action rule that is applicable to the current state, apply the rule to generate the next state, and continue until they find a state satisfying the goal condition (or the resource limit is exceeded). In case of failure, an alternative action rule is tried.

The first approach starts with finding any plan using the depth-first search. The initial limit for plan cost can (optionally) be imposed. Then the planner tries to find a plan with smaller cost so a stricter cost limit is imposed. This process is repeated until no plan is found so the last plan found is an optimal plan. This approach is very close to *branch-and-bound* technique (Land and Doig 1960). Note that tabling is used there – the underlying solver remembers the best plans found for all visited states so when visiting the state next time, the plan is retrieved rather than looked for again.

The second approach exploits the idea of iteratively extending the plan length as proposed first for SAT-based planners (Kautz and Selman 1992). It first tries to find a plan with cost zero. If no plan is found, then it increases the cost by 1. In this way, the first plan that is found is guaranteed to be optimal. Unlike the *IDA* search algorithm* (Korf 1985), which starts a new round from scratch, Picat reuses the states that were tabled in previous rounds.

Picat uses linear tabling (Zhou et al. 2008) to memorize answers to calls which implicitly prevents loops and brings properties of graph search (not exploring the same state more than once) to classical depth-first search used by Prolog-like languages. Tabling can also be used to remember "best" answers so it can be exploited to find optimal plans. Picat supports so called resource-bounded search that prevents exploring paths exceeding a given length (Zhou 2014).

Domain Models

We have selected several domains from past IPCs to compare the modeling approaches described in (Barták et al. 2015). Namely we selected Depots introduced in IPC 2002, Nomystery and Visitall from IPC 2011, and Childsnack from IPC 2014. For selecting the domains, we used several criteria. First, we looked for domains with some natural control knowledge and heuristics. In other words there should be some "common sense" guidelines that humans would use to solve problems in these domains. Second, the number of operators should be kept small as we need to control how the domain models evolve. Third, we looked for domains from different areas that would likely show different properties of domain models. Finally, we wanted a mix of old well-known domains and new domains to show that the studied modeling techniques are applicable not only to domains used for benchmarking for a long time but also to new domains.

In this section, we will explain factored and structured representations for each domain and we will describe heuristics and control knowledge proposed for these domains. The factored representations basically mimic the original PDDL encodings of the domains. The structured representations as well as heuristics and control knowledge are tailored for each domain separately though the reader can observe some unifying principles.

Depots

Depots is a combination of other well known planning domains: Logistics and Blocksworld. They are combined to form a domain in which trucks can transport crates around and then the crates must be stacked onto pallets at their destinations. The stacking is achieved using hoists, so the stacking problem is like a blocks-world problem with hands. Trucks can behave like "tables", since the pallets on which crates are stacked are limited.

The factored representation mimics the original PDDL domain model. This model uses predicates `at/2` to describe location of crates, hoists, pallets, and trucks, predicates `on/2` and `clear/1` to model towers of crates, predicates `lifting/2` and `available/1` to describe that hoist holds a specific crate or it is empty, and finally predicate `in/2` specifying that a crate is loaded to some truck. In Picat, we model a set of predicates as an ordered list to make set operations efficient. For example the set of predicates:

```
{at(p0,l3), at(p1,l4), at(p2,l3), at(p3,l2)}
```

is modeled as an ordered list :

```
[[p0|l3], [p1|l4], [p2|l3], [p3|l2]]
```

In summary, we model the state as a tuple {`ClearList, AvailList, AtPred, OnPred, InPred, LiftPred`} where each component is a list of attribute tuples for a corresponding predicate.

The structured representation exploits the fact that several predicates describe a compound object such as a tower or a truck with crates loaded to it. We model each tower as a list of crates that forms it (the top crate is first in the list). The last element of this list is a palette so via a rigid predicate `at/2` we can find the location of the tower. Note that rigid predicates can be stored as Picat facts and they are not represented in states. We model each hoist as a pair of its location and loaded crate (or an empty list if no crate is being lifted) and similarly we model each truck; the loaded crates are represented as an ordered list of crate names. In summary, we model the state as a tuple {`Towers,Hoists,Trucks`}. The structured representation is more compact, but otherwise it is not significantly different from the factored representation.

