Automated Transformation of PDDL Representations

Pat Riddle, Mike Barley, Santiago Franco, and Jordan Douglas
Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, 1142 NZ

Abstract

This paper describes a system that automatically transforms a PDDL encoding, calls a planner to solve the transformed representation, and translates the solution back into the original representation. The approach involves counting objects that are indistinguishable, rather than treating them as individuals, which eliminates some unnecessary combinatorial explosion.

Introduction We claim that no single representation is best, but it is clear that some representations of the same fundamental problem can be solved much faster by certain planner-heuristic combinations (Riddle, Holte, and Barley 2011). Given a single representation, *how can we efficiently produce an alternative encoding?* We describe a system that automatically transforms a given PDDL representation, finds a solution for the transformed representation, and then translates the solution back into the original representation.

A System for Representation Change We created an automated transformation system. Given a pair of PDDL files, this system produces a transformed representation. If the transformation is applicable, the planner uses this new PDDL encoding to solve the problem; if not, it returns to the original representation. After it finds a solution in the transformed space, the system then translates this solution back into the original representation. We believe this system is sound and complete and returns optimal solutions. The primary *Transform* procedure calls three subroutines: *MergeObjects*, *MergePreds*, and *ChangeRep*.

MergeObjects *MergeObjects*, uses three criteria to identify the objects which can be merged. The objects must be action-equivalent, single-valued and either goal-equivalent or initial-state equivalent. To qualify as **action-equivalent**, each action in the domain must treat the objects as indistinguishable; actions cannot, for instance, refer to these objects by name. A type is **single-valued** if every predicate with an argument of this type is mutex-ed with a predicate (not necessarily different) with an argument of this type and no predicate has more than 1 argument of this type. Objects are **single-valued** if, the type is single-valued and if only one predicate in each mutex occurs in the initial state. In gripper, balls are single-valued because they are only in one room at a time, but rooms are not single-valued because many balls can be "at" them at the same time. The third and final criterion for transformation stipulates that objects must be either goal-equivalent or initial-state-equivalent. They are **goal-equivalent** if the problem's goal description treats them as indistinguishable. Two objects, x and y, are goal-equivalent, if for every goal predicate that includes x, there must be another goal predicate that has exactly the same arguments except that where x appeared in the first predicate, y appears in the second; and vice versa. The balls in gripper also satisfy this criterion, as each predicate in the original goal description specifies that a particular ball must be in roomb. The balls are therefore the only argument that differs between these predicates. Since they are action-equivalent, single-valued and goal-equivalent, all of the balls can be merged into a single bag. In problems where only some of the objects map to the same goal value, the transformation code will make multiple bags. For instance, in the gripper domain, if the goal description had 3 of the balls in *roomb* and 3 of the balls in *rooma*, then 2 bags of balls would be retained each having 3 instances. The original and transformed representations for the examples are given at (Riddle 2015). Objects can also be merged if they are initial-state-equivalent. **Initial-state-equivalent** objects are indistinguishable in the problem's initial state. They appear in the same predicates with the same other arguments (objects and constants) in the initial state. In the barman domain, two types of objects meet the criteria for merging: shots and hands.

MergePreds *MergePreds*, merges predicates into a single macro-predicate. This is only necessary if the merged objects occur in more than one type of predicate. In barman, there are many predicates that contain a shot, so the system combines the predicates *used*, *clean*, *contains*, *empty*, *holding*, and *ontable*, to create the *(count shotX cl1 em3 on5 4)* predicate. This ensures that the bags of shots are counted correctly.

ChangeRep The third subroutine is *ChangeRep*, which alters the problem's initial state and goal descriptions, and the domain's actions, so that they support the new predicates. In Gripper, the four *at* predicates — *(at ball1 roomb)*, *(at ball2 roomb)*, *(at ball3 roomb)*, *(at ballX roomb)* — become the *(count ballX roomb 4)* predicate in the goal descrip-

tion; and two predicates, *(count ballX rooma 4)* and *(count ballX roomb 0)*, appear in the initial state. The system creates arithmetic predicates, such as *(more 0 1) (more 1 2) (more 2 3) (more 3 4)*. In the domain file, the *pick* action has *(count ?obj ?room ?num1)* as a precondition; it removes this in the effects and adds *(count ?obj ?room ?num2)*, where *(more ?num2 ?num1)*, i.e., the number of balls at that location is decreased by 1.

