The History of Partial Order Planning and RubyPOP

by Sammy Larbi

What is Partial-Order Planning?

For our case, we'll explore partial-order planning in a classical planning environment. Such an environment is fully observable (as opposed to only partially so) and deterministic (as opposed to having randomness, or being stochastic). Further, the space is finite and static in nature - it does not change in the middle of deliberation. Finally, the environment is "discrete (in time, action, objects, and effects)," as opposed to continuous along any of these axes (Russell, 375. For further reading on the characteristics of environments, see Russell pages 41-42).

To understand what partial-order planning entails, it might be helpful to know what planning is, and then describe totally ordered planning. To that end, planning is "the task of coming up with a sequence of actions that will achieve a goal" (Russell, 375). That is a fairly straightforward - exactly as we would expect if we weren't speaking of computers and programs.

An example is simple: given a set of actions I can perform, which ones do I choose (and in what order should I apply them) in order to reach my goal? I've got to get to work this morning - what should I do to get there? I might need to wake up, turn off the alarm, shower, take off my wet pajamas and put on something suitable for doing business, and so on, until I reach work in the morning (or, afternoon if it was a late night). However, we wouldn't consider having our agent perform the actual driving of the car in a classical environment. This is because in doing so, our environment loses several of the characteristics we laid forth above - chief among these are that the new environment would become stochastic and continuous.

There are several types of algorithms that allow us to construct a plan. For examples, we briefly examine progression planning, regression planning, and our main topic, partial-order planning. Progression planning is done with a forward state-space search, which is to say that, "we start in the problem's initial state [and consider] sequences of actions until we find a sequence that reaches a goal state" (Russell, 383). This can pose major performance problems because it considers even completely irrelevant actions. As you might guess from its name, regression planning is the opposite - it works backwards from the goal state. This removes the problems associated with examining irrelevant actions, but as Russell notes, it is not without its problems: oftentimes it is not "obvious how to generate a description of the possible predecessors of the set of goal states" (Russell, 384).

What distinguishes partial-order planning from the other two is all in its name - it is not totally ordered as we see in progression and regression planning. Instead, partial-order planning enables us to "take advantage of problem decomposition." The algorithm "works on several subgoals independently, solves them with several subplans, then combines the subplans" (Russell, 387). In addition, Russell notes that, "such an approach also has the advantage of flexibility in the order in which it *constructs* the plan. That is, the planner can work on 'obvious' or 'important' decisions first, rather than being forced to work on steps in chronological order."

We can see a similar phenomenon in application design - we may not choose to work in any chronological order, but instead work on the parts to construct the whole. Whereas some of the most important design decisions are often made at the beginning of a project, when we know the

least about it, the prudent designer may choose to work on more obvious decisions, or decide to choose to work on the most important one *at the time*.

Selected History of Partial Order Planning and its Implementations

It was apparent as far back as 1975 that "linear planning" (or totally ordered planning, as described above) was not sufficient. Russell and Norvig relay the story of Allen Brown's experiment: that it could not solve simple problems such as the Sussman anomaly, where given 3 blocks labeled A, B, and C, with block B on the table and C on top of A which is on the table, get to the goal state of A on top of B on top of C (Russell, 410, 414) (See Figure 1). Around that time, as part of his Ph.D., Austin Tate released a paper which described INTERPLAN, a system to solve the problem of interleaving shown by Brown using Sussman's HACKER program (the Sussman Anomaly) (Tate [A] / Russell 410).

Shortly thereafter, Earl D. Sacerdoti released his paper, "The Nonlinear Nature of Plans" which presented a new data structure to represent plans and gave a glimpse of the first partial order planners, Nets Of Action Hierarchies (NOAH) (and compared that to INTERPLAN, among others) (Sacerdoti, 8).

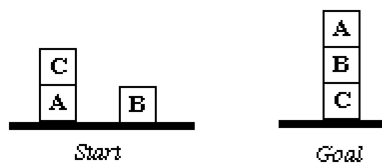

Figure 1. The Sussman Anomaly.

Sacerdoti begins by acknowledging that, "we usually think of plans as linear sequences of actions … because plans are usually executed one step at a time." However, he observed, the "plans themselves are not constrained by limitations of linearity." Because of this, a new structure is needed, which he calls a "procedural net," that would represent "a plan as a partial ordering of actions with respect to time" (Sacerdoti, 1). He backs up his assertion showing the famous Sussman anomaly (described above), and moves on to describe procedural nets and NOAH itself.

