Second Annual Conference on Advances in Cognitive Systems Poster Collection (2013) 199-216

Problem Reformulation as Meta-Level Search

Patricia J Riddle PAT@CS.AUCKLAND.AC.NZ
Michael W Barley BARLEY@CS.AUCKLAND.AC.NZ
Santiago M. Franco SANTIAGO.FRANCO@GMAIL.COM
Private Bag 92019, Department of Computer Science, University of Auckland, Auckland, 1142 NZ

Abstract

A problem's representation has a big impact on how hard a problem is to solve, ranging from easy to solve to intractable. We describe a framework for meta-level search through a space of problem representations. These new problem representations are not predefined but are generated using a set of transformations. We define a set of component transformations and describe a case study of a single composite transformation. Using this composite transformation, we automatically transform the PDDL representation of the GRIPPER domain into a new PDDL representation. In this new representation, the previously intractable GRIPPER problems can now be solved in a few seconds.

1. Introduction

A problem's representation can make it either easier or harder to solve a problem. This is true for both humans and for computers (Newell, 1966; Newell & Simon, 1972; Hayes & Simon, 1974). A natural response for a human when encountering a difficult problem, is to change the problem representation. For instance, Polya states "Trying to find the solution, we may repeatedly change our point of view, our way of looking at the problem. We have to shift our position again and again. Our conception of the problem is likely to be rather incomplete when we start the work; our outlook is different when we have made some progress; it is again different when we have almost obtained the solution." (Pólya, 1957) Our framework allows an automated problem solver to exhibit the same behavior, and change the problem representation in hopes of making the problem easier to solve.

It has been shown by many researchers since the 1970s that the ability to change a problem representation to a new (and hopefully easier to solve) representation is a key strategy that people use, and that this would be a key asset for a true artificially intelligent system(Amarel, 1968; Amarel, 1971; Simon, 1972). In this paper we give a framework for addressing this challenge. Our approach is a meta-level search through a space of problem representations. These new problem representations are not predefined but are generated using a set of transformations. To accomplish this we need a language for representing the problem representations and we need a set of transformations for moving between them. Given that we have generated a new problem representation, a problem solver can search the space defined by this new representation. If necessary, solutions in the new problem representation can be transformed back into solutions in the original representation.

Our claims are that this powerful framework enables flexible problem solving, by supporting automatic transformations from intractable problem representations to easier problem representations.

This allows a problem solver to solve new problems that it could not solve before, in reasonable time and space. We define a set of component transformations, out of which higher-level composite transformations can be defined. We explore a case study of a single composite transformation. We automatically transform the PDDL description of the GRIPPER domain into a new PDDL description, where previously intractable problems can now be solved in a few seconds.

The next section discusses the standard theory of problem reformulation. In the following sections, we discuss representing problem representations, transformations for problem reformulation, and a framework for problem reformulation. We then explore a case study of a composite transformation in the GRIPPER domain. After this we conclude and offer directions for future research.

2. Standard Theory of Problem Reformulation

We briefly review the work adopted in our framework. Early work from Gesalt psychology talks about reformulation on insight problems. A good survey of this work is Ohlsson (1992). The early research in the area of problem reformulation in Artificial Intelligence (AI) started in the 1960s (Amarel, 1968; Amarel, 1971; Simon, 1972; Korf, 1980).

Reformulation is a general term meaning a change to the way one thinks about a problem. The early AI research assumed that the problem would be solved by searching through a problem space. A problem space is defined by the states of a problem and the operators which move between the states. So the states can be thought of as nodes in a graph and the operators are the edges between them. This is called a state space search, where the initial state and the operators implicitly define the state space. To be complete, a problem representation must include the initial state, a goal description, the operators, and a set of predicates and objects from which the previous items are defined. To solve the problem a search is carried out though this space of states. The solution is a path in the graph from the initial state to a state that satisfies the goal description.

The AI type of problem reformulation moves from one state space problem description to another. This is called a "change of representation". The notion of changing from one problem representation of a state space to another problem representation of a state space, we will call the standard theory of problem reformulation, throughout the rest of this paper.

Not all state spaces are tractable, one might not be able to search through the space in a reasonable time. Of course, depending on how important the problem is or how quickly the solution is needed, "reasonable time" can grow and shrink. Some state spaces can be very large (e.g., the total number of chess positions is roughly 10^{120}). The purpose of reformulation is to move from one problem representation to another problem representation which is "more tractable", in that it takes less time to solve.

A "change of representation" (from one state space representation to another), alters the state and/or operator representation, to arrive at a new problem representation. Now you have two problem representations for the problem and you actually have three search spaces. There is the meta-level search space that moves from one problem representation to another. Each of the individual problem representations has its own base-level search space in which to solve the problem.

This meta-level space lets a problem solving system move from one problem representation to another using a set of transformations. You could be given a set of hand-coded problem represen-

tations and just move between them or you could have generative transformations that create the new problem representations (e.g., the difference between searching an explicit graph or an implicit one). A meta-level search can be conducted through the space of problem representations, trying to find a more tractable representation or more explicitly a more tractable representation for that particular problem solver.

The reason this meta-level search is required, is that there does not exist a single representation that is necessarily more tractable for all problem solvers. In previous research (Riddle, Holte, & Barley, 2011), we showed that even within PDDL domains, given the same problem solver, one problem representation was faster for one problem instance and another problem representation was faster for another problem instance. We showed the same trend between problem solvers where one problem solver preferred one problem representation for a particular problem instance while another problem solver preferred another problem representation. This motivates the need for a problem reformulation framework. If there was a perfect problem representation that would work with every problem solver and every problem instance, then once we found it we would never have to do problem reformulation again. We do not believe there is a perfect problem representation. There will always be the need to move from one problem representation to another problem representation.

