Multi System Qualitative Planner

James Bebbington Project Report, February 2010

Table of Contents

1 Introduction 1

1.1 Qualitative Representation of Learned Behaviour 1

1.2 The objectives of and uses for the Planning Agent 1

1.3 Document Structure 2

2 Single System Planner 3

2.1 More detailed reasoning as to why the Algorithm works 3

2.2 A Problem with the Algorithm 5

2.3 Efficiency of the Updated Algorithm 6

3 Multi System Planner 8

3.1 Description of the Algorithm 8

3.1.1 Matching (and Scoring) a system 8

3.1.2 Overview of how Regression Planning is performed 11

3.1.3 Details of the Algorithm 14

3.1.3.1 The “must be true” set of assertions 14

3.1.3.2 Entry and Exit Conditions 14

3.1.3.3 Inference 15

3.1.3.4 Object types and dealing with created and deleted objects 16

3.1.3.5 Dealing with “move” actions 17

3.1.4 Time Complexity 18

3.1.5 Issues 19

3.2 Enhanced Representation of Models 20

3.3 Example Models and Plans 21

3.3.1 Boiling Water in Cup 21

3.3.2 Drying a Towel with fan on Exercise Bike 21

3.3.3 Using Shower to produce Steam 21

3.3.4 Apply *Change Tyre* system to get item out from under Car 21

3.3.5 Building an unusual Mechanism (vibrating sand in box) 21

3.3.6 Other Models 22

3.4 Discussion 22
3.4.1 Using a system even though several assertions are unmatched 22
3.4.2 Concurrency Issues 23
3.4.3 Specialised versus General Models 23
3.4.4 Specifying values as Variables 24
3.4.5 Several Other Points 24
3.4.6 Probabilities and Costs 25
3.4.7 25
4 Conclusion 26
5 Appendix 27
5.1 Qualitative Best & Worst Expression Evaluation Tables 27
5.2 Pseudo-code of the Single System Algorithm 27
5.3 Pseudo-code of the Multi System Algorithm 28

1 Introduction

This report discusses the design, implementation, and review of a planning algorithm that attempts to produce good contingent plans to achieve goals (specified as a set of assertions), given an initial world state, and using a specific type of knowledge representation. The knowledge is represented as a collection of system models, which are able to be generated by a learning agent using actual observations in a simulated 3D world (as developed in Adam Clarke's PhD). These models capture the state changes of a small set of objects given a number of possible actions on some of these objects. The knowledge is represented using finite state automata, first-order logic, and qualitative physics.

1.1 Qualitative Representation of Learned Behaviour

Description of systems (can be taken directly from Honours report).

Aside: Using the planning algorithm on alternative representations.

Note that in this report we do not focus specifically on the problem of how to best represent knowledge in a particular system (what specific assertions should be considered, etc.). Various papers [Davis08 ("Pouring Liquids"), etc] provide detailed specifications of this. Instead we focus on planning; our planner should be capable of constructing short plans that are as detailed as the systems it is planning in are – note that the maximum amount that can be achieve by a plan that can be produced in a feasible time frame will be dependent on how detailed the systems are. How detailed systems are will depend on how careful the observations were. Detailed observations will allow very specific plans. For example, a plan might be as low level as "tilt the pitcher so that it is beyond the point where the height of the water inside is over the edge of the spout, will cause the water to flow out of the pitcher into whatever is below it", or as high level as "pour the water out of the pitcher".

We do, however, require that systems use a representation at least somewhat similar to that described above: assertions must be described using first order logic (additionally qualitative—potentially partially quantitative—reasoning is imperative to restrict the number of states that need to be evaluated) and knowledge must be encapsulated in modular finite state machines (this is key to restricting the amount of places the planning algorithm will need to look in order to come up with plans).

1.2 The objectives of and uses for the Planning Agent

Single system objectives (can be taken directly from Honours report).

The second task of the planning agent is to construct plans that go through several systems. Again, the initial (current) and goal assertions will be specified by a set of assertions. These assertions may, however, span multiple systems. Not only will the planning algorithm have to find a plan that achieves each of the goal assertions, but it will also take generalised models that have assertions specified in terms of variables which it will need to bind to the specific objects mentioned in the goal (and initial) assertions.

Producing plans that span multiple systems within a pool of knowledge that may contain a very large number of systems is an extremely difficult task. Realising this, we only aim to produce reasonably short plans that involve between one and around six systems, and we will only try to find one plan (any plan that achieves the goal assertions, not necessarily the

shortest one). Note that these goals align with what we feel are the potential uses for the planning algorithm. We imagine the planning algorithm would only ever need to be used to fill in the gaps of a hierarchical planner (as it is always preferable to follow rules learned via a hierarchical planner when available). This may mean performing unusual tasks, or performing tasks in an unusual way (such as using the sink in the bathroom to clean dishes if the sink in the kitchen becomes blocked). As such, when evaluating the algorithm, we attempted to consider several examples of unusual tasks that people are not likely to have learned standard rules for. Note that after a plan is produced and executed, it should be converted into a new decomposition rule and incorporated into a hierarchical planner. Section 3.4 presents some ideas on how this could be done.

The algorithm we have come up with is an amalgamation of several standard planning techniques implemented with the specific kind of knowledge representation in mind, with the addition of some novel aspects.

1.3 Document Structure

Section two discusses the design, implementation, and review of the single system planning algorithm. Section three discusses the design, implementation, and review of the multi system planning algorithm. Section four provides some concluding remarks, and suggests possible future work that could be taken to extend the planner's functionality.

2 Single System Planner

See section 4.3 of the honours report. This section describes a problem with the original algorithm, and the changes that were made to solve this problem, as well as giving additional details as to why the algorithm works (also see the new pseudo code in the appendix).

2.1 More detailed reasoning as to why the Algorithm works

First, we give some reasoning as to why the original algorithm works (this is unrelated to the problem that we describe in the following section).

The (original) algorithm labels each node in the graph with an expression in terms of the values of other nodes that it has transitions pointing to. We then traverse through the graph.

Figure 1 shows the way in which a graph is traversed. We follow a particular kind of path (shown in red) through the graph. This path will go from the initial (starting) node, ending at a node that only has transitions that lead to either a goal state, or back to nodes that have already been visited. The path will not necessarily be just a straight line. There may be any number of branches that end at a node that only has transitions that lead to either a goal state, or back to nodes that have already been visited (also note, that there can be branches off of branches, etc.). In Figure 1, the red lines going from node 4 to node 6, and from node 12 to node 15 show such sub-branches (similarly, the straight line from node 1 to node 17 is a branch—the main branch). A key point to note is that *all* of the nodes in a given branch can only point to a goal state or nodes that have already been visited.

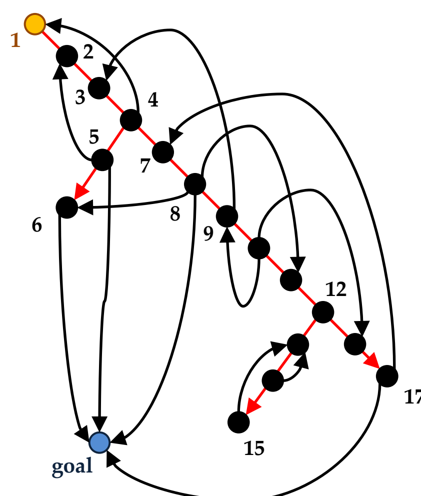

Figure 1: Traversing a Graph

been visited? Any nodes that occur earlier on the same branch (nodes 1 to 4, and 7 to 8), and any nodes that occur on sub-branches that have been visited before the branch that this node is on (in Figure 1, sub-branch 4 to 6 must have been visited before the branch beginning at node 7, as node 8 has a transition to node 6, meaning nodes 5 and 6 have been visited before node 8—the traversal could just as easily have been performed the other way, in which the traversal could look like the one shown in Figure 2).

This allows us to draw the conclusion that any visited node must either:

a) *Have a grounded value*; or
b) *Have a path that leads back to the node that is currently being evaluated*.

