Relevance Based Planning: A Worked Example

Eric B. Baum

Baum Research Enterprises
41 Allison Road
Princeton NJ 08540
ebaum@fastmail.fm

Abstract

The problem addressed is of planning how to achieve goals in a domain simulation. A method called Relevance Based Planning(RBP) is discussed and exemplified in the game Sokoban. RBP forms a high level plan, then refines it in a causal way that substantially reproduces introspection. As the plan is refined, recognition of problems or patterns in a simulation summons methods for solving the problems (which may recursively involve RBP).

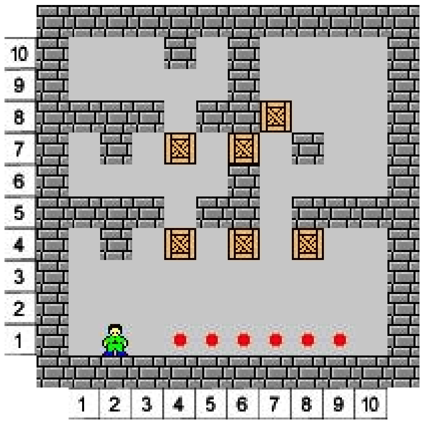
Introduction

Look at figure 1. This is a level from the game "Sokoban", which the reader is encouraged to play on the web a few times. The player controls the man, who can take one step in any direction, and can push (but not pull) one box, but only one at a time, and only if there is empty space on the other side of the box. The object of the game is to push all of the boxes onto goals (places marked with red stars). A particular map is called a level. (Another level is shown in figure 3.)

This game makes an interesting example for AGI Projects because it is extremely hard to solve (technically speaking, the game is P-Space complete) and because levels have been crafted that exercise a variety of human cognitive abilities (the goal of the level creators having been to create levels that are interesting to players). Humans make complex use of a large number of interesting concepts in solving these levels. A program to solve this game in a way mirroring human solution will thus be hierarchically built on top of subprograms implementing various concepts, such as "deadlock".

By a deadlock I mean, a (partial) box configuration where the level is no longer solvable. For example, if two boxes are side by side along a wall then neither of them can be pushed, because there is no position where the man would be pushing only one box. So neither of them can be pushed onto goals, and (excepting cases where they are both on goals already) the level has become not solvable. There are lots of locally recognizable configurations like this which constitute deadlocks, and with experience a human player learns to recognize numerous deadlock patterns, and an important part of playing is analyzing suspicious local configurations of boxes to see if they have become deadlocks. This analysis can be subtle (indeed, the general problem of identifying deadlocks is P-Space complete). Eventually, if we are going to build a program to solve Sokoban in human fashion, we will need subprograms that recognize deadlocks. A first start at such a subprogram might contain a number of deadlock patterns, such as the pattern: two boxes against a wall, and shout when it recognizes a pattern indicating that a deadlock has been created, and return a specification of which boxes are involved.

Figure 1. A Sokoban level



A slightly more sophisticated deadlock recognition program (which would invoke the first start as a subprogram) would perform a local search to see if boxes could be usefully pushed, whenever a suspicious pattern was recognized. For example, whenever a collection of boxes and walls form a barrier that topologically divides the level into two pieces, and thus prevents the man from reaching some part of the level without pushing a box (in order to penetrate the barrier), that is a candidate deadlock. For example, in figure 1, the boxes at (4,4) and at (6,7) form a barrier, preventing the man from reaching the region at (4,6) without pushing one of them. If there were no way for him to get behind the box at (4,4), then he could never push it down to a goal, and there would be a deadlock. So it would be useful to perform a search to see whether in fact there is a way that he can push boxes out of the way and get behind this barrier, without causing some other recognizable deadlock in so doing.

Notice how human reasoning about the game exploits topological concepts (such as that a barrier separates a 2 dimensional region into two disconnected components) and locality (for example, you do a spatially local analysis on a pattern to determine deadlock), and is able to do these things by analyzing the 2-d game simulation.