3. Representing Problem Representations

In order to have a space of problem representations, we need a language in which we can represent them. We call this the representation language, and it had a critical influence on the early reformulation research. The notion of problem reformulation has been around for at least 40 years, but there have been very few systems over the years which have actually implemented these ideas. There was quite a bit of work in the 1980s in this area, some of it was only theoretical (Korf, 1980; Subramanian, 1989) and others that actually were implemented (Holte, 1988; Iba, 1989; Lowry, 1989; Nadel, 1990; Riddle, 1991). One of the major problems with this early work was that representation languages themselves were a new field in AI and there was not a consensus on what was necessary for a "good representation language". Different researchers were using different representation languages. This made it difficult to compare their results.

One way to create a research community with a shared agenda is to provide a common set of tools. In the 1980s, different researchers worked on problem reformulations but they each had their own representation language, and therefore the results were not comparable. PDDL(McDermott et al., 1998) is the planning domain definition language, which has been used by the domain independent planning community since the 1990s. A PDDL representation is defined by two files: a domain file and a problem instance file. The domain file contains the predicates and the actions. The problem instance file contains the objects, the initial state and the goal description. Examples of domain and problem instance descriptions are given later in this paper. We can now write problem reformulation transformations that change a problem representation represented in PDDL into another problem representation represented in PDDL. This is one of the main reasons that it is now a good time for a resurgence in reformulation research.

PDDL is not a perfect language, but it is being carefully extended and made more powerful. It includes numeric fluents, durative/continuous actions, derived predicates, state trajectory con-

straints, preferences, and object fluents(Helmert, 2008). It is certainly not a powerful enough language for human-level reasoning, but the opportunity to do problem reformulation in a language with a strong community should not be missed.

The International Planning Competition (IPC) for domain independent planners is a biennial competition and has been run since 1998. This competition is based on the PDDL language. All the planners submitted become public domain software. This gives researchers in problem reformulation a publicly available set of domain independent problem solvers. Any domain independent planner can run on the generated PDDL representations. We can see which representation is better for which problem solver on which problem instance (Riddle, Holte, & Barley, 2011). There are a large number of different heuristics used in these competitions. We can explore which problem solver and which heuristics are favored for each problem domain or even each problem instance.

In addition, there are new domains (each with 20 problem instances) created for every competition. This gives us a large number of different problem domains on which to test our problem reformulation transformations. Now, different researchers' problem reformulation operators can all be compared using the same space of domain independent PDDL planners. All transformations created by different research groups can be compared to one another. Reformulation researchers should take advantage of the PDDL language and publicly available domain-independent planners.

4. Transformations for Problem Reformulation

In order to define the meta-level space of problem representations, we need the nodes (problem representations) and the edges (transformation). In the last section we discussed the problem representations, in this section we will describe the transformations. The new problem representations are not prespecified, but are automatically generated by the transformation operators. In this section we will discuss the component transformations operators. In Section 4.5, we will discuss the higher level single composite transformation that we have created, which guarantees that the new problem representation and the old problem representation both share certain properties.

First we will discuss the component transformations from which the composite transformations are created. There are four main types of transformations. These transformations alter the objects, the predicates, the actions, or the initial state and/or goal description. In addition, for the component transformations we will point out whether the new representation is guaranteed to have fewer nodes or edges than the old representation or conversely more of either. This affects the size of the base-level search space which affects the amount of search performed to find a solution. We will look at each of the four types of transformations in the following sections.

4.1 Altering Objects

One of the key aspects of a representation are the objects used. These objects may be merged or split along either a part-whole or an is-a dimension. Splitting objects is frequently problematic because it can make the problem more difficult to solve, but merging objects is often useful in problem solving.

For instance, in the SOKOBAN domain (in its original formulation) it does not matter which stone is at which goal location. SOKOBAN is a type of transportation puzzle in which a man pushes boxes around a warehouse trying to get them to their final location. It is normally defined

such that it does not matter which box is at which location, only that all the crates are at final crate locations(Wikipedia, 2013). So renaming Stone-01 and Stone-02 to both be Stone-01 will remove many states from the state space, without actually losing any solutions. Problem solvers can still solve this problem with identically named objects because in the SOKOBAN domain there can only ever be 1 object at a location at a time. In the GRIPPER domain, which we will explore later, there can be more than 1 object at a location at a time, so a more complex reformulation is required. For instance, it would remove 1/2 of the states in the search space if you rename stones Stone-01 and Stone-02 to both be called Stone-01 (assuming every stone could reach every location in the problem). Of course, care must be taken with the specification of the goal description to insure that both stones are in a goal position and not just one of the stones! For instance, if you just take the standard goal description "(and (at-goal stone-01) (at-goal stone-02))" and change it to "(and (at-goal stone-01) (at-goal stone-01))", then the problem solver will put 1 stone-01 in a goal location and stop. This will not be a viable solution to the original problem. Table 1 shows a goal description for SOKOBAN which allows the two stones to have the same name, but also requires the problem solver to place both the Stone-01 stones in a goal location, not just one of them.