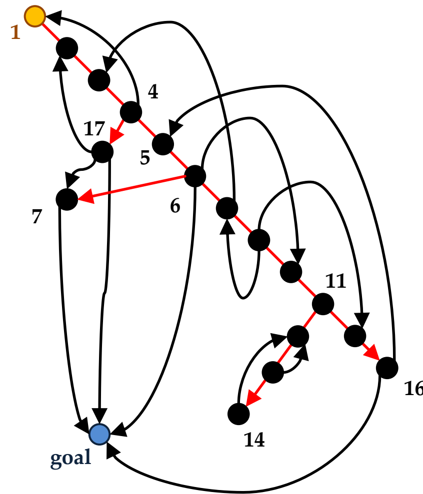

Figure 2: An Alternative Traversal

If, on the first branch visited after a split point (such as the branch 4 to 6, which splits at node 4 in Figure 1), there is a sequence of nodes that have no transitions leading back to nodes before the starting point of the branch or to earlier nodes in the same branch that have such a transition themselves (such as node 6 on its own), then these nodes must all achieve grounded values by the time we move on from the branch and begin evaluating other nodes. This is because the value of these nodes will only depend on each other, and goal states (we already inferred above that there cannot be any transitions leading from any nodes on a branch to nodes on a later branch). Recall that once we reach the end of a branch we will begin backtracking and updating the values of nodes. This corresponds to (a) above.

If, on the other hand, there is a transition from a node on a given branch that leads back to a node that is on a part of the path that occurs before the branch off point (such as the transition from node 5 to node 2), then this node and any earlier nodes on the branch will not have grounded values, at least not until the earlier node is evaluated. But this means that any nodes on branches that begin after this earlier node (node 2) can be reached by all nodes in this branch before the node that had the transition pointing back to the earlier node (node 5). This corresponds to (b) above.

Finally, if there is a transition from a node on a given branch (we now consider the given branch, as the branch 7 to 17 in Figure 1) that leads back to a node that is on an earlier branch (such as the transition from node 8 to node 6), then since the value of the node that this transition points to (node 6) is either grounded or has a path that leads back a node before the branching point, it does not change the fact that the value of the node that contains this transition (node 8) will have a value that is either grounded or leads back to a node before the branching point (node 6 will have a grounded value by the time node 8 is being evaluated, so this will be able to be substituted directly—node 8 itself will not actually be immediately grounded itself, due to the transition from node 9, which comes after node 8, to node 3).

The same logic can be applied to branches off of branches, etc. (see Figure 3). In this example, node 12 has a transition leading to node 5 initially preventing nodes 6, 9, and 10 from being updated. Once we being updating and eventually backtrack to node 5, however, all the values will be able to be updated (as no nodes in any branch beyond this point have a

transition pointing back to an earlier node), and so when we go on to evaluate the final branch which includes node 14, the transition from node 14 to node 12 will be updated with grounded values.

Why was it important to prove this? Because our algorithm makes the important assumption that if a node ever links back to an earlier node that does not have a grounded value, then we can insert the *at-least-loop* value. This is the key as to why the technique of substituting in *at-least-loop* values works.

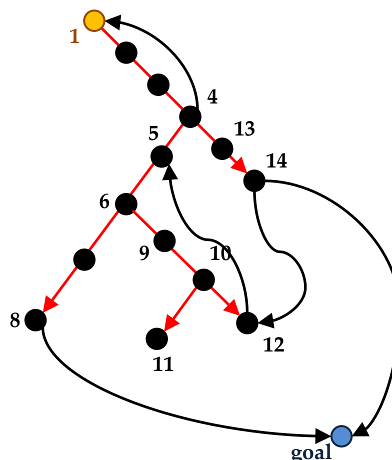

Figure 3: Branches off of Branches

2.2 A Problem with the Algorithm

We now move on to explaining the problem with the original algorithm. Note that the problem had nothing to do with the reasoning in section 2.1. The problem was instead to do with the way node values were being updated. Consider Figure 4 below. Since node 3 has a transition leading back to node 1, neither node 3 nor node 2 can obtain a grounded value until node 1 is evaluated (this is true for node 2 because it has a transition leading to node 3).

Previously, once node 3 was reached, we would substitute in the value *at-least-loop* (as node 3 only loops back to earlier nodes). This would give nodes 1 and 2 a temporary value for node 3 that could be used to evaluate their values (node 3 would itself be updated again later once node 1 and 2 have grounded values). The problem was that after backtracking to node 2, we discover some information about node 3 that node 1 needs to use to update its value properly. Thus, we cannot fully evaluate node 2 or 3 (because node 3 has a transition leading back to node 1)—but we can partially evaluate them. We would have automatically partially evaluated node 2 as part of the backtracking process (substitute in *at-least-loop* for node 3), but we ignored the fact that node 3 should be partially evaluated based on node 2 as well. When evaluating node 1 it must use the updated value of node 3 (that is, the value of node 3 updated with the value of node 2)!

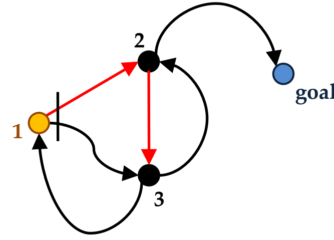

Figure 4: A Problem with the Algorithm

value (either by evaluating itself for the first time as is the case for node 2 in Figure 4, or as a result of receiving an updated value from another node), that it updates all nodes that occur later in the current path (as well as then passing its value back to earlier nodes).

To achieve this, we have altered the algorithm substantially. All nodes now maintain a "current value", as well as the "expression" of how this value was derived. Every time a node receives an updated value from another node, it looks at this expression and substitutes in the new value accordingly. The value produced from this expression is the new current value for this node. If the node's current value *improves*, then all other nodes[1] are updated (from the last node visited to the first node visited) with the improved value (and so on).

For example, consider the graph shown in Figure 1Figure 4. Node 3 will begin with the expression `value(3) = best(node1,node2)+1`. When we backtrack to node 1, node 3 will be evaluated first. Both variables will be substituted for the value *at-least-loop* (as both node 1 and node 2 will have already been visited—refer to the original algorithm or pseudo code for more details):

`value(3) = best(at-least-loop{node1},at-least-loop{node2})+1 =` *at-least-loop*

Now that node 3 has an updated current value (*at-least-loop*), it will initiate a cascading update of all the other nodes, starting with node 2. Evaluating node 2's expression, which is `value(2) = best(0,node3)+1`, will give a current value of *no-loop,1*. Now that node 2 has an updated, and grounded, value this new value will in turn result in updates to node 3 and node 1. When updating node 3 a second time, we again look at its original representation:

`value(3) = best(at-least-loop{node1},no-loop,1{node2})+1 =` *no-loop,2*

As this is a new value for node 3 (improves upon the old value of *at-least-loop*), more updates will be initiated. Note that we ignored updates to node 1 in the above example, but it should be clear to see that node 1 will end receiving the correct values from both node 2 and node 3.

2.3 Efficiency of the Updated Algorithm

It may at first seem like the updating process could go on forever, although it is fairly easy to realise that the structure of the graphs will prevent this from being the case. In order to determine the efficiency of the algorithm, however, we need to work the maximum number

---

[1] In fact, we only update nodes which have been traversed (on the path so far). This will not include nodes that are to be traversed on later branches. Because of the reasons discussed in section 2.1, we know this is not an issue.

of updates may be carried out for any given graph. We show below that the updating part of the algorithm is guaranteed to terminate and that it will do so in polynomial time efficiency.

Before we begin, note that in section 2.2 we stated that updates are only carried out when a node's value is updated and *improved*. By determining when it is not possible for a series of updates to result in a nodes value improving we can infer when a cascading update must terminate.