Procedural nets are networks of four different types of nodes (GOAL, PHANTOM, SPLIT, and JOIN) and each node represents an action. The nodes are then linked together to form plans. GOAL nodes, clearly, represent a goal that should be achieved. PHANTOM nodes "represent goals that should already be true at the time they are encountered." And, as one might imagine, SPLIT nodes represent splits in the plan, while JOIN nodes represent forks that are coming to an end. Finally, as it is implemented, each of the nodes has a pointer to some code (also known as a closure, lambda expression, or anonymous function) (Sacerdoti, 2).

NOAH uses that structure to represent plans, and a "generic" domain specific language called SOUP (Semantics of a Users' Problem) to give the system knowledge about the task domain (Sacerdoti, 2). To clarify, I call it generic here because it is suitable for describing problems in many domains, but it is specific in that its sole purpose is to describe problems.

To create a new plan, first give NOAH a goal to achieve and tell it about the problem with SOUP. Then it builds a procedural net with only a goal node, which contains a "list of all relevant SOUP functions as its body." Finally, run the planning algorithm. The planning algorithm is described as (quoting Sacerdoti, 3):

> 1) Simulate the most detailed plan in the procedural net. This will have the effect of producing a new, more detailed plan.
> 2) Criticize the new plan, performing any necessary reordering or elimination of redundant operations.
> 3) Go to Step 1.
>
> (Sacerdoti, 3)

Step one is performed in effect by calling the anonymous function pointed to by the node. Step two runs the plan against several critics, namely "Resolve Conflicts," "Use Existing Objects," and "Eliminate Redundant Preconditions," which all perform the actions one would expect from knowing their names (Sacerdoti, 4).

After Sacerdoti's release of NOAH, Tate created a new partial order planner dubbed NONLIN. From Sacerdoti, he recognizes "that ordering constraints should only be imposed between the actions comprising a plan if these are necessary for the achievement of the overall purpose of the plan" and bases his work upon it (Tate [B], 889). On the other hand, Tate realized the need for NONLIN because

> [NOAH] still had to make choices as to the order that actions were to be placed-in a plan to correct for interactions. NOAH made this choice in one particular way. It did not keep any backtrack choice points, so this decision, once made, was irreversible. This leads to an incompleteness of the search space which can render some simple block pushing tasks unachieveable (sic) by NOAH… NONLIN is capable of correcting for an interaction by suggesting two orderings (which are sufficient to ensure the incompleteness of NOAH mentioned above is avoided…). (Tate [B], 889)

Additionally, Tate said, "Other operations performed by NOAH deterministically … should also be considered as choice points" and gives two examples where "if such decisions cannot be undone, some problems are unsolvable." Of course, NONLIN fixed these problems (Tate [B], 889).

As David Chapman noted in 1985, "planners of the most promising ('nonlinear') sort have been complicated, heuristic, ill-defined AI programs, without clear conditions under which they work." And since the time of NOAH, INTERPLAN, and NONLIN (and others I've left out for space reasons), there have been various improvements in the realm of partial-order planning. Chapman's program, TWEAK, is one of them, and was followed by the UCPOP and RePOP planners.

Chapman said that he "decided to copy exactly" Sacerdoti's work on NOAH in implementing his own planner, but he struggled to make it work (and TWEAK was the result of it when he did finally get it to work). In explaining the reason for publishing his work on TWEAK, Chapman quotes Sacerdoti who said NOAH's "basic operations" (Chapman, 1) "were developed in an ad

hoc fashion. No attempt has been made to justify the transformations that they perform, or to enable them to generate all transformations." (Sacerdoti, quoted in Chapman, 1) He then goes on to compare what he has done with Sacerdoti's work to the longstanding AI varieties of "scruffy" versus "neat." In general, the "scruffies" just want to try solutions and figure out what seems to work, and tolerate less mathematical rigor and proof than the "neats" (Russell, 25).

In essence, TWEAK was Chapman's successful attempt at formalizing the "theory about the ways in which interacting subgoals can be dealt with" (Sacerdoti, quoted in Chapman, 1), or formalizing partial-order planning.

Chapman called TWEAK "a rigorous mathematical reconstruction of previous nonlinear planners," and "an implemented, running program," which he described and proved correct in his paper. He was right: eighteen years later Russell and Norvig said his work "led to what arguably the first simple and readable description of a complete partial-order planner," which was incarnated as SNLP (Russell, 410) (on which Russell and Norvig base the POP in their book).