Most of the component transformations we explore, must be composed with multiple transformations to arrive at a "useable representation", see Section 4.5. In this case, after you merge the stones together, you need to alter the goal description to maintain a problem representation which has solutions which have the same length as the solutions in the original space. This composite transformation described for SOKOBAN is not currently implemented.

```
(define (problem p012-microban-sequential)
  (:domain sokoban-sequential)
  (:objects  dir-down - direction  dir-left - direction
    dir-right - direction  dir-up - direction  player-01 - player
    pos-1-1 - location ...  pos-9-8 - location
    stone-01 - stone  stone-01 - stone)
  (:init
    (IS-GOAL pos-6-5) (IS-GOAL pos-6-6) (IS-NONGOAL pos-1-1)
    (IS-NONGOAL pos-1-2) ...  (IS-NONGOAL pos-9-7)
    (IS-NONGOAL pos-9-8) (MOVE-DIR pos-1-5 pos-1-6 dir-down)
    (MOVE-DIR pos-1-6 pos-1-5 dir-up) ...
    (MOVE-DIR pos-9-3 pos-8-3 dir-left)
    (MOVE-DIR pos-9-3 pos-9-2 dir-up) (at player-01 pos-5-5)
    (at stone-01 pos-3-3) (at stone-01 pos-4-4) (clear pos-1-5)
    (clear pos-1-6) ... (clear pos-9-2) (clear pos-9-3))
  (:goal (and  (exists (?x - location ?y - location)
              (and (at stone-01 ?x) (at stone-01 ?y)
               (IS-GOAL ?x) (IS-GOAL ?y) (not (= ?x ?y)))))))
```

Table 1. SOKOBAN domain problem description in p01.pddl, abridged to save space

4.2 Altering Predicates

Predicates are an equally fundamental part of the problem representation, since they define the relationships between the objects. When altering the predicates, you can alter the predicate name or its arguments. You can add new predicates or delete predicates you currently have, or you can split a predicate into multiple predicates.

This can also be illustrated using the SOKOBAN example in Table 1. In the original problem representation every location that was clear had to be specified explicitly, such as (clear pos-1-5). When using a representation that allows negated action preconditions and negated goal conditions, this is easily remedied by specifying (full ?location) whenever a stone or a player is at a location. Then you can use (not (full ?location)) instead of clear in the operators' preconditions. To be precise, only (full ?location) predicates are ever stored in a state description. Within action descriptions or goal descriptions, (not (full ?location)) is used. Assuming that the problem solver uses the closed world assumption, this makes the state representations much smaller without losing any information. This transformation can be automated by analyzing the PDDL actions, to determine that (clear pos-1-5) and (at ?X pos-1-5) are mutual exclusive (only one can be true at a time).

This transformation will also cause other transformations to need to be applied. The actions themselves would need to be altered to use the new predicate. The initial state and the goal description would also have to be altered. Altering the predicates does not normally change the number of states or edges in the state space. It only changes the representation within a state. This second composite transformation described for SOKOBAN is not currently implemented.

4.3 Altering Actions

The actions allow you to move from one state to another in the state space. So they are one of the most important components. When altering actions, you can merge two actions together, or split actions apart. Alternatively you can add or delete either preconditions or effects (a PDDL effect is basically a combined add and delete list). An example of merging actions is the standard macro creation (Korf, 1985; Iba, 1989). For instance in the GRIPPER domain, Table 2, you can create a single "transfer" action that picks up two balls, moves to the other room, drops off the two objects, and returns. These macro-operators would decrease the number of nodes and edges in the search space (if you only allow the macro-operators and not the original actions as well). If you had both the macro-operators and the actions you would have the same number of states but more edges, which might make the problem harder to solve. This is called the utility problem (Minton, 1988).

The previous macro would not necessarily return a solution in the original space that was an optimal solution. It could include an extra action where "robby" moves back to rooma (robby is the name of the robot with the grippers in the GRIPPER domain). Alternative macros could also be created, such as "transfer2" which picks up two balls, moves to roomb, and drops them off, thus leaving the "move from roomb to rooma" as a separate action. This too can cause problems if there are an odd number of balls to move from rooma to roomb. Additionally if not all the balls are accounted for in the goal description, then a ball could remain in its original position or even remain in the gripper. Therefore great care must be taken in the creation of macro-operators, especially if you care about optimal solutions.

In general, macro creation does not require any further changes to the objects, predicates, the initial state or the goal description. It is an example of a transformation which could be applied independently. We currently have not implemented the automatic creation of macro-operators.

```
(define (domain gripper-strips)
   (:predicates (room ?r) (ball ?b) (gripper ?g) (at-robby ?r)
                (at ?b ?r) (free ?g) (carry ?o ?g))

   (:action move
    :parameters  (?from ?to)
    :precondition (and  (room ?from) (room ?to) (at-robby ?from))
    :effect (and  (at-robby ?to) (not (at-robby ?from))))

   (:action pick
    :parameters (?obj ?room ?gripper)
    :precondition  (and  (ball ?obj) (room ?room) (gripper <?gripper)
                         (at ?obj ?room) (at-robby ?room)
                         (free ?gripper))
    :effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
                 (not (free ?gripper))))

   (:action drop
    :parameters  (?obj  ?room ?gripper)
    :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                         (carry ?obj ?gripper) (at-robby ?room))
    :effect (and (at ?obj ?room) (free ?gripper)
                 (not (carry ?obj ?gripper)))))
```

Table 2. GRIPPER domain actions in domain.pddl

4.4 Altering the Initial State or the Goal Description

The initial state is required to define the state space and the goal description tells the problem solver when to stop, so they are both critical components of the problem representation. You can add, alter or delete predicates from either the initial state or the goal description. This will almost always change the ability of the problem solver to find all the solutions from the original space, because you are almost guaranteed to either gain or lose information.

This is not the case when you are altering the initial state or the goal description to facilitate a change in the actions, predicates or objects. It can easily be the case that you have a mapping between solutions (either one-to-one or one-to-many), but this would need to be determined for each composite transformation.

Altering the initial state or the goal description will not necessarily change the state space (i.e., the number of nodes or edges in the graph). It will change the location of the starting point or the ending point of the solution within this graph, therefore the solution could have a different length.

4.5 Composite Transformations

We are creating a meta-level search framework to search through the space of problem representations. The transformations we discussed in the last several sections, were at a very low level. Most of these transformations cannot be applied independently, because most of the problem representations in that search space would be "broken". For example, the initial state would be defined with "(not (full pos-5-5))" but the actions preconditions would specify "(clear ?L)". This would not be a useable representation, no problem solver could search this space. You cannot alter a predicate without also altering all the actions that use that predicate!