The key insight is that if node x updates node y and causes an improvement, then no later updates that were a result of updating any other node z with the value of node y can result in an improved value for node x (or node y). This is because all improvements to values as a result of the update must arise due to paths going through x. If x were to receive an update from z, then there must be a loop ($x \rightarrow z \rightarrow \ldots \rightarrow y \rightarrow x$), and the value of x cannot improve by following a loop back to itself. The exception is in when we are updating values for the first time (rather than improving values) as in the example given in the previous section, but subsequent updates must still abide by this rule.

Consider a node, node x, that has just received an updated value. Now any other nodes in the graph that have a transition going to this node can potentially be updated and may be *improved* with the new value—that is a maximum of $n-1$ updates. Each of these nodes could potentially update all the rest of the nodes ($n-2$ updates). Using the insight above, however, we can see that if node y updates node z, then node z cannot update node y or node x. This limits the number of possible updates to $n-1 + n-2 + \ldots + 1 = \frac{n(n-1)}{2}$; node y can improve all other nodes other than node x, but then node z can only improve all the other nodes except node x and node y, and so on—note that the second node to apply its updates can improve some of the same nodes that the first node improved if it improves them by a larger amount (if, for example, the transition going from the second node to node x is part of a non deterministic action, where the path cost down one of the other transitions is worse than $value(x)+1$). This is taken account of in the equation above, but is likely to be a quite rare occurrence in actual system models.

There are also additional redundant updates that may be performed. Node b could update node c, then receive a better value from node a, and so have to update node c again. In the worst case, one update could be redundantly performed $n-1$ times, two updates could be redundantly performed $n-2$ times, etc. This would result in the potential maximum number of updates being of order n^3. It is unlikely, however, that redundant updates will ever occur this often in practice, as they will only occur when the order in which we update nodes happens to be the opposite of how they are linked.

Thus, the algorithm's overall efficiency could potentially be $O(n^4)$, as there could be up to n updates, each affecting a maximum of from 1 to n nodes, which are all reasonably expensive. In reality, the efficiency is more likely to be $O(n^3)$ in almost all (if not all) cases.

3 Multi System Planner

This section begins with a description of the multi system planning algorithm that was developed. It then reiterates the features added to models for use by this algorithm (most of these features are first mentioned in the description of the algorithm where appropriate). Finally, a collection of example systems and plans are provided to demonstrate the ability of the algorithm.

Note that the term *model* is used to describe the structure of a *system* (our modular graphical representation of knowledge). These words are often used interchangeably.

3.1 Description of the Algorithm

To begin, the algorithm is given a set of assertions that it needs to ensure are true. Additionally, a set of assertions that are true in the current world state are given (if used in practice, these assertions would not need to be specified, but instead could be directly observed from the world, when required).

The algorithm, after first removing any goal assertions that are true in the current state, will start by matching goal assertions to assertions found in models.

3.1.1 Matching (and Scoring) a system

Matching describes the process of mapping variables in a system to specific objects (where these objects are specified in the goal assertions, are observable in the current state of the world, or are temporary objects that will be created at some point in the plan). The matching process is difficult because there are an exponentially large number of ways that we could look at mapping variables to objects, and we get more information on how good our matching is as we move from looking at a single state to an initial and goal state pair. We always start off by matching a set of goal assertions to a system, which significantly limits the number of possible matchings, but leaves enough possibilities to prevent an exhaustive search. This has led us to develop a tiered binding process—we only keep the best looking bindings at each of several steps (where more information becomes available)—which is described below.

Also note that there are some interesting properties when finding matchings in our systems that we have tried to take advantage of in the design of the algorithm. For example, where one state in a model matches a goal assertion, it’s likely that there will be several other states that also match this goal assertion. As such, we perform most of the matching work only once per model.

Note on notation: where we refer to binding below, we are referring to the mapping of one variable to an object. Where we refer to matching, we are referring to the mapping of several variables to corresponding objects (a set of bindings that map different variables to objects).

The first step is to find any states that contain an assertion that can potentially match at least one of the goal assertions. An assertion can potentially match another assertion if it has the same type, name, and value, and for each corresponding object, either one or both of the assertions contain a variable (or both contain the exact same object). A hash function would be required to find states with possible matching assertions in an efficient manner.

Aside: Using a Hash Function

A hash function that is capable of taking us directly to potentially matching assertions should not be particularly difficult to implement. We would simply need to maintain a map of hashed assertion signatures (an assertion excluding the variable / object) to sets of states that contain an assertion with the corresponding signature. As the total amount of knowledge (number of systems) gets very large, this data structure would increase in size proportionally, but would also become more important. More advanced techniques, perhaps similar to those used by web search engine indexes, may be needed to determine which states to return when the number of results is large.

Note that in the version of the algorithm that has been implemented, we have not produced such a hash function. Instead, we pre-produce a list of all possible assertions in each system, and examine the list for each system in the world to find matching assertions. Once a potentially matching system is found, we then examine all the states in the system to determine which states could match at least one goal assertion.

When an assertion is found that potentially matches a goal assertion, a corresponding binding (or multiple bindings for a relational assertion) will be inferred—that is, a mapping of one or more variables to objects such that the matching assertion will become the same as the goal assertion. With this matching as a base, we then need to go on and match all the remaining variables specified in a system to objects that exist in the current world observations (the exception being for objects that are created in the model, or for objects that we believe may be created at an earlier step in the plan, since the planning algorithm constructs plans backwards).

Before moving on, it should be noted that we divide assertions into changeable and unchangeable assertions. Assertions that are unchangeable (such as assertions that describe what an object is—"this object is a pen") will only show up in the context of a model, and these *must* be matched. It is *desirable* to match as many changeable assertions as possible, but those that cannot be matched can be altered (or added if an object is created) in earlier steps in the plan (note that all goal assertions should be changeable). Additionally, some changeable assertions may not be true in all states of the model. If a changeable assertion is not true in the initial state we end up selecting to begin this model from, it would not end up being necessary to have matched this assertion at all. When learning systems, it would be possible, and not too difficult, to maintain a list of which assertions are unchangeable. This system could be enhanced further by labelling assertions with how "difficult" they are likely to be to change, based on previously cached plans where changing the given assertion was the goal.

For each model that contains an assertion that potentially matches a goal assertion, we build lists of assertions (made up of assertions found in states, or in the context) that refer to each variable mentioned in the system. From these lists, we can generate a set of possible objects (from our current observations) that could match all of the unchangeable assertions (or at least one changeable assertion if there are no unchangeable assertions) in each list. Considering all possible combinations of objects that could be matched to each variable is too expensive, so at this point we generate a *score* for each possible object, per variable.

The score generated for an object represents its closeness to the list of assertions that are true of the variable in question. The less assertions that are untrue in the current state of the world, the less work the planner will have to do at earlier steps in the plan to make them true. Assertions that don't match a current observation, but instead match one of the other goal assertions (this can often be true of context assertions) are also valued. The score is currently inversely proportional to the number of assertions that are untrue—that is, assertions that do not match a current observation or goal assertion (where an assertion has several different values across different states in the model, we only enforce that an assertion that can match it must match one of these values—we are lenient here because we have not yet worked out what our initial state will be). After scoring each potential object for a given variable, we keep only the best couple. Additionally, we add the possibility that a temporary object could match the variable—that is an object that will be created either in this system or at an earlier step in the plan.

Next, we look at each of the different matchings that were inferred earlier from matching goal assertions in this model, as well as each combination of matchings that were inferred from matching goal assertions that belong to the same state. Each of these partial matchings (partial because they don't bind every variable in the system) will satisfy at least one goal assertion. For each of these partial matchings then, we follow the process described below. (Note that applying this process to large models with many objects can result in many different possible matchings being produced. In such cases, a limited number of the largest matchings—those that contain the most variables—should be used.)

For each variable that does not have a corresponding object in the partial matching, we consider a combination of this matching with each possible combination of bindings for each of the remaining variables (these combinations are constructed from the lists of objects that can match each variable, that we described two paragraphs earlier). Finally, once we have a set of possible full matchings for a model, we score each of these full bindings.