To estimate the number of actions to the goal, we adapted the classical heuristics for the Blocksworld domain. If a crate is wrongly placed and it is being lifted by an hoist then we need at least one action to drop the crate to the right place. If a wrongly placed crate is not lifted by any hoist then we need at least two actions (to lift it and to drop it).

The control knowledge for the Depots domain is also derived from the Blocksworld problem and it is very similar to control rules developed for Blocksworld (Ghallab et al. 2004). They are based on the notion of good and bad towers. The tower below some crate is good, if it is not necessary to move any of its crates, otherwise the tower is bad. The control knowledge says that a crate is dropped to a good tower (where it belongs) only and the crate is lifted from a bad tower only. Next, the crate is unloaded from a truck only if it can be dropped to a good tower at the same location. Finally, the truck moves only to location where some hoist lifts a crate (to be loaded to that truck later) or if a crate loaded at the truck belongs to that location.

Nomystery

In the Nomystery domain, there is a single truck with unlimited load capacity, but with a given (limited) quantity of fuel. The truck moves in a weighted graph where a set of packages must be transported between nodes. Actions move the truck along edges and load/unload packages. Each move consumes the edge weight in fuel so the initial fuel quantity limits how far the truck can move (no refueling is assumed).

The factored representation uses predicates `at/2` to define locations of cargo items and truck and predicates `in/2` telling that cargo item is loaded in a truck. There is also a single predicate `fuel/2` describing a fuel level of the truck. In Picat we represent these predicates as ordered lists containing the predicate arguments. Together, we represent the state using a triple {`Fuel,AtPreds,InPreds`}, where `Fuel` is the current fuel level of the truck. Recall that there is a single truck so in each state there is a single predicate `fuel(Truck,Fuel)`.

The structured representation focuses on removing object symmetries by representing objects via their (possibly changing) attributes rather than via their names. For example, we can represent each item as a pair `[CurrLoc|GoalLoc]` describing the current location and goal location of the item. The current state is then represented using the tuple {`Loc,Fuel,LCGs,WCGs`}, where `Loc` is the location of the truck, `Fuel` is the truck's fuel level, `LCGs` is an ordered list of destinations of loaded cargo items, and `WCGs` is an ordered list of unloaded (waiting) cargo items, each item is represented as we described above.

The straightforward heuristic estimates the distance to goal as follows. For each unloaded item, we need at least two actions, to load it and to unload it. For each loaded item, we need at least one action to unload it. The drive actions are "shared" between the cargo items, but we need at least as many drive actions as the number of locations to visit. The number of steps to goal is the sum of above numbers.

The control knowledge for the Nomystery domain exploits information that there is no capacity restriction of the single truck. The following restrictions can be used for ordering actions in the model and adding extra "preconditions". First, when the cargo item is delivered, it can be removed from the state representation because there will be no other actions related to this item (any action that manipulates an already delivered item only enlarges the plan). Second, the truck should load all cargo items at its current location before driving somewhere else. If any item is left there then the truck needs to return to that location to load that item later, which only enlarges the plan (recall, that there is a single truck in the domain). This can be achieved by putting the load action before the driving action and setting the action rule for loading to be deterministic (if anything can be loaded then it is loaded first). Finally, any loaded cargo is kept loaded until the truck reaches cargo's destination. This is achieved by unloading the cargo item only at its destination. We can make this rule even stronger – the truck does not leave a given location until all loaded cargo items destined to this location are unloaded. Together, we put the unload action before the drive action, we add extra condition that an item is unloaded only at its destination, and we make the action rule for unloading deterministic so all cargo items (for current location) are unloaded before trying another action. The only remaining non-determinism to be explored by search is where the truck should go.

Visitall

The Visitall domain was proposed to challenge planners based on delete-relaxation heuristics as it contains many conflicting goals. The task is for an agent placed in the middle of square grid to visit all the cells in the grid. This domain is also specific by using a single operator only for moving from one cell to a neighboring cell.