If the meta-level search framework had to search through a space where 98% of the problem representations were unexecutable, then the framework would not be a viable proposition. Therefore we combine these low level component transformations into higher level composite transformations. The advantages of these higher level transformations is that at every point in the space, the problem representations are "executable" (i.e., "not broken"). They might be better (or worse) than the previous problem representation, but they are all valid problem representations for this problem.

The composite transformations from one representation to another may preserve certain properties (like the same number of solutions) or they may not. While some of the transformations may always be desirable to apply, most of the transformations will only sometimes be desirable. This means that a search through a meta-level space of representations will be a critical component of the framework for applying these transformations.

Whenever possible these composite transformations maintain a one-to-one or a one-to-many mapping between a solution in the new problem representation and solutions in the old representation. This means that solutions found in the new representation can be transformed back into solutions in the original representation, if required.

4.6 Transforming the Reformulated Solution Back Into the Original Representation

In some cases, we are guaranteed that there is a one-to-many mapping of the solution produced in the new problem representation to a set of solutions in the original representation. For instance, in Section 6 we are guaranteed that corresponding solutions in both representations will have the same number of actions (the same solution length). In these cases, we are able to transform the solution backward by changing each action in the transformed representation's solution into a valid action in the original representation's solution. If there is a one-to-many mapping there is some search involved to find variable bindings that guarantee the solution is correct. This is typically much less search than would be needed to simply solve the problem in the original space. Namely the solution in the transformed representation gives a lot of structure to the search.

In other cases (like the macros example), there will be a difference in the number of actions between the two solutions. In these cases there will also be search involved. There might even be some plan repair, to remove extra moves off the end of the solution (like robby walking back to **rooma** unnecessarily). In our experiments, we have found the search required to transform the solution from the transformed representation back into a solution for the original representation is usually quite focused. This will be discussed in more detail in Section 6.

5. The Meta-level Search Framework

In the last two sections we have defined the meta-level search space. This consists of the representation of the states (i.e., problem representations) and the operators between them (i.e., transformation operators). Now we can describe the framework for searching over this meta-level space. Our framework uses generative transformations. These transformations generate new problem representations from the given problem representations. A meta-level search will traverse the space of representations to find a good representation for the current problem solver. There are 3 main techniques we are exploring: 1) heuristic search, 2) adaptive problem solving, and 3) portfolio systems. We will discuss these in turn.

In Section 6 we will discuss one heuristic we can use in PDDL problems to predict when one representation will be better than the other. As we implement more composite transformations, we might be able to predict when one transformation will be better with a certain combinations of problem solvers and heuristics. I should caution that a lot more work remains. We have not explored every combination of problem solver and heuristic with these two representations. So more experimentation is necessary to determine whether these heuristics are reliable.

Our second approach is to use an adaptive problem solver. RA* (Franco & Barley, 2008a; Franco & Barley, 2008b; Franco & Barley, 2009; Franco, Barley, & Riddle, 2013a; Franco, Barley, & Riddle, 2013b) uses sampling and a runtime model to predict which combination of heuristics is the "best" for solving this particular problem. It then uses that "best" heuristic combination to solve the rest of the problem. We can use the same technique for determining which of the two representations performs better. We now have a number of heuristic/representation combinations, and we use sampling to determine which will perform better.

The last option is a standard portfolio system. The meta-level framework could choose more than one representation and try to solve the problem, each of them in parallel (or by time slicing). This is a standard approach in the IPC, where several different problem solvers or one problem solver with several different heuristics are used in parallel.

The search at the meta-level does add to the overall problem solving time. Currently we have a very small set of transformations, but as we develop more transformations, heuristics must be devised for determining which transformations to apply within this meta-level space. This is a very powerful framework. In Section 6 we discuss a case study showing a transformation between two problem representations in the GRIPPER domain. The results of this study are very encouraging.

```
(define (problem strips-gripper-x-1)
   (:domain gripper-strips)
   (:objects rooma roomb ball4 ball3 ball2 ball1 left right)
   (:init (room rooma) (room roomb) (ball ball4) (ball ball3)
          (ball ball2) (ball ball1) (at-robby rooma) (free left)
          (free right) (at ball4 rooma) (at ball3 rooma)
          (at ball2 rooma) (at ball1 rooma) (gripper left)
          (gripper right))
   (:goal (and (at ball4 roomb) (at ball3 roomb) (at ball2 roomb)
               (at ball1 roomb))))
```

Table 3. Gripper domain problem description in prob01.pddl

6. GRIPPER Domain Case Study

The case study transformation we explore in this section is a combination of merging objects, changing predicates and changing the initial state and goal description. We will look at a concrete example by examining the GRIPPER domain. Bear in mind that this transformation is actually implemented as a generic, domain-independent transformation. The transformed representations for the domain and problem instance and the transformed solution were all automatically generated.

The GRIPPER domain involves a robot "robby" with two grippers "left" and "right". It has X balls (the smallest problem shown in Figure 3 has 4 balls) which it must move from rooma to roomb. All the problems in this domain involve moving all the balls from rooma to roomb. They only differ in the number of balls. The complexity of the problem is increased by the fact that it can pick up each ball in either the left gripper or the right gripper. It does not matter which it uses in the overall solution, it does increase the complexity of the problem (especially for optimal planners).