After SNLP came UCPOP in 1992 from Daniel Weld (who also was an author of SNLP) and J. Scott Penberthy. As the title states, it is "a sound, complete, partial order planner for ADL." (ADL is a more advanced way to represent problems, Action Description Language). Their desire to create UCPOP stemmed from two problems they found with existing research into POP algorithms. The first hearkens back to the scruffies vs. neats debate: Many researchers "looked at formal characteristics of languages for describing change," as others built "actual planners, often losing a precise understanding of their programs in a forest of pragmatic choices" (Penberthy, 1). Further, the very few that had "*complete* algorithms" (emphasis added) either only implemented "the restrictive STRIPS representation" of problems (citing TWEAK and SNLP), or represented "plans as totally ordered sequences of actions" (Penberthy, 1).

Penberthy and Weld described UCPOP as "a theorem prover" at its heart. The algorithm itself requires three parameters: a plan, the set of goals that remain, and a set of actions. The goals are expressed as a tuple with two elements: a precondition and a step in the plan (Penberthy 6-7). An overview of the algorithm can be described as follows:

1) If the set of goal states is empty, return the plan (reporting "success").
2) Select a goal from the set of goals. If there is an invalid link that makes the plan impossible, exit, reporting failure.
3) Choose an operator
4) Generate subgoals
5) Protect against threats that may "cause the undoing of a needed goal if that step is done at the wrong time" (Dyer).
6) Recursively call the algorithm if the plan is not inconsistent.

(Penberthy, 6)

Although UCPOP was novel for its time, a visit today to its website (maintained by the authors at the University of Washington) shows it is quite outdated. The authors note, "UCPOP is an *aging system* - we recommend Sensory Graphplan (SGP) instead. SGP handles a superset of UCPOP

functionality and is *much, much faster*" (Weld). On the other hand, research by XuanLong Nguyen and Subbarao Kambhampati in 2000 has shown that quite a few improvements can be made to UCPOP in particular, and partial-order planning in general.

Nguyen and Kambhampati argue that "the prevailing pessimism about the scalability of partial order planning (POP) algorithms" is perhaps unwarranted (Nguyen, 1). The pair appear to regret that research on POP algorithms seemed to stop (around 5 years prior) and make that point that advances in "heuristic state space planners … and CSP-based planners" like Graphplan are perhaps "(mis)interpreted as establishing the supremacy of state space and CSP-based approaches over" those of the POP variety (Nguyen, 1).

As evidence of their claim, they created RePOP, which is a partial-order planner based on UCPOP. But the value of RePOP lies in the authors' "key insight … that the techniques responsible for the efficiency of the currently successful planners … can also be adapted to dramatically improve the efficiency of the POP algorithm" (Nguyen, 1).

The techniques they applied to RePOP that other strategies had applied, "distance based heuristics, disjunctive representations for planning constraints and reachability analysis," led to outstanding results in "several 'parallel planning' domains" (Nguyen, 6). In fact, since in general partial ordered planners are more flexible than their CSP counterparts, they obtained greater flexibility than Graphplan, while also outperforming it in most of their experiments (Nguyen, 5-6). The performance increase wasn't slight, however - in one problem that RePOP took less than 3 seconds to solve, Graphplan labored for 47 minutes. On average, when GraphPlan could solve a problem within 3 hours or without using all 250 MB of memory, it still took around 10 times longer than RePOP to find a solution (Nguyen, 5).

The systems described above have a common link in that they are all focused on generic problems. However, it should be noted that in some domains, such as medicine, planners of these sorts do not tend to work well. As a consequence, great improvements – in performance *and* in accuracy – can be made by building a planner that can be supplemented with domain-specific information (Miksch). Despite the interesting nature of such systems, covering planners of that sort is outside the scope of this paper, and I now turn to the second task of this report – building a partial order planner in Ruby.

Constructing a POP Domain Specific Language in Ruby

Our goal is to give commands to the partial order planner, telling it what the goal is, the initial state (if it exists), and actions it can perform. The actions contain the name of the action, any preconditions that must be fulfilled before that action can be performed, and a set of effects the action has on the world state. After giving this information to the planner, it should output a plan on demand if one exists.

For simplicity's sake, I've used a STRIPS-like notation, without the complexity of existentially or universally quantified variables, among other simplifications. Further, only one possible plan is returned, rather than attempting to find all plans. The one returned is not guaranteed to be optimal, though (inadequate) tests have shown that it is correct. Plans are to improve these

limitations in the future, moving to a less restrictive ADL syntax, and adding support for returning multiple plans.