Translating the Solution into the Original Representation Once the planner generates a PDDL solution in the transformed space, our system translates that solution back into the original representation. There is a one-to-many mapping between the solution in the transformed space and solutions in the original space. The translator starts with the initial state of the original representation and the first action of the transformed solution. The transformer previously stored the original action from which each the new action was created. It copies over any action parameters which are parameters in the original action and are not merged objects. The transformer saved the set of original objects that it merged into bags. It picks an object from this set and places it in the action. It checks that the action can be applied in the initial state and that it matches all the additional predicates from the transformed action. For example in the first barman grasp action, the shot that is grasped must be clean and empty, since cl1 and em3 are arguments in the grasp2a action. If the action can be applied, it instantiates the action arguments and moves on to the next action. It generates the second state in the original representation (using the first action it just instantiated) and recurses. The first object it picks might not make the first action applicable in the current state. It might have to try several objects to find one that works. But it never backtracks to a previous state.

If our system merged objects that were initial-state-equivalent, it might find that the goal description and the final state of our plan do not match. For instance, our final state might now contain *(contains shot2 cocktail3) (contains shot3 cocktail1) (contains shot1 cocktail2)*, while the goal state specifies *(contains shot1 cocktail3) (contains shot2 cocktail1) (contains shot3 cocktail2)*. The system must complete a comparison of initial-state-equivalent objects and create a list of swaps. In this example: [shot2/shot1, shot3/shot2, shot1/shot3]. The solution is now traversed replacing each instance of the object with its correct replacement in each solution action.

Complexity We only bag objects that are "indistinguishable" either in the initial state or in the goal description. Additionally they are action-equivalent, so they are indistinguishable to all the operators. Since they are indistinguishable we can use a bagged representation. If there exists a solution to the bagged representation, there exists a solution in the original solution and vice versa. The complexity of the transformation is theoretically dominated by $O(|OBJ|^2)$ as the number of objects grow (or more precisely $O(|OBJ|^3)$ if the goal description or initial state is growing along with the number of objects), where OBJ is the number of merged objects. The complexity of the translation process is $|OBJ|$ x $|SP|$, where $|SP|$ is the length of the solution path.

Concluding Remarks We have developed a transformation that creates a better representation any time there are a large number of objects merged. In our previous paper (Riddle et al. 2015), we have shown that one can effect substantial savings in solution time by transforming the representation. Our transformation involves grouping into bags, objects that are indistinguishable with respect to the actions and the initial state or the goal description. By doing so, we only have to keep track of their counts instead of identifying them individually, and we avoid unnecessary combinatorial explosion. Since the transformations produce PDDL files, they can be used with any planning system seamlessly. This opens up a new approach to improving planning systems. Another advantage is that we can run with any PDDL planner and the system does not have to waste time grounding a large number of objects that are then merged during symmetry reduction.

These encouraging results suggest many directions for future work. First, we should explore additional types of transformations. A second direction for future work relates to heuristics. Certain heuristics perform poorly on these bagged representations. It is irrelevant whether they are automatically generated or not, they could have been created manually. We believe that, at present, the field is mostly focused on domain representations for which their heuristics work well. To really judge the current state of the art heuristics, we must look at very different representations.

We should create a problem solver that can take advantage of multiple representations. One possibility is a portfolio system that runs each different representation and returns the first solution. This would be effective with a few representations; but with many choices it would no longer be a viable option. Another alternative is to use the cross-over studies to predict when each representation will perform better. The final possibility is to use the RIDA* system (Barley, Franco, and Riddle 2014; Franco, Barley, and Riddle 2013), which uses sampling to determine the heuristic branching factor and the cost of applying heuristics, then uses these results to choose a heuristic to apply. We could apply this technique to multiple representations, to determine which one is most likely to solve the problem.

References

Barley, M.; Franco, S.; and Riddle, P. 2014. Overcoming the utility problem in heuristic generation: Why time matters. In *ICAPS*.

Franco, S.; Barley, M.; and Riddle, P. 2013. In situ selection of heuristic subsets for randomization in IDA* and A. In *Heuristics and Search for Domain-Independent Planning*.

Riddle, P.; Barley, M.; Franco, S.; and Douglas, J. 2015. Analysis of bagged representations in pddl. In *Heuristics and Search for Domain-Independent Planning*.

Riddle, P.; Holte, R.; and Barley, M. 2011. Does representation matter in the planning competition? In *SARA*.

Riddle, P. 2015. PDDL files for the representations. http://www.cs.auckland.ac.nz/∼pat/socs-2015/. Created: 2015-02-20.