Currently (with a single composite transformation) we have not set up the meta-level search framework. We have been analyzing which representation, the original or the transformed works better for a particular problem solver. Our current problem solver is A* with the LM-cut heuristic (Helmert & Domshlak, 2009) using the Fast Downward system (Helmert, 2006). This has been one of the top optimal planners in the IPC competition for the last few years. In the GRIPPER domain, no optimal planner performs well. Helmert states "If we apply two state-of-the-art optimal planning algorithms (Haslum, 2007; Helmert, Haslum, & Hoffmann, 2007) to the GRIPPER domain, neither of them can optimally solve more than 8 of the standard suite of 20 benchmarks within reasonable run-time and memory limits,..." (Helmert & Röger, 2008).

A* using the Fast Downward system has trouble with the original GRIPPER domain for two main reasons: it is an optimal planner and it is a grounded planner. Grounded planners find the GRIPPER domain difficult because there are roughly N^2 grounded pick up operators when you have N balls and two grippers. In addition, optimal planners find it difficult to prove optimality in large search spaces.

In Figure 1 we give the timing results and nodes generated for the GRIPPER domain on the original and transformed representations. The problem solver cannot solve past problem prob07 in the original representation because it runs out of memory. The time includes the time of all transformation costs. The planner ran out of memory with ulimit set at -t 7200 -s 100000 -v 15000000.) So basically in the original representation the nodes generated and the time grows exponential with the solution length. In the transformed representation, the the nodes generated grow linearly with the solution length. While the timing appears to be constant, it actually grows quadratically.

The original GRIPPER domain PDDL file is shown in Table 2, while the original problem instance file for the first problem is shown in Table 3. In this problem instance (see Table 3), there are four objects of type ball named **ball1**, **ball2**, **ball3**, and **ball4**. There are two objects of type room named **rooma** and **roomb**. There are two objects of type gripper named **left** and **right**. Because GRIPPER domain uses an early version of PDDL, they typing is done with unitary predicates. Later versions of PDDL have explicit typing. An initial state is given where robby and all the balls are in **rooma** and both grippers are empty. The goal description specifies that all the balls are in **roomb**. In the domain file (see Table 2), the things that don't vary from problem instance to problem instance are specified. The predicates are given, in this case they are the "type predicates" **room**, **ball**, and

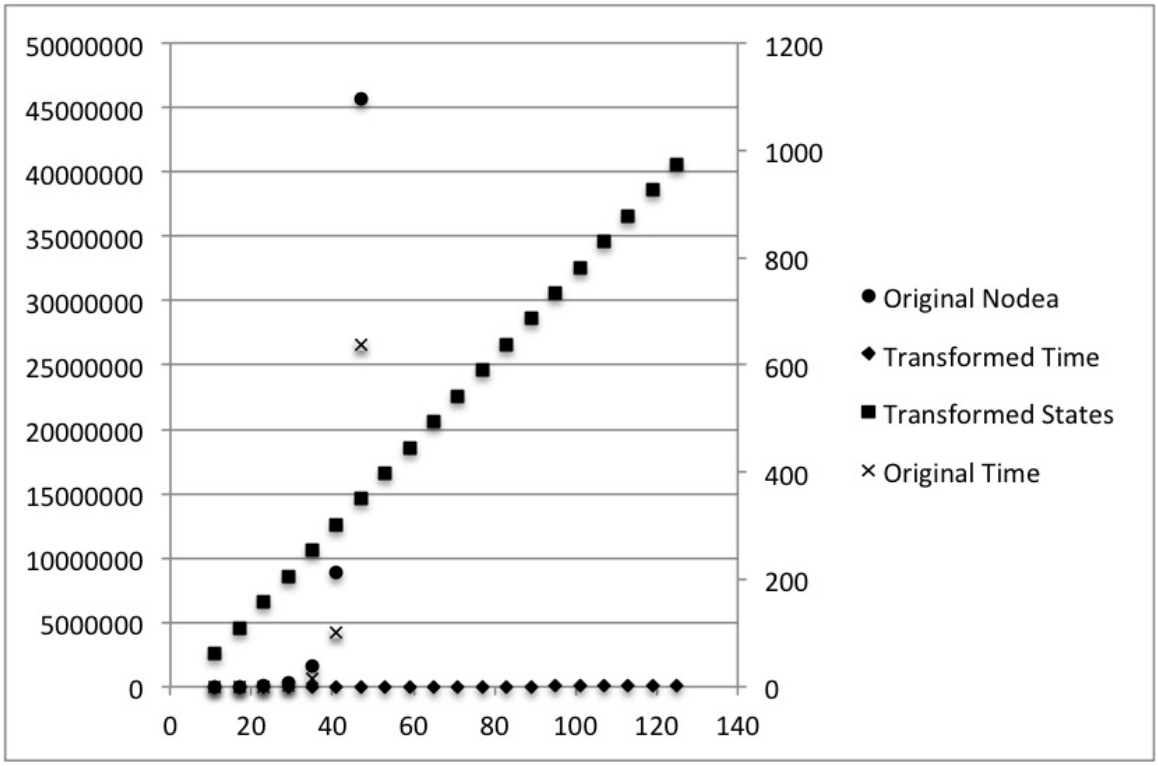

Figure 1. Time & Nodes for both Representations as function of Solution Length. Nodes in the original representation plotted on left axis, the rest plotted on right axis.

gripper; other unary predicates such as **at-robby** and **free**; and relational predicates such as **at** and **carry**. The domain file also contains the action definitions. In the GRIPPER domain the actions are **move**, **pick**, and **drop**.

Our transformation system inputs the two PDDL files (the problem and the domain). The pseudocode is shown in Table 4. First it must determine whether there are any objects that can be merged. If it determines that there are no objects which can be merged, it will just run the planner on the original representation and return that solution. Two objects can be merged if:

- They are of the same type (using either PDDL typing or as in the old GRIPPER domain using singleton predicates).
- They appear in the same predicate in the goal description and have the same additional predicate values.
- Neither object is mentioned directly in any of the actions.