Once we have a collection of the full matchings, we first remove any duplicate bindings and remove invalid bindings (ones where context assertions conflict with other goal assertions, etc). Next, to score a given matching, we begin by finding the *best* goal states. For each of the matching goal states that were used to construct the partial matching that this full matching was constructed from, we find the one that has the most assertions satisfied in the final matching. To further distinguish between states that match the same number of assertions, we rank states that *can be reached* by more states in the model as better than states that are can be reached by fewer states. The set of states that can be reached by a given state could be pre-computed and stored at each state in each system, or can generated reasonably quickly by the algorithm as it goes—an $O(n2)$ operation. The reasoning behind using the number will become clear later in this section. We only keep the best couple of states per matching to use potential goal states.

Now that we have a set of possible goal states (and their corresponding matchings) that each match at least one of the goal assertions we started with, we can generate a more detailed score than before, and only retain the best looking states. This time, when computing a state's score, we take into account not only the number of changeable assertions in the model that aren't matched to a current observation or goal assertion, but also the number of goal

assertions that *may be* achieved by the state (note that the number of assertions that may be achieved does not include context assertions as they cannot change in the system).

Finally, we further refine the information obtained by constructing initial and goal state pairs—in order to achieve a goal, we have to start from some state in the system. For each goal state that remains, we look at each state that *can reach* the goal state as a potential initial state. For the state to be considered, at least one of the goal assertions that are achieved in the goal state must not be true in the initial state (this means the assertion will actually be achieved). Once we have a list of state pairs, we have all the information needed to consider one step of the plan. Thus we can perform a final round of scoring and retain only the number of pairs that we want to evaluate further.

When scoring an initial and goal state pair, we can now consider only the relevant assertions—that is, those that are true in the context and the initial state. Assertions that are true in other states cannot be used to inflate scores (as they may have in earlier steps). Additionally, we limit the number of state pairs that can come from the same matching. The final number of state pairs we retain will depend on several factors as described in the following section.

Aside: Improving the Heuristic Used

Heuristics are obviously extremely important when constructing a planning algorithm of this type. The better the heuristic is the more likely it will be to find a solution quickly. The heuristic we currently use could almost certainly be improved upon. Our heuristic is based upon what we expect a regression planner should aim for, and some significant, but limited testing on constructed examples. A better heuristic would examine assertions in more detail. It could potentially include consideration of typical initial and goal states in certain systems, ensure that variables are only ever matched to objects that "potentially" have a value for each of the possible assertions that describe properties of that variable (note that this has been partially implemented), and consider a wide range of other factors. This could potentially have an enormous impact on the performance of the algorithm. Note that simply using the number of matching or non-matching assertions as a heuristic, as we currently do, is not necessarily the best indicator of closeness (as several non important assertions may be less valuable than one important one).

3.1.2 Overview of how Regression Planning is performed

The planning algorithm starts from a set of goal assertions and works backwards to get to a set of states where all assertions match assertions that are true in the current world state.

Aside: Regression Planner versus Partial Order Planner

Planning in reverse is significantly easier than planning from the current world state forwards to a goal. Planning in a forward direction would be extremely difficult because not only can states from many systems match the initial current world state, but we also may have no idea where to head from a given starting point (in order to achieve our goal assertions we are likely to have to achieve other unknown assertions first). When working backwards, we are always beginning from at least a state that achieves one of the goal assertions. We then try to find systems that are closer to our initial world state, but we also look for systems that appear to be potential last steps involved in achieving the goal (recall

our heuristic that looked for systems that achieve part of a goal, while requiring other goal assertions to be true).

The justification behind using a regressive planner as opposed to a partial order planner (or other algorithms with different kinds of orderings such as Stackplan) is that we expect the goal assertions in our plans will have lots of interactions—that is, we expect that each system should solve quite a few assertions at once, as opposed to having several independent parts to a plan that could be solved in either order. Even if certain goal assertions can be solved in different orders, we would face the issue of how many assertions can be solved together. By using a regressive planning approach, we simply need to find one order that works, which is all we are trying to achieve (after achieving a task once, it would become significantly easier to work out other possible ways of achieving it if desired).

As mentioned earlier, the first step involved is to remove any goal assertions that are already true in the current world state. The algorithm then performs the matching process described above to find a list of initial and goal state pairs that achieve at least one goal assertion. Note that no planning work needs to be performed in a single system (except for a reachability analysis) to generate these pairs.

After generating this list, the algorithm selects the first pair and updates certain values: any goal assertions that are achieved by the pair (goal assertions that are true in the goal state, but not the initial state) are removed from the set of goal assertions remaining, and any context or initial state assertions that are not true in the current world state are added to the same set of goal assertions remaining.

After updating these sets, the planning algorithm simply repeats this step recursively. If we reach a point where one of the assertions in the set of remaining goal assertions can't be achieved, this branch fails and we backtrack to the last set of state pairs we had and try the next option.

With each step taken, the set of remaining goal assertions should eventually decrease until all assertions are achieved. Note, though, that there is no guarantee this process will actually come to an end. It may be that at least as many assertions continue to be added to the set of remaining goal assertions as are removed from it. If the agent does, however, have enough knowledge about the systems involved in completing a task, then a solution should be able to be found. We rely on the heuristic to guide the algorithm down paths that are likely to achieve all assertions as early as possible. Of course, it's likely the heuristic will often lead us down non-optimal paths. To take this into account, the planning algorithm performs dovetailed iterative *deepening* and *widening*.

Aside: Reasons for using dovetailed iterative deepening and widening

Iterative deepening means we evaluate paths up to a certain max *depth*, then if we do not find any paths of this length or shorter, we increase the max depth by one, and repeat. This limits the work done evaluating non-optimal paths. Iterative widening means we restrict the number of alterative solutions (*width*) evaluated to a fairly small number, then if no paths are found within these limits, we increase the width by a specified amount, and repeat. This limits the amount of backtracking required, preventing lots of work being carried out to evaluate paths that score poorly. To obtain the benefits of both approaches, dovetailed

deepening and widening involves evaluating all paths within a given depth and width, then if no paths are found, we increase both the depth and width, and repeat.

The justification for using iterative deepening is that, since we are attempting to find just one plan of a reasonably short length, we can begin by checking short paths, and then slowly increase the length of plans we check if we cannot find any plans at a given length. The justification for using iterative widening is that since we have a good heuristic to guide our search, we can begin by restricting the number of alternative solutions evaluated to a fairly small number.

Also note that for each iteration with a fixed depth and width, we reduce the number of additional paths (the width) that are examined as we go further down each path (add more systems to it). This is because we expect that as our paths get longer, and we get closer to our initial world state, the planner becomes more likely to select good systems, so we can reduce the search space at little cost.

When the set of remaining assertions eventually becomes empty, we know we have a solution to the plan. At this point, we evaluate a single system plan (from the specified initial state to the specified goal state) in each system involved. We know this plan cannot fail as the goal state is reachable from the initial state. The worst case is that the plan type will be *unsafe*, but it cannot be *none*. The final plan returned is the sequential combination of each of these single system plans. The overall cost of the plan is the sum of the costs of each of the single system plans, and the plan type is the worst type of plan out of each these single system plans.

Aside: Cleaning up unsafe and other plans

After the algorithm is completed, it may return a plan that is unsafe (or that involves loops). It is possible (and indeed, quite likely) that there are alternative plans that are not unsafe, but we are unlikely to detect these. Because we have limited our algorithm to selecting only *one goal state* per system, we never considered the fact that states that were dead ends in one system may match to states in other systems that can return us to a step in the plan (whether it be an earlier step that creates a loop or a later step) and produce a plan that is safe overall. An example of this can occur in the *pour water* system. When pouring water from one cup into another, if the agent accidentally pours too much water fills up receiving cup, when its goal was to have only some water in the cup, it will arrive at a dead end state in this model, producing an unsafe plan. To improve the plan, the agent should be able to realise that if it arrives at this state, it can move to the *tip out water* system, and tip a little water out of the cup. Note that similarly, it may be possible that in some cases plans that involve loops could be converted into non loop plans (although this is likely to be a rare occurrence).