It is a goal to build a hierarchic collection of modules, procedures, that acts on the simulation domain and roughly recreates the thought processes a human player might go through in solving Sokoban problems. An attempt to do this by hand coding discussed previously (Baum,2007). (I now expect that construction of such programs will require a collaborative effort between human and automated processes(Baum, 2008).) Here we will work an example in some detail to indicate how the function that we call Relevance Based Planning works, and then describe how the hand coded program was based on it, and make a few comments on how we might go about constructing a fully successful program.

An Example of Move(a,b)

Lets first consider a problem simpler than actually solving the game, that of getting the man to other locations, without creating a deadlock. A procedure move(a,b) that does this would be a very useful subprogram-- we will constantly need to use it to check if we can get behind barriers, and constantly need to use it to get behind boxes we want to push into goals or out of the way of other boxes. So in particular, consider the problem of moving the man from its current location at (2,1) to (4,9).

Now, obviously, there are two high level plans we might consider. We might go through the portals at (4,5) and (4,8) (which will require dealing with the boxes at (4,4) and (4,7) first) or we might go through the portals at (7,5) (6,7) and (4,8) (which will require dealing with the boxes at (6,7) and (4,7) first). This is a good example of hierarchical planning: we first form high level plans, based on analysis of the simulation domain, and knowledge about what obstacles in it are affectable, and then proceed to analyze whether the obstacles can in fact be dealt with. In the case of navigation in domains like this, you can just engage in search, or dynamic programming, to find paths through boxes but avoiding walls. More generally, you might invoke a dynamic programming that allows contrafactuals-- if only this box weren't here-- and considers paths that could reach the goal. To make this work, however, you need knowledge about which contrafactuals are sensible-- you allow contrafactuals about affectable obstacles, like boxes, (because you might later be able to find a way to deal with these obstacles and refine the plan until it is implementable) but not about unaffectable obstacles, like walls.

Lets consider first the path through (4,5). The first obstacle is the box at (4,4). We consider pushing it left, to (3,4), but that causes a deadlock (a box in a corner). So instead we consider pushing it right, to (5,4) but that also causes a deadlock, two boxes along a wall. But this time, the box at (6,4) is part of the deadlock, and that is an affectable object. So lets consider pushing this box out of the way, before we move the box at (4,4).

So we consider first pushing the box at (6,4) to (7,4). Now this forms a new barrier (with the box at (6,7) potentially preventing us from accessing the upper right chamber. So we first have to analyze whether this barrier could later be penetrated, or whether it will form a deadlock. (This amounts to a recursive call on move(a,b) to see if we can get behind this barrier.) That analysis shows this is a deadlock, unless we take another preparatory move. For example, before moving the box from (6,4) to (7,4), we push the box from (8,4) to (9,4). Doing this makes a space at (8,4) that allows the pusher access to push the box at (7,4) back to (6,4) and open a path to the chamber, so the deadlock is avoided.

OK, so our current plan is: first push (8,4) to (9,4); then push (7,4) to (8,4); then push (4,4) to (5,4), and now we can access the chamber at (4,6). So let's proceed forward on our first candidate plan to get to our goal at (4,9), which now requires pushing the box at (4,7) out of the way. We can't push it left, without deadlock, to push it right requires first moving the box at (6,7). We can't push this box right, unless we first push the box at (7,8) up. To push the box at (7,8) up, we need to access (7,7) (because that's the square the man must push from).

So now we realize, we should have pushed (7,8) up to (7,9) before pushing the box at (7,4) to (8,4)! So we revise our plan to insert this push earlier, and we have a complete plan:
first push (8,4) to (9,4); then push (7,8) to (7,9); then push (7,4) to (8,4); then push (4,4) to (5,4); then push (6,7) to (7,7); then push (4,7) to (5,7); and now the man can access (4,9).

I hope the reader will agree that this is basically the way he would think about the problem (at least after a little practice with Sokoban). A high level algorithm for doing this in a more general context is sketched in figure 2.

Figure 2, Relevance Based Planning Pseudocode:

(1) Find high level plans by search or dynamic programming in which you consider plans that ignore affectable obstacles (allow counterfactual steps which are known to be potentially remediable).

(2) Refine one of these in time order.