In the GRIPPER domain, this means that all the balls can be merged into 1 bag of "ball1"s at each location.

Now the problem is that most problem solvers cannot handle multiple objects with the same name and make sure they all get handled "correctly" in the goal state. Most problem solvers will put

```
Transform:
 if NOT CanMerge
  then run old representation
 else
    change predicate in goal state
    change predicate in the initial state
    add predicates to initial state for
       empty locations (count X1 ?Y 0)
    add predicates to the initial state to
       handle arithmetic (more X+1, X)
    change predicate list in domain file
    change actions to refer to new predicate
    change parameters of action

CanMerge:
 objects X are not referred to in actions
   (as constant)
 objects X are all of the same type and
 objects X have the same predicates and
  additional values in the goal description
```

Table 4. Transforming PDDL Pseudocode

one ball1 into roomb and then stop, assuming the problem is solved. In SOKOBAN, we get around this problem by using variables in the PDDL goal description to guarantee all the stones are put into separate goal squares, see Table 1.

In problems like the GRIPPER domain, the balls are all going to have predicates in the goal description with identical values (not different ones like SOKOBAN). Therefore these predicates all collapse into a single predicate instance and we have no way to distinguish them. Given this, it is more difficult to get the problem solver to exhibit the behavior we want. To achieve this, you can change the predicates so that instead of saying where each ball is, they count the number of balls at each location. This is a transformation that was explored by many of the problem reformulation researchers in the 1970s and 1980s. This allows you to easily make a goal which ensures all the "ball1"s are at their final location. This has the added benefit of making a problem representation with many fewer objects and a state space with far fewer states and edges.

Of course when you change the predicates, you must also change the actions, the initial state and the goal description to work with the new predicates. Looking back at Table 4, we must now change the **at** predicates **(at ball1 roomb) (at ball2 roomb) (at ball3 roomb) (at ball4 roomb)** to a single **count** predicate **(count ball1 roomb 4)** in the goal description. We must put a similar predicate in the initial state **(count ball1 rooma 4)** as well as an additional predicate that states **(count ball1 roomb 0)**. In addition we must add arithmetic predicates, such as **(more 0 1) (more 1 2) (more 2 3) (more 3 4)**. These changes to the initial state and goal description can be seen in Table 6. Lastly we must change all the actions to refer to the new predicates instead of the old ones. For instance, the **pick** action has a precondition of **(count ?obj ?room ?num1)** which it removes

```
( define ( domain gripper-strips )
   (:predicates ( room ?r ) ( ball ?b ) ( gripper ?g ) ( at-robby ?r )
   ( count ?obj ?room ?num0 ) ( free ?g ) ( carry ?o ?g ) )

   ( :action move
     :parameters ( ?from ?to )
     :precondition ( and ( room ?from ) ( room ?to )
                         ( at-robby ?from ) )
     :effect ( and ( at-robby ?to ) ( not ( at-robby ?from ) ) ) )

   ( :action pick
     :parameters ( ?num2 ?num1 ?obj ?room ?gripper )
     :precondition ( and ( more ?num2 ?num1 ) ( count ?obj ?room ?num1 )
                         ( ball ?obj ) ( room ?room )
                         ( gripper ?gripper ) ( at-robby ?room )
                         ( free ?gripper ) )
     :effect ( and ( carry ?obj ?gripper )
                   ( not ( count ?obj ?room ?num1 ) )
                   ( count ?obj ?room ?num2 )
                   ( not ( free ?gripper ) ) ) )

   ( :action drop
     :parameters ( ?num2 ?num1 ?obj ?room ?gripper )
     :precondition ( and ( more ?num1 ?num2 ) ( count ?obj ?room ?num1 )
                         ( ball ?obj ) ( room ?room )
                         ( gripper ?gripper ) ( carry ?obj ?gripper )
                         ( at-robby ?room ) )
     :effect ( and ( not ( count ?obj ?room ?num1 ) )
                   ( count ?obj ?room ?num2 ) ( free ?gripper )
                   ( not ( carry ?obj ?gripper ) ) ) ) )
```

Table 5. Transformed GRIPPER domain operators in domain.pddl

in the effects and adds **(count ?obj ?room ?num2)** where **(more ?num2 ?num1)**. These actions are shown in Table 5.

In problems where only some of the objects map to the same goal value, the composite transformation will make multiple bags. For instance, in the GRIPPER domain, if the goal description had two balls in **roomb** and two balls in **rooma**, then two balls would be retained each having two instances, giving a goal description of **(:goal (and (count ball3 roomb 2) (count ball1 rooma 2)))**.

The last thing our system does is transform the solution for the new problem representation back into a solution for the original problem representation. The solution, generated by a domain independent planner for the transformed representation, is shown on the left side of Table 7. The solution, automatically generated by our backward transformation to the original representation, is shown on the right side of Table 7. For this transformation there is a one-to-many mapping between the transformed solution and a set of original solutions. Moreover there is guaranteed to be a solution of the same length that is still optimal. To be more precise, given a sequence of

actions that solves the problem in the transformed representation, we are guaranteed that it can be transformed into a solution in the original representation by going through a two step process. First, we remove those new action arguments which were introduced by the transformation. Second, we replace the occurrences of the new "merged object" in the actions' arguments by the appropriate original "mergee objects". There must exist a "mergee object" which can be placed as an argument in this action. For example, the solution shown on the left side of Table 7 was transformed into the solution on the right side by removing the first two arguments of the **pick** and the **drop** actions and by replacing the merged object, **ball4**, by **ball4** in the first action, by **ball3** in the second action, by **ball4** in the fourth action, by **ball3** in the fifth action, etc. To find the correct "mergee objects", a search must be performed. This is currently done with depth first search with backtracking, verifying each operator and generating the resulting state before moving on to the next operator. Although there may be many solutions in the original problem's representation which satisfy this mapping, our system finds a single set of bindings for the variables that makes a valid solution for the original representation. It returns the first solution it finds.

```
( define ( problem strips-gripper-x-1 )
    ( :domain gripper-strips )
    ( :objects 4 3 2 1 0 rooma roomb ball1 left right )
    ( :init ( room rooma ) ( room roomb ) ( ball ball1 )
            ( at-robby rooma ) ( free left ) ( free right )
            ( gripper left ) ( gripper right ) ( count ball1 rooma 4 )
            ( more 0 1 ) ( more 1 2 ) ( more 2 3 ) ( more 3 4 )
            ( count ball1 roomb 0 ) )
    ( :goal ( and ( count ball1 roomb 4 ) ) ) )
```