This cleanup of plans could be implemented as a post processing step that activates after the planning algorithm has found a successful plan. The planner would then find all dead end states in each system, and attempt to find a plan from this state to any other state in another system that is in the plan.

3.1.3 Details of the Algorithm

3.1.3.1 The "must be true" set of assertions

After selecting an initial and goal state pair, any assertions that are in the context or initial state of a system and are true in the current world state are added to a set of assertions denoted the "must be true" assertions. This includes any assertions that are in the set of goal assertions the planning algorithm is given to start with, but happen to already be true in the current world state. Assertions are never removed from this set, though they will be altered (or potentially turned into explicit *not* assertions) if an assertion in the initial state of an earlier system in the plan contains an assertion with the same signature (has the same type, name, and object(s)) but a different value.

When finding possible goal states (states that match at least one goal assertion), the planning algorithm will check to make sure that none of the assertions in the "must be true" set conflict with assertions in the goal state. This prevents the planning algorithm from producing invalid plans, where it requires the value of some assertion to change, even though no action is taken by the agent to causes this change.

3.1.3.2 Entry and Exit Conditions

It will usually be the case that an action in one system will have an effect on several (often many) other systems. When determining which of a system's assertions are valid, we require a way of ensuring that actions taken in other systems do not have undesired effects in this system. For example, consider a system that represents information about what happens when a container is filled with water from a tap. If in another system an action is performed that moves the container so that it is no longer under the tap, then we will have exited out of this system. The effect will be that the water level in the container will no longer be rising (if it was beforehand).

One possible way to take into account these kinds of effects would be to add any actions we have noted that cause us to enter or exit the model directly into the model itself as transitions that lead to new states. Whenever we encounter an action that effects this model, we would simply add a new state to the model that describes the effect (in terms of assertions, or partial assertions—a partial assertion gives the new value for a value or derivative that has changed in a qualitative property, but not both) and add a transition to (or from) any relevant states to (from) this new state, with the corresponding action taken specified on the transition. This technique would, however, greatly reduce the benefit we gain by modularising our knowledge into models in the first place. Each model would potentially need to have very large numbers of exit and entry transitions and states added.

Instead, we specify entry and exit effects in terms of the context assertions that are made false or true. Recall that context assertions are those which must be true in order for the model to be valid. Thus, we can only enter (or exit) a system when at least one of these assertions becomes true (false). Whenever we encounter an action that effects this model, we note which context assertions are made true or false by the action, and specify effects (in the same way as above) against these assertions. This prevents us from having to specify actions explicitly and thus stops the models from dramatically increasing in size. Note that this approach relies on the fact that regardless of the action taken, when a certain subset of context assertions are validated (or invalidated), the same effects will take place in this

system. This should always be the case, although different actions may have different effects outside of this one system (this information will be represented in other systems).

To take account of entry and exit effects in the planning algorithm, we add several steps. Firstly, to take into account exit effects, whenever we are considering a new system to add to the beginning of the plan, we check the exit effects of that system. We begin moving along the rest of the plan that has been calculated so far (a system at a time). If a system causes an exit effect to trigger by invalidating all of the specified assertions (usually just one assertion), then from that point onwards we check to see if the "effect" part of the exit effect invalidates the plan. If it does, we cannot use this model.

Taking account of entry effects if more problematic, because we are planning in reverse. If we want to take advantage of an entry effect, the planner is able to add as goal assertions the assertions that need to become valid to cause the effect (in addition to any context assertions that aren't true as well). This is not currently implemented in our version of the algorithm. Also note that because move actions are handled in a unique way (see section 3.1.3.5), the entry effects of any move action that is used to achieve a context assertion in this model can be considered immediately.

It is also worth pointing out that there may (and almost certainly will) be many effects of actions (that are part of a plan) that occur in systems that we do not consider. Usually these effects won't interfere with our plan, but in some cases these effects may prevent the computed plan from working correctly. When executing plans, if an unexpected change in state means the plan can no longer be carried out, then a new plan will need to be constructed from that point. Alternatively, if we want the plan to be robust before performing it, we could check for conflicts by performing a symbolic execution of the plan, and considering what other systems *could be* active at each step.

Aside: Checking for plan conflicts in other systems

At each state in the plan, we have a set of assertions that are true of the world (made up of the initial assertions with any assertions that have been changed, added, or removed up until this point in the plan). We can use this set of assertions to infer other models that are active. For each state in the plan, we would need to perform a matching process to find other models whose context (and default) assertions are contained in this list—note that we would only need to look for exact matches so this would be less difficult than the matching process undertaken when discovering plans. Discovering which models might match the set would again require the use of some kind of hashing function (as there would be too many models in the world to examine them all).

Without having implemented this step, it is difficult to make a judgement as to how expensive this operation might be (clearly the cost would be proportional to the number of systems that make up the agent's total knowledge). Performing this process in full would ensure that a given plan does not fail because of effects that may occur in other systems that are running at the same time as some part of the plan.

3.1.3.3 Inference

Certain types of assertions can be inferred from others, particularly when considering relationships. For example, if object A is inside object B, and object B is inside object C, then

object A is inside object C. At a high level, inference essentially just tells us that, when some set of assertions are all true, there is some other set of additional assertions that are also true. Additionally, it can be used to infer when a set of assertions is invalid (due to conflicting assertions being present). There is a large amount of standard inference rules that the planning algorithm could perform to aid it in determining if goal assertions are true. In our implementation, we have limited inference to only basic relational properties; for example, we infer transitive assertions for inside, above, and below relations (as shown above), as well as ensuring these relations are nonsymmetric (if object A is inside object B, object B cannot be inside object A), and we also infer symmetric relations for nearby and connected (if object A is nearby to object B, then object B is nearby to object A).

Inference is performed on demand—that is, when evaluating a set of assertions, the planner will perform inference to see if any additional assertions are true or if the new set of assertions will be invalid. Since inference is performed on demand, we do not need to worry about when to remove inferred assertions (as we never always calculate and never store them).

Also note that since we are planning backwards, inference is difficult. We are only able to combine a set of assertions with the current world state assertions to see what additional assertions are inferred. It may be that to achieve a goal, we need to combine a set of assertions with other assertions that will be achieved at an earlier step in the plan. This kind of inference is currently not considered by the planning algorithm, except in the case of *move* actions that lead to a system, due to the special way in which move actions are represented (see section 3.1.3.5).

3.1.3.4 Object types and dealing with created and deleted objects

The representation of objects is important in our systems, as the systems are usually describing the relationships between several objects (and the effects of actions an agent can perform on them). A rich hierarchy of classes would aid in inference and the ability to construct concise systems—for example, we would like to be able to specify the assertion "`property is-a vehicle`" and have assertions such as "`property is-a car`" match to this without having to do any more work explicitly. Our implementation currently does not have any class information built into it, as it was not necessary to consider when developing specific example systems. If, however, there was a large amount of knowledge (lots of systems) being represented, involving many different types of objects, then a well constructed class hierarchy is likely to be essential.

A somewhat related issue arises when we consider how to handle the creation and deletion of objects. An example of a created object can be found in the fill a sink model where water is effectively *created* (as far as the agent is concerned). When an object like this is created, it can have a very large range of properties that become true (its temperature, saltiness, etc). Currently, we have to specify each of these properties explicitly in the model. In order to avoid having to specify each of these properties on each state where the object exists, we have a separate set of assertions which represent assertions that do not change in the model, other than being added when an object is created and/or removed with the object is destroyed.