The state of the Visitall domain is very simple, it is a location (cell) of agent in the grid plus some indication of already visited cells. Hence for this specific domain, we do not distinguish between factored and structured representations as they are identical. We model the state as a pair {`Loc,ToVisit`}, where `Loc` is a the current location of the agent and `ToVisit` is a list of locations to be visited. We used this "complementary" representation of the set of predicates `visited/1` as it simplifies detection of the final state (`ToVisit` is empty). This deviates from the original PDDL model.

Good heuristics for the Visitall domain are computationally expensive. The simplest heuristic is to count the number of not-yet visited locations. We used a more advanced but still straightforward heuristic that is basically a combination (maximum) of two heuristics. One computes shortest paths to all not-yet visited nodes and takes the longest of these shortest paths. The other heuristic takes the shortest from these shortest paths (hence this path does not contain other not-yet visited locations) plus the number of remaining not-yet visited locations. The shortest paths are computed on demand and tabling is used to remember them so they are not recomputed when required later (in some sense, this is similar to running Dijkstra's algorithm several times while remembering and reusing already computed shortest paths).

As control knowledge we use a simple observation – the agent should go only to cells that are on some shortest path to any of not-yet visited cells. Any cell that does not satisfy this condition is omitted from the next move.

Childsnack

The task in the Childsnack domain is to plan how to make and serve sandwiches for a group of children in which some are allergic to gluten. There are two actions for making sandwiches from their ingredients. The first one makes a sandwich and the second one makes a sandwich taking into account that all ingredients are gluten-free. There are also actions to put a sandwich on a tray and to serve sandwiches. Problems in this domain define the ingredients to make sandwiches at the initial state. Goals consist of having all kids served with a sandwich to which they are not allergic.

The factored representation again mimics the original PDDL encoding that uses predicates to identify bread `at_kitchen_bread/1`, content `at_kitchen_content/1`, and sandwiches `at_kitchen_sandwich/1` in kitchen, sandwiches placed on trays `on_tray/2`, locations of trays `at/2`, and kinds that have been served `served/1`. There are some rigid predicates defining if the content and bread is gluten-free, while this information must be kept as a fluent for sandwiches (`no_gluten_sandwich/1`). Notice that names of sandwiches are used to identify sandwiches so predicates `notexist/1` are used to indicate that there is a prospective sandwich. Hence the "life" of sandwich starts as `notexist` which then changes to `at_kitchen_sandwich` possibly accompanied by `no_gluten_sandwich`, followed by `on_tray`, and finally disappearing after being served to a child. In summary, the state is represented as tuple {`Bread, Content, Sandwiches, OnTray, SwNoGluten, SwNames, TrayLocs, Childs`} where each component is an ordered list of names (constants) or pairs of names (in case of `OnTray` and `TrayLocs`). We used a complementary representation of predicate `served/1` so the list `Childs` represents not-yet served children.

The structured representation removes object symmetries by ignoring names of bread, content, and sandwiches, and representing each of them using either constant `no_gluten` or constant `gluten`. The not-yet made sandwich is represented by constant `free`. For children we can also ignore the names so we can represent each children using a pair {`Loc,Type`}, describing location and "type" of children (gluten or no_gluten). Also the trays are represented by their location and loaded sandwiches as a pair {`Location,Load`}, where `Loc` is a current location of the tray and `Load` is an ordered list of sandwiches loaded to that tray (recall that sandwich is only identified by its type). So the path of the sandwich starts as `free`. When the sandwich is made, its identification changes to `no_gluten` or `gluten`. If the sandwich is put on tray, it is placed to the corresponding `Load` list, and finally, if the sandwich is served then it disappears from the state. In summary, the structured state is represented by a tuple {`Bread, Content, Sandwiches, Trays, Childs`}, where the first threes components are ordered lists of object types, `Trays` is an ordered list of pairs {`Location,Load`}, and `Chils` is an ordered list of pairs {`Loc,Type`}.