Table 6. Transformed GRIPPER domain problem description in prob01.pddl

7. Summary Insights

The transformation currently works on 3 other PDDL domains: TRANSPORT-OPT11, NOMYSTERY-OPT11, and ELEVATORS-OPT11. In these domains, in some problems, there is no transformation done because no two objects are moving to the same location in the goal description. When a transformation is done, the total time taken in the transformed representation is always longer with LM-cut in these 3 domains! So the question is what is different about the GRIPPER domain and these 3 other domains?

We have developed a transformation that creates a better representation any time there are substantially more objects than locations, and when a large number of the objects are going to the same location. Even though it always reduces the size of the blind search space, there are domains where it makes the problem solver run slower. We have done two cross-over experiments. In one experiment we took the GRIPPER domain and added more locations until the original representation solved the problem faster. In the other experiment we took the NOMYSTERY-OPT11 domain and added more packages until the transformed representation solved the problem faster. This shows that once the number of objects exceeds the number of locations by a "large enough" amount, the transformed representation does better. This leads us to believe a heuristic could be used to decide when to bother with transforming the problem representation.

(pick 3 4 ball4 rooma left)	(pick ball4 rooma left)
(pick 2 3 ball4 rooma right)	(pick ball3 rooma right)
(move rooma roomb)	(move rooma roomb)
(drop 1 0 ball4 roomb left)	(drop ball4 roomb left)
(drop 2 1 ball4 roomb right)	(drop ball3 roomb right)
(move roomb rooma)	(move roomb rooma)
(pick 1 2 ball4 rooma left)	(pick ball2 rooma left)
(pick 0 1 ball4 rooma right)	(pick ball1 rooma right)
(move rooma roomb)	(move rooma roomb)
(drop 3 2 ball4 roomb left)	(drop ball2 roomb left)
(drop 4 3 ball4 roomb right)	(drop ball1 roomb right)

Table 7. Solution in Transformed Space (left) and Original Space (right) for the GRIPPER domain

Taking a broader view (more general than transportation problems), the transformed representation is good for domains with a lot of objects that can only have one value at a time (a ball or package can only be in one location at a time). Whereas the original representation is good for domains with a lot of objects that can have many values at once (a location can have many balls or packages at that location at once).

A number of interesting questions remain. What effect will these new PDDL representations have on different problem solvers? Can we alter the transformed representation so that it has the same state space reduction but does not cause the heuristics to be slower and less accurate? Can we find a "good" heuristic for the transformed representation?

8. Related Research

The early work on problem reformulation was theoretical and did not include implementations. (Amarel, 1961; Amarel, 1965; Amarel, 1968; Amarel, 1971; Newell & Simon, 1972; Korf, 1980). Several researchers developed automatic creation of macro-operators (Korf, 1985; Iba, 1989; Riddle, 1991). Lowry (1989), looked into problem reformulation as a type of automatic programming. Subramanian (1989) explored the importance of relevance in problem reformulation. Nadel (1990) explored 8 representations of the 4-queens problem. Preditis (1993) automatically generated admissible heuristics by automatically generating abstractions.

The most recent work on problem reformulation is by Fink (2003), who automates some changes of representation. The constraint propagation community has done a lot of work in the area of symmetry breaking (Walsh, 2012). In addition, research by Helmert (Helmert, 2009) focuses on turning PDDL into a concise grounded representation of SAS+. Further work on transforming problem representations has been done (Haslum, 2007; Helmert, 2006), they transform between PDDL and binary decision diagrams and causal graphs respectively.

9. Conclusions and Future Work

Now is an excellent time to return to the area of problem reformulation. A problem's representation can make it either easier or harder to solve a problem. We want to make an adaptive problem solver

that can find a more tractable problem representation which it can then use to solve the problem. The transformations which provide the new problem representation should be generative.

We have a powerful framework that creates a flexible problem solver, by providing automatic transformations from intractable problem representations to easier problem representations. This allows many problem solvers to solve problems that they could not solve before, in reasonable time and space. We defined a set of component transformations, out of which higher-level composite transformations can be defined. We explored a case study of a single composite transformation, presenting encouraging results in the GRIPPER domain where previously intractable problems can now be solved in a few seconds.

We have shown that you can affect a huge savings in problem solving time by spending a little time transforming the representation. This opens a brand new approach to improving problem solving systems. With the results varying so widely from one representation to another we think there will be increased interest in this area at ICAPS and in the IPC. Since the transformations produce PDDL files, they can be used seamlessly with any problem solving system. We plan to extend this work and develop a set of transformations and a method for searching through the space of transformations to decide which representation is the best choice for a given problem.