Objects that are created and destroyed can be quite problematic for the planning algorithm; it can be quite difficult to score systems that involve the creation or destruction of an object because we only need to attempt to match assertions that relate to a created or deleted object to the current world state if the object exists in an initial state, and similarly we can only match goal assertions to assertions that related to a created or deleted object if the object exists in a goal state. We won't know which if this is the case until we produce initial and goal state pairs. Depending on the structure of a given system, it may only be possible to create or only destroy an object (invalidating our earlier matching that assumed the opposite). To deal with these issues, we give weights to the scoring of assertions, and set the weight of scorings that are generated from created and deleted assertions to a much lower value than the weights of other assertions (in the earlier scoring phases). Additionally, we ensure destroyed objects cannot be recreated (this prevents the planning algorithm from attempting to construct some foolish plans).

Additionally, there is the issue that a variable that exists in one system may be bound to an object that is created in an earlier system. In section 3.1.1, we described how we might bind temporary objects to systems. Even when examining an initial and goal state pair, however, there is no way to determine what properties make sense for a temporary object (can we create water that is coloured green?). With sophisticated class information (as mentioned earlier), we could look at the default (common) values for objects of a certain type and may even have more detailed information specified, such as "water is always transparent". This would greatly improve the ability of the planner.

Objects can also be created out of other objects—for example, splitting a piece of wood in half creates two smaller pieces (or you could represent this as just one new piece and the original piece is half as large as it was before). In our example systems, there is a case where water is poured from one cup to another. We treat this as though the water from the pouring cup is destroyed (if the cup empties), and the water in the receiving cup is created. When an object is created from another object like this, it (usually) inherits all of properties of the object it is created from: the water in the receiving cup will have the same temperature, and saltiness, etc. as the water in the pouring cup.

In order to avoid having to specify that all these properties carry over to the new object, we introduce a new kind of assertion that is specified in terms of objects for a whole system. We say that the water in the receiving cup *comes from* the water in the pouring cup. This means the water in the receiving cup receives all the properties of the water in the receiving cup, except for any that may be explicitly overridden in the assertions that are present in states in the model (a process somewhat similar to cloning). When matching objects to system variables, we can carry over any properties referring to a given object to the object that it is created from. So if our goal is to have boiling water in a particular cup, then we can achieve this by pouring water into it from another cup, and our new goal is for the water in that pouring cup to be boiling.

3.1.3.5 Dealing with "move" actions

The action of moving an object is special for a number of reasons. *Move* actions have a starting location and a final location. Such actions don't usually represent the change in relationship between objects (whereas almost all actions will generally represent changes in the properties of or relationships between objects). Instead, move actions are a way of

getting an object to where it can be used for a given task—in terms of our representation, they are essentially just used for transitioning between systems. Because of this, we treat move actions differently from other actions.

When an assertion is found in the context or initial state of a system that is not true in the current world state, we check to see if the assertion can be achieved by a move action. An assertion will be achievable via move if it is a relation type assertion that specifies a particular kind of relation (such as *above*, or *below*)—the specifics of what relations can be achieved by move are configurable; in our system we allow above, below, inside, and nearby to be achievable via move actions, but a more complex knowledge representation may specify locations explicitly and provides more details about what objects will fit in where, etc. Additionally, we require that objects be specified as *movable* objects. To determine if an object is movable, we first check the current world state to see if the object exists and is specified as being movable. If the object does not exist, we can still attempt to achieve the assertion through a move operation, but require that when the object is created, it is a movable object (see the previous section for more details on this). If the object does exist, and is not specified as being movable, then we do not allow the assertion to be achieved via a move action. Additionally, the assertion describing the object's current location (if it does exist) must not be in the "must be true" list or else we cannot solve this assertion via a move action.

After determining that an assertion can be achieved via a move action, we prefix the action to the start of the plan (before the plan that will be generated in the single system that the move action achieves an assertion for). Additionally, we annotate the assertion with constraints specifying where the object is being moved to. If we want to ensure that objects are only moved when they are able to be moved, we should add assertions such as "`property not-obstructed object`" and "`property not-obstructed location`" to the "must be true" list of assertions.

Note that we are only able to treat move in this way because it does not matter where we are moving an object from (as long as it is not obstructed). Additionally, path finding is a fairly low level task (as compared with the kinds of reasoning we tend to see in our systems), and it makes sense that this kind of task would can be carried out via other planning methods. Additionally, it is worth noting that there can be a somewhat grey line when determining what actions should constitute as moves and which should not. Almost every action we perform involves moving something (whether that is our own fingers, or a box we are holding). The key insight given above was that moves do not really represent a relationship change between two objects, and that it does not matter where an object is moved from. For most actions, such as pouring water from one cup to another (which should not be considered as a move), there is a clear relational property between the objects involved in the action (one cup is above another).

3.1.4 Time Complexity

Clearly, the algorithm is exponential in terms of the time complexity it would take to perform an exhaustive search. There is no possible way to avoid the fact that systems can be bound to objects in a multitude of ways, and that plans can be found by combining systems can be together in an exponential number of possible orderings. By using heurstics that aim to achieve as many goal assertions as possible, while minimising the number of assertions

that will need to satisfied (earlier in the plan), we hope that we will be able to arrive at a solution without having to explore too many paths that do not lead to a solution. Because our goal was only to construct short plans that solve one conjunctive set of achievements at a time, this does appear to be feasible.

Recall, however, that no single system planning needs to be performed (other than a reachability analysis) until we have discovered a multiple system plan that will definitely work. This significantly reduces the potential time taken by the algorithm.

Aside: Lots of knowledge (systems) leads to many possible solutions

It is worth noting that our experimenting with the system has so far only involved examples that we have constructed by hand. Creating examples by hand allows us to ensure that assertions are consistent across systems (ie: describing similar things at the same level of detail). If we used systems that were constructed automatically by a learning agent, the systems may not be as consistent. We believe, however, that with enough knowledge (lots of systems), it is likely that valid matches should be able to be found for any set of goal assertions and scored highly. If there are lots of systems available, then there are likely to be many possible ways to achieve goals, and so at least some systems that are on a valid path should score significantly higher than any invalid ones (as long as our matching heuristic is reasonably good).

The current implementation of the algorithm does not incorporate a hashing function to speed up the time it takes to match goal assertions to assertions in systems. This is not too much of an issue as we have currently only been able to test it on reasonably small sets of models at a time (it takes quite a while to construct systems, and the agent which learns systems in a similar format is still under construction), so performing an exhaustive search of all models doesn't slow things down much. To perform a complete search that fails to a depth of six (a plan that goes through six systems not including move actions), with a maximum top level width of eight state pairs, on the example sets of systems provided in section 3.3, takes anywhere from a minute to fifteen plus minutes depending on the size of the systems being examined and number of objects in each system.

3.1.5 Issues

There are a couple of minor issues with the current planning algorithm, which may or may not be avoidable. Firstly, it is common for the planner to select the *wrong* initial state. This is best explained through an example—if our goal is to obtain *boiling* water, the planning algorithm might match to a goal state with *boiling* water and select as an initial state in that model a state where the temperature of the water is *between room temperature and boiling*, rather than at *room temperature*. This problem is mitigated by the fact that we select several potential initial states per system (and if they both score highly, they can both be evaluated).

This problem is almost impossible to prevent—there is no way the planner can know if it should start with water that is at *room temperature* or *between room temperature and boiling*. First note though, that if the water that was to be boiled already existed in the initial world state, our heuristic would give a better score to the initial state that specifies the temperature that matches the current temperature of this water. A possible solution to the above example, then, is to specify default created object values, as discussed in section 3.1.3.4. This

is somewhat analogous to caching the plan where we create the default water (pouring water from a tap into a cup)—this process was discussed in the introduction—and may or may not provide any additional benefits. Note that where we are evaluating an assertion that the specified object does not have at all in the initial world state (which may be a property or a relation), a similar problem (and possible solution) arises.

Another minor problem arises because we are only able to select a *single* goal state per model—allowing multiple goal states to be selected per model produces the possibility of plans branching to multiple systems, which we believed would decrease the efficiency of the planner too much. Selecting only one goal state can often mean doing extra work within a single system that is really needed. For example, if we are trying to fill a cup to a specific level, then if we fill it too high, we must tip some out, and if we tip too much out we must fill it back up a bit more. It may be that we did not actually need to fill the cup to a specific level (the next higher level may have been fine as well), as the next step in the plan involves, for example, doing something to the water and then pouring into another container.