For the Childsnack domain, there is a "perfect" heuristic that computes the exact number of actions to reach the goal. Each not-yet served children needs to be served and each location where some not-yet served children is located must be visited (it is possible to move directly between any pair of locations). If there are not enough gluten-free sandwiches made, they must be made. If not enough gluten-free sandwiches (made or not yet made) are placed on trays, they must be placed on trays. Similarly for other sandwiches, where we can also assume superfluous gluten-free sandwiches. Hence we can easily count how many actions of each type we need to reach the goal and this is in fact the minimal number of actions as any other action is not necessary.

There is a deterministic method to find an optimal plan. First, for each children we need to make a sandwich. Gluten-free sandwiches are made first to ensure that there is enough gluten-free bread and content. Note that this requires to modify the representation of children as we need to distinguish between children with sandwich ready for them and children with no sandwich made for them yet. The factored representation can be extended by a new predicate for it; the structured representation can add one argument to the model. Only when all sandwiches are made, they are all placed to a single tray (that may need to be moved to kitchen first). As no parallel plans are assumed, using more trays will not shorten the plan and hence a single tray is enough. That tray is then moved between locations and all children in each location are served before moving to another location. We used non-deterministic selection of the next location to visit to include some search decisions on the model.

Experimental Evaluation

The major goal of this study is showing how various modeling techniques contribute to performance of planning algorithms rather than giving the best so-far model for each domain. We started with core models, both using factored and structured representations, and we gradually added extensions with heuristics and control knowledge. This way we obtained eight models for each domain (except Visitall, where structured and factored representation are identical). We used problem instances from IPC competitions and we looked for shortest plans. Table 1 shows the number of problem instances per domain and the number of instances solved optimally by the best model. For each problem, we limited runtime to 30 minutes (if exceeded then the problem is treated as unsolved) and memory to 1GB. We have used parallel computation for our experiments (Tange 2011). The experiments run on a computer with Intel® Core^{TM} i7-960 running at 3.20GHz with 24 GB (1066 MHz).

Table 1: The number of problem instances.

domain	#instances	#optimal
Depots	20	13
Nomystery	30	30
Visitall	20	5
Childsnack	20	20

Factored vs Structured Representations

The first hypothesis is that the structured representation requires less time and memory to solve the problem. Recall, that the structured representation (among others) removes the names of objects – the objects are represented via their properties – and consequently removes object symmetries. Another question is if there are differences between search approaches depending on the used state representation. We compared pure factored and structured representations and their complete extensions with heuristics and control knowledge. Figure 1 shows how the number of problems solved optimally depends on time across all the domains.

The results clearly indicate the advantage of structured representation for both search approaches though the improvement gap diminishes when heuristics and control knowledge are used especially for iterative deepening. The same behavior can be observed for individual domains (except Visitall where both representations are identical). Even for the Depots domain, where the structured representation is not significantly different from the factored representation (no object symmetries broken), the structured representation brings some runtime improvement thanks to its smaller memory requirements, which also means faster comparison of states. We do not present the graphs for memory consumption as they show basically the same trend as graphs with time.

Heuristics and Control Knowledge

In the second experiment we looked at the effect of heuristic and control knowledge on performance of planning algorithms, in particular how they compare with each other and how they complement each other. The obvious hypothesis is that heuristics and control knowledge improve performance of planning algorithms, but the question is how much and whether there is a difference between different search approaches. One may compare different heuristics theoretically with respect to their dominance, but recall that