(3) For each obstacle in the plan as you consider it, invoke a method that attempts to deal with the obstacle. This method typically is informed about the type of obstacle. The method typically performs some search on the

simulation domain (and may often invoke or even recursively call RBP).

Such calls to clearing methods typically pass information about higher level goals within the plan, and the called clearing method may then avoid searching a set of actions to remove the obstacle that would have as prerequisite previously achieving a higher level goal. For example, in clearing a box out of the way so that the pusher can get to the other side of a barrier, the clearing method will not consider pushes that require already accessing the other side of the barrier.

(4) If the search encounters other problems that would require earlier actions, back up to attempt those actions first (typically by invoking a method for dealing with that obstacle). For example, each box move must be checked to see if it causes a recognized deadlock. If it does, you need to back up and first invoke a method for preventing the deadlock from arising. This method performs a search over ways of first pushing other boxes involved in the recognized deadlock, trying to achieve the goal of finding a configuration where the desired push won't cause a deadlock. (Note, the deadlock detector needs to return the collection of boxes involved in the deadlock for this to be done.)

(5) When changes are made in the domain, mark them, so that if they encumber later actions, you can back up and try to insert the later actions first to avoid the problem.

(6) You also need a method of backing up to other high level plans when it is discovered that a high level plan can not be made to work, or of switching between high level plans as more information is uncovered about them. One effective method, that finds the cheapest plan in Sokoban man moves, is to maintain a set of all candidate plans in parallel, working on the cheapest estimated one at any time. As pushes are added to a candidate plan, update its score (and switch to working on another one if candidate plan with a less expensive estimate is available). As alternative fixes are discovered (for example, to prevent a deadlock from occurring involving barrel A you may try different directions of pushing barrel A out of the way) add a new candidate plan to the set for each such push. Keep a hash table of board configurations, and if you ever hit a duplicate position, delete all candidate plans but the cheapest for reaching it.

Discussion

RBP illustrates the power of the use of the simulation domain. You form a high level plan over the simulation domain. Then you proceed to analyze it in interaction with the simulation domain. As modules are executed to solve problems, the interaction with the simulation domain creates patterns that summon other modules/agents to solve them. This all happens in a causal way: things being done on the simulation cause other things to be done, and because the simulation realizes the underlying causal structure of the world, this causal structure is grounded. This interaction with the simulation domain is unlike any other I am familiar with in the planning literature. It also powerfully focuses the search to consider only quantities causally relevant to achieving the goal. It also reproduces introspection.

Many planning methods choose not to work in time order for various reasons. But by working on a candidate plan in time order, you are assured, at the inner most loops of the program, of only spending time considering positions that you know you can reach, and which are causally relevant to a high level plan.

Note also that this construction of the program by composing a series of feasible-sized goal oriented searches mirrors the basic structure of biological development (cf (Baum 2008)), and is similarly concisely coded and robust.

A larger meta-goal of the project described in (Baum 2008) is to construct Occam programs, programs that are coded in extremely concise fashion so that they generalize to new problems as such problems arise (or so that the programs can be rapidly modified, say by a search through meaningful modifications, to solve new problems). As has been discussed(Baum 2007, 2008), search programs are incredibly concisely coded, so that coding the program as an interaction of a number of search programs can be a mode of achieving great conciseness. Note how this concise program generalizes over Sokoban levels and positions and man origins and destinations.

Caching the results of the local searches enables them to be rapidly swapped into alternative plans (plans that differ in other regions, for example), speeding the overall computation and again mirroring introspection (which shows that I solve Sokoban and other problems by finding and then composing local solutions to parts of the problem using the overall simulation environment to interleave the local solutions causally.)