While this problem causes somewhat inflexible plans to be produced, it should be able to be mostly solved via a post processing step that goes back to check systems in the plan that outputted an unsafe or looping plan to see if they can be improved, as described in an aside in section 3.1.2.

Finally, another issue is that pure regression planning prevents the heuristic from being able to incorporate the use of entry transitions and inference from earlier steps in the plan. This is a trade-off that seems acceptable. It would be interesting, however, to look at developing a forwards or partial order planner to draw comparisons from.

3.2 Enhanced Representation of Models

Initially, we used the same models as in the single system planner. It soon became apparent, however, that this representation would not be sufficient without several enhancements. Most of the changes made have been mentioned above in section 3.1, but we reiterate these here for clarity.

- **Entry and exit effects** (see section 3.1.3.2)**:** The specification of the effects (in terms of assertions that exist in the given system) of validating or invalidating a context assertion (or a set of context assertions).
- **Created and destroyed object context assertions** (3.1.3.4)**:** A set of assertions that are always true of objects in this system once they are created, or before they are destroyed.
- **Comes from object relations** (see section 3.1.3.4)**:** A set of assertions that describe any objects that come from—that is, are produced out of and hence inherit the properties of—another object.
- **Ranges of values** (see section 3.4.4)**:** We allow the value of property assertions to be specified as a range of qualitative landmarks and ranges (rather than just one). This means the actual value for this property may be any one of the landmarks or ranges specified. Similarly, we allow a range of possible relations to be specified in relational assertions.
- **Explicit negation** (discussed below)**:** Any assertion that is mentioned in one (or more) state(s) in a system must be mentioned in all other states in that same system. This often

requires the use of *not* assertions. If, for example, the assertion "`related connected objectA objectB`" is specified in one state, then the assertion "`not related connected objectA objectB`" must be added to all other states in the system where the original assertion is not true. Note that explicit negation does not add to the representation of systems, but simply makes it substantially easier to plan with systems (as we do not have to consider whether or not assertions exist). Also note that explicit negation can be added to systems as a post-learning / pre-planning step (a learning agent would not need to ensure it adds explicit negation to learned systems). Note that *not* assertions do not need to be explicitly matched—they are simply assumed to be valid as long as they do not conflict with any other assertions.

3.3 Example Models and Plans

We have produced a range of example sets of systems, and possible goal assertions (as well as providing a set of initial assertions) with which to use to test to the planning algorithm. In this section, we provide a description of most of these sets of systems and the corresponding plans that were produced.

3.3.1 Boiling Water in Cup

Plug In Device (Kettle) -> *Fill Container* (Kettle) -> *Boil Kettle* -> *Pour Water* (Kettle to Cup)

or: *Fill Container* (Cup) -> *Microwave Fluid* (in Cup)

3.3.2 Drying a Towel with fan on Exercise Bike

Exercise Bike -> *Dry Towel*

Issue: The planning algorithm does not initially match the object that will be used to produce airflow to the exercise bike.

Solution: Improve the heuristic to ensure that variables are matched to objects that "potentially" have a value for each of the possible assertions that describe that variable (in this case, airflow out of), as mentioned in the aside in section 3.1.1.

3.3.3 Using Shower to produce Steam

Shower -> *Use Steam*

3.3.4 Apply *Change Tyre* system to get item out from under Car

Change Tyre

Issue: We are forced to specify that a tyre is punctured in order to get planning algorithm to match to the correct system.

Solution: Currently, we are just specifying that a tyre is punctured. In order to use this system without doing so, we would need to implement the ability to use a system even though some assertions are not matched. This is discussed in detail in section 3.4.1.

3.3.5 Building an unusual Mechanism (vibrating sand in box)

Eccentric Cam -> *Sand in Container*

3.3.6 Other Models

- **Heating sand in Oven:** Similarly to the Change Tyre system, we use the Oven system to perform an unusual action. Normally we would put food in the oven, but in this case we attempt to use it to heat sand.
- **Mow Lawns and Plant Flowers systems:** As a test, we constructed a pair of systems that were entirely independent—one for mowing lawns, and another for planting flowers. The planning algorithm performed surprisingly well, and was able to construct plans of reasonable length (four plus systems).
- **Driving and navigating Car systems:** We also constructed systems that represented a car driving around streets, and entered assertions representing a map into the initial assertions. The planner performed, as expected, quite poorly. This is because in order to solve a path planning type problem we would need to use a better heuristic involving distance. This is a good example of what our planning algorithm is not attempting to achieve.

3.4 Discussion

In this section, we will provide a discussion of several additional issues that have an effect on the planning algorithm, and mention ways in which any problems could be addressed.

3.4.1 Using a system even though several assertions are unmatched

In some scenarios, given a set of goal assertions and initial world assertions, we should match to a particular system, even though several of the context assertions are not satisfied. It may be that it would be otherwise impossible to satisfy these assertions, or that they could be achieved with more effort (by the planning algorithm or even by hierarchical rules). Regardless, if we discover a situation where there is a good chance a system will work as expected in spite of unmatched assertions, it is a good idea to attempt to use the system. When executing the plan, its success or failure can be reported back to the learning agent to refine the system information.

In order to work well, the ideal approach would be to have weights on each context assertion stating how "important" they are believed to be. When learning, if a context assertion is present in 95% of the cases in which the system was in use, it would be considered very important, whereas if another context assertion was only present 70% of the time, it would be given a smaller weighting. The planning algorithm could then ignore matching assertions with smaller weightings if desired. An alternative that could be implemented in the absence of any weighting information would be to first attempt to find a plan looking for full matchings in all systems. If no short plans can be found, then try again, but allow one (or possibly more) assertions to not be matched.

Also note that there are two different cases where we may want to use a system even though several assertions are not matched:

- **Removing the assertion(s) has no effect:** This case will occur far more often. It occurs when there are one or more assertions in the context that are not needed for the system to run properly. For example, we may have a *Fill the Sink* system that has a context assertion stating "`property colour tap red`". This assertion is clearly irrelevant to the running of the system, and so we could safely ignore it (leave it unmatched).

- **Removing the assertion(s) means we also remove all related assertions in the model:** This case will occur when we want to use a system but a part of the system is irrelevant—that is, we only need to use a certain parts of the system (it may highlight a case where we would be better off having two separate systems). It may be that detecting such systems would be better done during the learning process. For example, if we again have a *Fill a Sink* system, and want to use this system to fill a container from the tap (for this to be the case, we must not have a *Fill Container* and/or *Turn Tap* system), we would be able to ignore (leave unmatched) any context assertions referring to the sink: "`exists sink`", "`related under tapSpout sink`", etc. At each state in the model, we would also need to remove any assertions that relate to the sink, such as "`qproperty waterLevel sink ...`". This would leave only assertions relating to the tap and the water flow out of it. The effects of actions would remain unchanged so the system could still be used.

3.4.2 Concurrency Issues

We mentioned earlier that it is quite possible, in fact it will almost always be the case, that an agent will be in multiple systems at once (concurrently). It is likely then, that performing an action in one system may also cause changes in another system (either because that action is explicitly in that other model or via an exit transition). Checking to see what other effects our actions may cause is probably best handled via a post processing step (see the aside in section 3.1.3.2).

Another possibility is that the only plan that can achieve a goal is one that acts concurrently in two systems—that is, we need to interleave actions within two systems to achieve some goal. It would be possible to infer this on the fly, but computationally extremely expensive. Instead the planner should be able to determine that while each goal assertion can be achieved in isolation, achieving either assertion prevents the achievement of the other (in many cases, this could be inferred by analysing why the planner failed on certain branches). A second phase planner that considers qualitative and timing information could then explore the possibility of a concurrent plan.