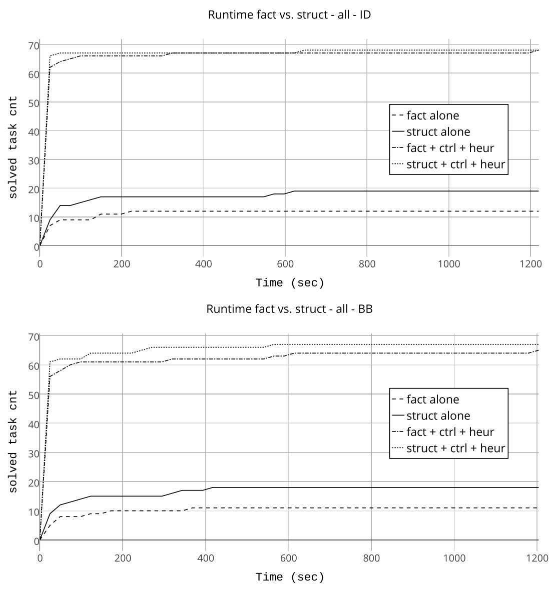
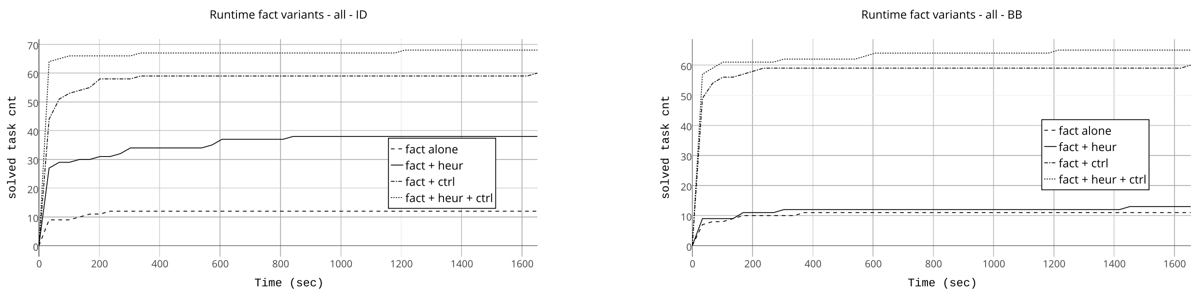


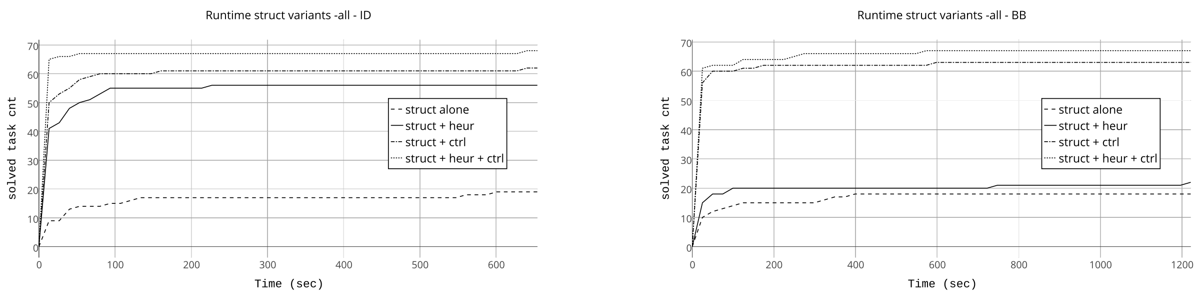
Figure 1: The number of problems solved within a given time for iterative deepening (top) and branch-and-bound (bottom).

heuristics in the presented solving approach are used differently than in the A*-family of algorithms. Heuristics are used there to cutoff sub-optimal search branches so in some sense they have a similar role as control knowledge that also cutoffs or forces some search branches but using a very different mechanism. Hence, there is no obvious "syntactic" comparison of contribution of heuristics and control knowledge. Still, based on the role of heuristics, we can fine tune the hypothesis that heuristics will be more beneficial for iterative deepening in particular to quickly find the size of an optimal plan, but less beneficial for branch-and-bound. Control knowledge is expected to contribute more to improved performance of both search approaches. Figure 2 shows how the number of problems solved optimally depends on time across all the domains for all versions of models and for both search approaches.

There are several observations taken from this experiment. First, heuristics are more beneficial for the iterative deepening approach than for branch-and-bound. The explanation is following – heuristics in iterative deepening help to find quickly the level corresponding to the size of the optimal plan while in branch-and-bound, heuristics do not contribute to find the initial plan. When finding the size of the optimal plan using iterative deepening, heuristics do not help a lot to find the actual plan. So in iterative deepening, heuristics play a significant role in the initial stage of finding the proper depth level, but they are less important when looking for the actual plan of a given length. Figure 3 shows for a single problem how much time the iterative deepening algorithm spends when looking for plans of a given length (the time is cumulative). The heuristic helps the planner to quickly skip the iterations where no plan exists (due to too



(a) factored representation



(b) structured representation

Figure 2: The number of problems solved within a given time using iterative deepening (left) and branch-and-bound (right).