At the lowest level searches, for example preventing deadlocks from occurring, an initial class of methods may simply do a search over all actions that might be relevant. For example, when we were trying to push (4,4) to (5,4) above, and realized we would form a deadlock with (6,4), we called a method to prevent this deadlock, and a naive version of this method would simply try all pushes of (6,4), and might initially consider pushing (6,4) to (5,4) (which a human player would recognize doesn't make any sense). Even being naive about these methods in this way still greatly reduces work that must be done, because the problem is factored into a number of small searches that only consider potentially relevant actions. An alternative to considering all potentially relevant actions is to have (or learn) knowledge that allows this search to be pruned in other ways. For example, introspectively, I recognize lots of local patterns that I use to prioritize such searches. I will tend to consider first moves into open areas (and not consider moves into patterns I recognize as a deadlock). In fact, introspectively, a frequent reason for my getting stuck in solving a Sokoban instance is that I have rejected the correct move because it matches some pattern that I regarded as meaning it wouldn't work; and a big improvement in my game came when I learned a number of patterns that fool players in this way, that is by resembling deadlock patterns, (I suspect designers of building these in on purpose) and learned to search such moves first. In general, recognizing patterns involves a local search of its own, so unless the patterns are well chosen and implemented, need not be faster or more effective than simply doing a broader search. However, if an Evolutionary Economic System (EES) (Baum, in prep) based on local pattern recognizing agents is used to learn or improve these modules for dealing with problems, it should naturally learn this kind of pattern knowledge, reproducing introspection, and, because of the economic framework, learn only patterns that improve efficiency.

The general structure described in Figure 2 could be regarded as a Scaffold, a Procedure that takes a number of arguments, themselves procedures, that need to be filled in, for example by a module constructor, to work in a specific example(Baum, 2007). This scaffold would need to be supplied with a domain simulation, a procedure for finding high level plans, and procedures for recognizing and dealing with obstacles, and possibly other extensions. The RBP scaffold can then be reused in various contexts, for example can be used within module constructors to construct some of the submodules. Obviously such a structure promotes concise code through code reuse, and represents an analysis method that in a sense "understands" how to use high level plans and how to exploit causality.

Once you have a scaffold for RBP, and/or modules written implementing RBP, these can be supplied as instructions out of which EES's and other module constructors can build other modules or agents. A simple example of this was described in (Schaul, 2005).

RBP for Sokoban Solving

A draft program written by Tom Schaul to my direction (also discussed in (Baum 2007) was intended to solve Sokoban by applying a version of RBP, with all of the code, including all the subsidiary modules, written by hand. The method involved first finding high level plans for bringing barrels into goals under the contrafactual assumption that barrels not yet placed on a goal for the last time were permeable to other barrels, then selecting a high-level plan based on estimated cost, and following it in time order trying to resolve obstacles, and then backing up to other plans in a causal fashion to resolve pusher access constraints as found.

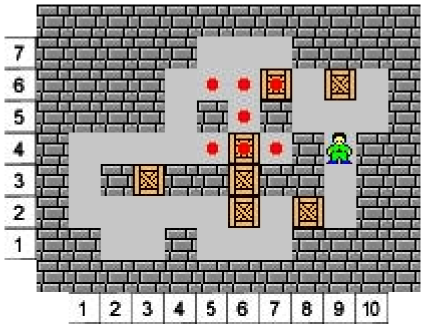
Although following the same basic structure as Figure 2, this is a somewhat more sophisticated planner in a few ways than that exemplified for move(a,b). First, step 1 in the pseudo code above, the high-level plan finder, is a composite of several modules. It involves a matching analysis and a goal area analysis. The matching analysis used a version of a standard matching algorithm to find one-to-one correspondences of boxes to goals, where each correspondence is achievable (given the flow pattern of the level) if boxes could pass through other boxes. The goal-area analysis found orderings of the goals. In any complete and working plan, there must be a first time for each goal that a box is placed on it and never moved again. Such a box was declared "petrified", since it is like a wall, in that it (by definition) can never be moved later in the plan. By analysis of the goal area(s), a planning module found an ordering of the goals from first petrified to last, assuming all non-petrified boxes are permeable, but no petrified boxes are permeable. This necessarily involves plans that include temporarily stowing boxes in intermediate locations where they will be able to reach later solved goals without violating petrified goals (because it is a property of Sokoban that such situations arise). This component of the problem (goal area analysis) mirrors an introspective component that humans use cognitively to factor the problem.

So the high level plan finder already includes a factorization of the problem (separating bringing boxes up to goal areas in such a way that one box corresponds to each goal, and goal area analysis) and the use of the simulation domain in sophisticated ways to incorporate local structure.