Our future research is currently going in two directions: a problem solver that uses multiple representations and developing more reformulations. Each of these will be addressed in turn. There are at least three obvious ways to use the representations we create. The first is to use them in portfolio type system where you just run each different representation and return the one that finished first. This will work fine as long as you only have a couple representations, but once you get 4 or more representations this will probably not be a viable alternative. A second alternative is to use cross-over studies to predict when each will perform better, it is unclear how well this will work, but it is at least an option for exploration. The last possibility is to use the RA* system (Franco & Barley, 2008a; Franco & Barley, 2008b; Franco & Barley, 2009; Franco, Barley, & Riddle, 2013a; Franco, Barley, & Riddle, 2013b). This system uses sampling to determine the heuristic branching factor and the cost of applying heuristics, to determine which heuristic to use. The same technique can be used to determine which representation is more likely to solve the problem. This is especially true when the optimal solution path is the same length in both representations, it might be more difficult if the two representations produce different solution path lengths.

The second direction for future research is to create more transformations. The first approach is to do the same type of transformation in domains like SOKOBAN where the objects can never be at the same location at the same time. In these domains we do not need to make a bag of objects, we can just call all the objects S1, but we do have to take some care with the goal description in the transformed problem representation, because the predicate only takes one variable and so the objects can collapse. There are other standard reformulations we are looking at (like macro-operators). We will also look at extending the current transformation to work in cases where an object is spread over many predicates in the goal description, like WOODWORKING-OPT11.

Acknowledgements

Thanks to Pat Langley for advice on earlier drafts of this paper.

References

Amarel, S. (1961). An approach to automatic theory formation. *Principles of Self-Organization: Transactions. Pergamon Press, New York, ny*.

Amarel, S. (1965). Problem-solving procedures for efficient syntactic analysis. *ACM Twentieth National Conference*.

Amarel, S. (1968). On representations of problems of reasoning about actions. *Machine Intelligence 3*.

Amarel, S. (1971). Representations and modeling in problems of program formation. *Machine Intelligence 6*.

Fink, E. (2003). *Changes of problem representation: Theory and experiments*. Springer-Verlag.

Franco, S., & Barley, M. (2008a). In situ reconfiguration of heuristic search on a problem instance basis. *2008 AAAI Workshop on Metareasoning: Thinking about Thinking*.

Franco, S., & Barley, M. (2008b). Using sampling to dynamically reconfigure problem-solvers. *The First International Symposium on Search Techniques in Artificial Intelligence and Robotics*.

Franco, S., & Barley, M. (2009). Predicting the optimal combination of pattern databases for solving a problem. *International Symposium on Combinatorial Search*.

Franco, S., Barley, M., & Riddle, P. (2013a). In situ selection of heuristic subsets for randomization in ida* and a. *Heuristics and Search for Domain-Independent Planning* (p. 5).

Franco, S., Barley, M., & Riddle, P. (2013b). A new efficient in situ sampling model for heuristic selection in optimal search. *Proceedings of the Australasian Joint Conference on Artificial Intelligence*.

Haslum, P. (2007). Reducing accidental complexity in planning problems. *Proceedings of the International Joint Conferences on Artificial Intelligence* (pp. 1898–1903).

Hayes, J., & Simon, H. (1974). Understanding written problem instructions. *Knowledge and Cognition*.

Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, *26*, 191–246.

Helmert, M. (2008). Changes in PDDL 3.1. Unpublished summary from the IPC-2008 website. http://ipc.informatik.uni-freiburg.de/PddlExtension. Retrieved November 2013.

Helmert, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, *173*, 503–535.

Helmert, M., & Domshlak, C. (2009). Landmarks, critical paths and abstractions: What's the difference anyway? *Proceedings of the International Conference on Automated Planning and Scheduling*.

Helmert, M., Haslum, P., & Hoffmann, J. (2007). Flexible abstraction heuristics for optimal sequential planning. *Proceedings of the International Conference on Automated Planning and Scheduling* (pp. 176–183).

Helmert, M., & Röger, G. (2008). How good is almost perfect?. *Proceedings of the Association for the Advancement of Artificial Intelligence* (pp. 944–949).

Holte, R. (1988). *An analytical framework for learning systems*. Doctoral dissertation, University of Texas at Austin.

Iba, G. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Korf, R. (1980). Toward a model of representation changes. *Artificial Intelligence*, *14*, 41–78.

Korf, R. E. (1985). Macro-operators: A weak method for learning. *Artificial intelligence*, *26*, 35–77.

Lowry, M. R. (1989). *Algorithm synthesis through problem reformulation.* Doctoral dissertation, Stanford University.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998). Pddl-the planning domain definition language.

Minton, S. (1988). *Learning search control knowledge: An explanation-based approach*, Vol. 61. Springer.

Nadel, B. (1990). Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, *5*, 16–23.

Newell, A. (1966). On the representations of problems. *Computer Science Research Reviews. Carnegie Institute of Technology, Pittsburgh, PA*.

Newell, A., & Simon, H. A. (1972). *Human problem solving*, Vol. 14. Prentice-Hall Englewood Cliffs, NJ.

Ohlsson, S. (1992). Information-processing explanations of insight and related phenomena. *Advances in the psychology of thinking*, 1–44.

Pólya, G. (1957). *How to solve it: A new aspect of mathematical method*. Princeton University Press. Second edition.

Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine learning*, *12*, 117–141.

Riddle, P. (1991). *Automatic Shifts of Problem Representation*. Doctoral dissertation, Rutgers University.

Riddle, P. J., Holte, R. C., & Barley, M. W. (2011). Does representation matter in the planning competition? *Proceedings of the Symposium on Abstraction, Reformulation, and Approximation*.

Simon, H. (1972). On reasoning about actions. In *Representation and meaning.* Englewood Cliffs, NJ: Prentice-Hall.

Subramanian, D. (1989). *A theory of justified reformulations*. Doctoral dissertation, Stanford University.

Walsh, T. (2012). Symmetry breaking constraints: Recent results. *Proceedings of the Association for the Advancement of Artificial Intelligence*.

Wikipedia (2013). Sokoban. [Online; accessed 13-November-2013].