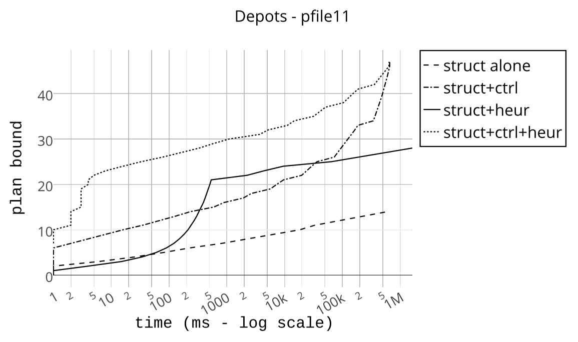
strict plan-length bound) but then the heuristic deteriorates and control knowledge helps more in the final stages of the algorithm. In branch-and-bound, the best time to use heuristic is completely opposite. Heuristics do not help when finding the initial plan as there is no bound that they can use to cutoff search branches. Nevertheless, as soon as some good bound for the plan length is found then heuristics become beneficial. In summary, heuristics are good to cutoff suboptimal plans when strong bound is given.

The role of control knowledge seems to be less dependent on the type of search approach. It almost always improved runtime performance more than heuristics. The reason is that control knowledge focuses more on finding a feasible plan while trying to omit "in-efficient" plans. There was only one exception in our test suite and this was the Visitall domain where the used control knowledge actually degraded performance. The reason was that control knowledge used expensive computation (finding many shortest paths) but did not cutoff many search branches. Though the proposed heuristic was also based on shortest paths it was much efficient there. The explanation could be that the heuristic used information about the lengths of the shortest paths while the control knowledge used less information (whether a shortest path goes through a given node). This basically indicates useful hint – if we have some information available, we should exploit it as much as possible.

Conclusions

The paper presented experimental comparison of various techniques for modeling planning domains and brought some interesting results in the area of comparing influence of different modeling techniques (heuristics and control knowledge) on planner's performance depending on used search techniques. The experiments clearly showed benefits of using structured representation, heuristics, and control knowledge. It is known that heuristics and control knowledge improve performance of planners (Haslum and Sholz 2003) but so far there was no comparison across them. Also, there was no attempt to exploit structured representation of states in planning as the state-of-the-art planning techniques are dominated by PDDL planners based on the factored representation.

The short conclusion is that if some useful information is available, it should be encoded in the domain model. The current trend in planning represented by International Planning Competitions can be characterized as "physics-only" modeling, that is, the model encodes only what the actions are doing but does not give any hints how the actions actually contribute to solve problems. The existing approaches to encode such information, namely Hierarchical Task Networks (Nau et al. 2003) and control rules (Bacchus and Kabanza 2000; Kvarnström and Magnusson 2003), are hard to grasp by users and in general there are no guidelines how to design efficient domain models (neither for PDDL, nor for HTN and control rules). There is a paper (Barták et al. 2015) giving some modeling guidelines and demonstrating that Picat models are much smaller than models with control rules, but it did not show how different modeling techniques contribute to planner's performance. In the current paper, we are narrowing this gap by experimentally comparing these techniques using four existing planning domains from IPC. One may argue that more domains would give more conclusive



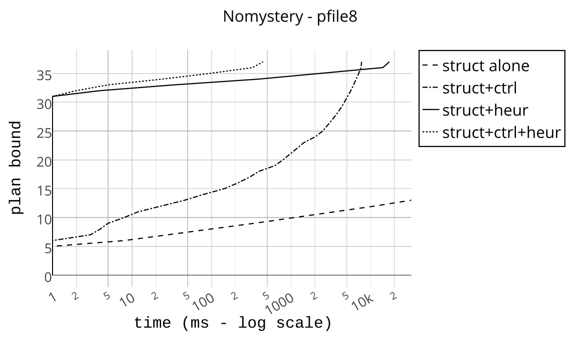


Figure 3: Relation between runtime and plan-length bound explored by iterative deepening for a single problem in the Depots domain (left) and in the Nomystery domain (right).

results, but as the results for specific domains showed the very same trends, we do not expect different conclusions if more domains were involved. In other words, the presented behavior depends more on quality of heuristics and control knowledge rather than on planning domain.