Aside: Constructing non concurrent systems

When evaluating how to handle concurrency, we determined that in most cases, systems can and should (and given how information is learned, most likely will) be constructed in such a way as to avoid tasks having to be performed concurrently. For example, we could have two separate models, one for heating food and another for stirring food. When learning these systems, however, the learning agent should notice the correlation between stirring and cooking and so produce another system that examines the effects of performing both tasks together. Similarly, if someone is told how to cook food, you would immediately construct a joint system. Due to this fact, the importance of dealing with concurrency is reduced.

3.4.3 Specialised versus General Models

A specialised model is one that contains a super set of the assertions that are true in another (more general) model—this means specialised models can always be inferred (although they could and probably should be stated explicitly as well). Wherever a specialised model can be matched, the more general model will be able to be matched as well. We should, however, always use more specialised models (instead of rather than as well as generalised models) where available, as they may contain more specific information about what will happen

when the agent performs certain actions. Note that the use of a regressive planner could potentially cause issues as we may believe a generalised system is appropriate, but earlier steps could unexpectedly satisfy the context assertions of a more specialised system instead (which could produce a different result than the more general system).

3.4.4 Specifying values as Variables

A possible enhancement to the system models that we use could be to specify values as variables. For example, the assertion "`property temperature water X`" states that the temperature of water is some variable X. This would allow us to specify other values in terms of X (for example, *the same as* X, *less than* X, *in between* X and Y). An example of where this might want to be used is in a Microwave model; we could state that the temperature of an object in the microwave will be *greater than* it was before after the microwave is turned on for some time. Alternatively, we could specify X in terms of something that can be directly inferred from the observations of the world (or from assertions in a plan). For example, in the assertion "`property direction X`", X could be specified as "`facing objectA`". Thus, with a little inference, we could work out the direction of X by observing the direction of A and taking the opposite.

Our current implementation does not specify any values as variables. While using variables for values does make some effects easier to express, the models become more complex (there will be more work required to bind values to variables, etc). As an alternative, we allow a range of values to be specified in any assertion. That is, the value of a property or qproperty assertion can be specified as a set of qualitative landmarks and ranges. Similarly, we allow a range of possible relations to be specified in relational assertions. This allows us to show "disinterest" at certain times. For example, we may say the temperature of a fluid in the microwave rises from *room temperature* to a value that is either in the *between room temperature and boiling* or *boiling* (but we do not care which). We currently ignore the possibility of specifying values as variables in terms of observations about the world. This is an area that could be looked into further.

3.4.5 Several Other Points

- **Cleaning up systems:** When we perform almost any task, we inevitably have to clean something up once we are done. Cleaning up is important as it generally involves restoring systems to their "ready" state. The planner may be able to infer to how to clean up systems without being told specifically, by performing certain exit transitions in systems that it were traversed through in the plan (as a post processing step). Alternatively, certain states, or exit effects could be explicitly specified as part of the cleanup process, and we could ensure that these states are reached or that these exit effects occur at some point in the planning process.
- **Maintaining goals:** Sometimes, we may want to specify that a set of goal assertions need to be "maintained". Essentially, this just requires checking to see if any of the goal states (states that the agent ends up in and that were used to achieve a goal assertion) have an outward *time passing* transition. If they do, then find a path back to the goal state from the state that the time passing transition leads to. This means the goal assertions can be maintained over time.
- **Using models for inference:** A possible way of incorporating inference into the system directly is to construct models that have no actions. Instead, they would require context

assertions, entry and exit effects, and only a single state. Essentially these models would denote that when the context assertions are achieved, the assertions in the state become true, and when the context assertions are invalidated, the assertions in the state become false. This technique has not been looked into in detail. It may turn out that representing inference in this manner is much more inefficient than incorporating it more specifically into the planning algorithm. At the same time, it may (or may not) be possible for a learning agent to learn these kinds of inference systems (it may be that this information could be extracted from a combination of other learned systems).

- **Using partial order planning lookahead as a heuristic:** This was not really considered. Worth mentioning?

3.4.6 Probabilities and Costs

Another look at probabilities and costs versus qualitativeness.

3.4.7

Actions at one level as plans at lower level stuff (see note 36).

4 Conclusion

Can solve simple to moderately difficult tasks (that don't involve too many systems). Improvements still need to be made.

Some notion that adding costs / probabilities, important but wouldn’t cover everything.

Concurrency is a bit of an issue.

5 Appendix

5.1 Qualitative Best & Worst Expression Evaluation Tables

See honours report—note that unsafe values still do not have numbers associated with them which ***will*** cause problems in some systems that involve unsafe plans. Ideally, unsafe plans should be numbered with the path cost best case path cost when loops are involved, and the worst case path cost if no loops are involved. Simply using best case path costs only, however, should prevent errors (whilst not always make the best choices).

5.2 Pseudo-code of the Single System Algorithm

This section provides a pseudo-code overview of the updated single system planning algorithm.

```
generate indistinguishable states (see honours report)
initialize all states with expressions (see honours report)

visitState(initialState)

highlight plan (see honours report)
```

```
1.  visitState(currState)
 add currState to list of visited states

 for each neighbouring state (who has a corresponding variable in currState's expression)
  if this neighbouring state has not yet been visited
   visitState(neighbouring state)

 update any occurrences of currState in currState's expression with the value at-least-loop, and evaluate
 for each state visited so far, from the last state to the first
  update any occurrences of currState in the state's expression with at-least-loop, and evaluate
  if currState's value is now grounded
   cascadingUpdate(state)
```

```
1.  cascadingUpdate(state)
2.    update(state)
3.    while the update queue is not empty
4.      update(state)
```

```
1.  update(state)
2.    take the nextState off the queue
3.    for each state visited so far, from the last state to the first
4.      update any occurrences of nextState in the state's expression with the updated value of nextState, and evaluate
5.      if state's expression became grounded or was already grounded and improved with the update
6.        add the state to the update queue (and remove any occurrences of state already in the queue)
```

5.3 Pseudo-code of the Multi System Algorithm

This section provides a pseudo-code overview of the multi system planning algorithm. Note that to make the following pseudo-code easier to understand, it ignores some of the more complex steps, such as taking into account created and destroyed objects, and computing scores (see the relevant sections of this report that describe the algorithm for more details):

```
1.  find all systems that contain a state that matches at least one goal assertion, and the corresponding partial bindings
2.  find all possible full bindings for each system that we have generated partial bindings for
3.  retain the best looking bindings based on heuristic (section 3.1.1)
4.
5.  for each system+binding, find all possible “goal” states that satisfy at least one goal assertion
6.  for each “goal” state, check to ensure no “exit effects” cause conflict with actions later in the plan
7.  for each “goal” state, check to ensure no “must be true” assertions are not true in the state (if so, ignore this state)
8.  retain the top “goal” states per system based on number of satisfied goal assertions and number of states that can reach it
9.  retain the best looking goal states based on more detailed heuristic
10.
11. for each goal state, find all possible initial+goal state pairs where at least one goal assertion is satisfied
12. for each initial+goal state pair, check to ensure no existing objects are being created or non existing objects destroyed
13. if there are no matching pairs, fail, and backtrack (see line 25)
14. retain, and order the best “widthAtLevel” initial+goal state pairs based on more detailed heuristic
15.
16. take the first initial+goal state pair off queue:
17.   if all context and initial state assertions are satisfied, and no goal assertions remain
18.     return a single system plan going from the initial to goal state specified in the pair
19.   else if at “maxDepthForThisIteration”
20.     fail, and backtrack (see 25)
21.   for all context and initial state assertions that are satisfied, add them to the “must be true” set of assertions
22.   for all context, initial state and remaining goal assertions that are not satisfied, recursively find a plan to satisfy them
23.   if able to recursively find a plan to satisfy all these assertions
24.     return recursively generated plan(s) with a single system plan based on this initial+goal state pair added to the front
25.   if unable to satisfy these assertions recursively (fail was returned)
26.     if the queue of initial+goal state pairs is not empty
27.       go to the next initial+goal state pair in the queue
28.     else
29.       fail, and backtrack
```