The high level plans generated here thus involved an order in which boxes are brought up to goals and parking locations, and an order for the goals to be petrified. Step 2 picked such a high level plan and went through it in time order. One way of doing this is to go through in order of goals from first petrified to last (or when a box must be stowed in a parking location before a goal is petrified, said parking location ahead of said goal) and to pick the box which is part of a legal matching that in a heuristic sense is nearest that destination (goal or parking location), and to attempt to solve that subtask next. Solving this subtask by the method of figure 2 involves planning to achieve accesses-- to move the man to locations where he is needed to accomplish pushes-- and involves planning to test whether various pushes that are made result in deadlocks, so it should involve recursive calls on RBP (or in any case, one or more implementations of RBP to solve subproblems).

In principle this can exploit causal constraints in a powerful way similar to how I introspectively would. Consider figure 3.

Figure 3: Another Sokoban Level



You need to recognize that you should first solve goal (7,4) because it is completely safe-- it can't block anything else you want to do. (Safety being a concept useful in goal ordering for which you must create a module.) Then you might (for example) try solving goal (5,6) next. If you do that, the box you place there would be marked (as described in step 5 of figure 2). As you continue, you will eventually discover this marked box is interfering with accesses, because there will be a barrier you will be unable to cross that includes the box at (5,6) and the box at (8,2). So RBP will say you should insert ahead of pushing the box to (5,6), moves that clear the box at (8,2) so that there is an access. Searching to clear (8,2) will involve discovering that you have to first push the box at (6,2) out of the way and etc. (to do this you can find by forward search a long sequence of preparatory moves) and will lead to a solution using essentially the same concepts I use introspectively-- it realizes it has to prioritize clearing this access and then does a forward, goal oriented search, that accomplishes that subject to access constraints.

Our attempts to build this Sokoban player by hand using RBP did not however succeed in realizing the full vision (although they resulted in a program that solved a third or so of standard hard levels it was tested on(Baum 2007)). Some lessons I learned from this are the following.

You can not write the inner modules, such as the deadlock detector, or the methods for dealing with various kinds of obstacles, by hand. Our project failed for similar reason (IMO) as did SHRDLU- because we tried to hand code it. None of the modules captured the desired concept adequately. These things need to be coded by module constructors(Baum 2008). Moreover the system, being hand coded, had no way to adjust the modules and their interaction to correct for the problems. If instead they were, for example, Evolutionary Economic Systems, that learned from new examples and adapted, the system could evolve to solve its problems.

And the coding needs to be concise, Occam. The coding that was attempted, by hand, did not attempt to be concise in the sense of Occam's razor. It did not, for example, make recursive call on RBP for internal problems, but rather inserted such in an ad hoc way. Similarly, it made no use of scaffolds. The coding should be (automatically) done in terms of a collection of goal oriented agents, built (for example, using Evolutionary Economic Systems) on scaffolds such as RBP.

The method that I use, introspectively, to solve Sokoban does not proceed from beginning to end by a master RBP. It proceeds in a goal oriented fashion, using goal oriented modules based on RBP, caching local knowledge as it goes, and gradually learns or discovers various components of how to solve the problem, including such things as, in the discussion below Figure 3, that you have to prioritize clearing the access at (8,2). It includes modules with goals such as safe parking, clearing access, going forward as far as forced in a position, and so on.(Baum 2007).

Each such module should be concisely built, typically using an RBP scaffold, calling other such modules as desirable, and adjusting itself, if problems are encountered, to address them.

.

References

Baum, Eric B. 2004. *What is Thought?* . Cambridge, MA: MIT Press.

Baum, Eric B. 2007. *A Working Hypotheses for Artificial General Intelligence.* In *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms,* eds. Goertzel, B., and Wang, P., IOS Press, pp 55-74.

Baum, Eric B. 2008. *Project to Build Programs That Understand.* Submitted (this volume).

Schaul, T. 2005. *Evolution of a compact Sokoban solver.* Master Thesis, École Polytechnique Fédérale de Lausanne. posted on http://whatisthought.com/eric.html.