The focus of this paper was more on manual modeling of planning domains where the modeler has some knowledge about how the actions are used to solve the problem. Note that it does not mean that the modeler knows how to find the optimal plan; search is still involved. Nevertheless, an interesting open question is whether it is possible to automatically deduce some of this extra knowledge from "plain" (say PDDL) models. There are already some attempts in this direction (Fox and Long 1999; Riddle et al. 2015) including the learning track of IPC. We believe that it is important to understand first how particular modeling techniques contribute to performance, how they interact, and how they are related to used planning approach, before attempting to automatically extract them.

Acknowledgements. Research is supported by the Czech Science Foundation under the project P103-15-19877S and by the Grant Agency of Charles University under the project No. 241515.

References

Bacchus F. and Kabanza F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191.

Barták R. and Zhou N-F. 2014. Using tabled logic programming to solve the Petrobras planning problem. *Theory and Practice of Logic Programming*, 14(4-5):697–710.

Barták R.; Dovier A.; and Zhou N-F. 2015. On modeling planning problems in tabled logic programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming – PPDP'15*, 32–42.

Haslum O. and Scholz U. 2003. Domain knowledge in planning: Representation and use. In ICAPS Workshop on PDDL.

Howe A. E. and Dahlman E. 2002. A critical assessment of benchmark comparison in planning. *Journal of Artificial Intelligence Research* 17: 1–33.

Fikes R. E. 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.

Fox M. and Long D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 1999)*, pp. 956–961.

Gelfond M. and Lifschitz V. 1998. Action languages. *Electronic Transactions on AI*, 3(16),193–210.

Ghallab M.; Nau D. S.; and Traverso P. 2004. *Automated Planning: Theory and Practice*, Elsevier.

Kautz H. and Selman B. 1992. Planning as satisfiability. In *Proceedings of ECAI*, 359–363.

Korf R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1), 97–109.

Kvarnström J. and Magnusson M. 2003. Talplanner in the third international planning competition: Extensions and control rules. *J. Artificial Intelligence Research (JAIR)*, 20:343–377.

Land A. H. and Doig A. G. 1960. An automatic method of solving discrete programming problems. *Econometrica* 28(3): 497–520.

McDermott D. 1998. The planning domain definition language manual. CVC Report 98-003, Yale Computer Science Report 1165.

Dana S. Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: an HTN planning system. *J. Artificial Intelligence Research* (JAIR), 20:379–404.

Nilsson N. J. 1984. Shakey The Robot, Technical Note 323. AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

Picat web site, *http://picat-lang.org/*, Accessed August 20, 2015.

Riddle P.; Barley M.; and Franco S. 2015. Automated Transformation of Problem Representations. In *Proceedings of*

The 8th Annual Symposium on Combinatorial Search (SoCS 2015).

Russell R. and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. Pearson.

Tange O. 2011. GNU Parallel - The Command-Line Power Tool, *The USENIX Magazine* 36(1):42–47.

Vallati M.; Hutter F.; Chrpa L.; and McCluskey T.L. 2015. On the Effective Configuration of Planning Domain Models. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pp. 1704–1711, AAAI Press.

Zhou N.F. and Dovier A. 2013. A tabled Prolog program for solving Sokoban. *Fundamenta Informaticae*, 124(4):561–575.

Zhou N.F.; Sato T.; and Shen, Y.-D. 2008. Linear tabling strategies and optimizations. *Theory and Practice of Logic Programming*, 8(1), 81–109.

Zhou N.F. 2014. Combinatorial Search With Picat. *http://arxiv.org/abs/1405.2538*.

Zhou N.F.; Barták R.; and Dovier A. 2015. Planning as Tabled Logic Programming. *Theory and Practice of Logic Programming*, 16(4-5): 543–558.