In the meantime, the first task is to allow a user to enter commands in a syntax that looks like:

```
PlanName("Put on Shoes")
Goal(:RightShoeOn ^ :LeftShoeOn)

Action(:RightShoe, EFFECT => :RightShoeOn, PRECOND => :RightSockOn)
Action(:RightSock, EFFECT => :RightSockOn)
Action(:LeftShoe, EFFECT => :LeftShoeOn, PRECOND => :LeftSockOn)
Action(:LeftSock, EFFECT => :LeftSockOn)
```

Code Sample 1

Also, it should allow function-like notation, such as:

```
PlanName("Change Tire")
Init(At(:Flat,:Axle) ^ At(:Spare,:Trunk))
Goal(At(:Spare,:Axle))

Action(Remove(:Spare,:Trunk),
       PRECOND => At(:Spare,:Trunk),
       EFFECT => NotAt(:Spare,:Trunk) ^ At(:Spare,:Ground))
Action(Remove(:Flat,:Axle),
       PRECOND => At(:Flat,:Axle),
       EFFECT => NotAt(:Flat,:Axle) ^ At(:Flat,:Ground))
Action(PutOn(:Spare,:Axle),
       PRECOND => At(:Spare,:Ground) ^ NotAt(:Flat,:Axle),
       EFFECT => NotAt(:Spare,:Ground) ^ At(:Spare,:Axle))
Action(:LeaveOvernight,
       EFFECT => NotAt(:Spare, :Ground) ^ NotAt(:Spare,:Axle) ^
       NotAt(:Spare,:Trunk) ^ NotAt(:Flat,:Ground) ^
       NotAt(:Flat,:Axle))
```

Code Sample 2

The domain described in Code Sample 1 should produce a plan such as: `LeftSock`→ `LeftShoe`→ `RightSock` → `RightShoe` and `RightSock`→ `LeftSock`→ `LeftShoe`→ `RightShoe`. As one can surmise from looking at the domain as it is written, any plan where the socks are on before the shoes is sufficient.

On the other hand, the domain given in Code Sample 2 should render plans like `Remove(Flat,Axle)` → `Remove(Spare,Trunk)` → `PutOn(Spare,Axle)`, switching the first two actions depending on which it decides to do first (since either one would work).

Implementing (or allowing) such syntax in Ruby turns out to be simple. To get the conjunction operator ^, we simply define a module with ^ as a method, and include that module in Ruby's String, Symbol, and Array classes, since we'll be using each of these as symbols in our "new" language (See Code Sample 3).

```
module Logic
      def ^(condition)
            [self, condition]
      end
end

#modify the symbol class to include the ^ operation
class Symbol
      include Logic
end
#modify the array class to include the ^ operation
class Array
      include Logic
end

#modify the string class to include the ^ operation
class String
      include Logic
end
```

Code Sample 3

Next, we need to allow the use of function-style symbols, such as `Remove(:Spare,:Trunk)`. As with most things in Ruby, this is also simple. We just use the `method_missing` method in our module:

```
# when the user enters a function, turn it into an action
def method_missing(method_id, *args)
      symbol_name = "#{method_id}("
      args.each { |arg| symbol_name += arg.to_s + "," }
      symbol_name[0,symbol_name.length-1] + ")"
end
```

Code Sample 4

We now have the ability to use the syntax we laid forth in Code Samples 1 and 2 to define our problems that need planning. All that remains are to implement the functions in our "language" that allow us to define the problem domain, and an algorithm to solve for plans.

To do so, first we initialize the start state with an `Init()` function that simply stores the conditions it is passed. Similarly, the goal state and initial open preconditions are stored into

member variables as they are passed via the `Goal()` method. Finally, actions are constructed from a name and a hash with keys PRECOND and EFFECT (See Code Sample 5).

```
#constants to use to store hash for precondition and effect
#(only for purposes of keeping the DSL looking close to the original)
PRECOND = :precondition
EFFECT = :effect

#store the start-state conditions
def Init(conditions)
      @start_state = conditions
end
alias init Init

#store the goal defined by the user
def Goal(conditions)
      @goal = conditions
      @open_preconditions = @goal
end
alias goal Goal

# store actions defined by the user
def Action(name, precondition_effect)
      action= ["name" => name,
                  "precondition" => precondition_effect[PRECOND],
                  "effect" => precondition_effect[EFFECT]]
      @actions = [] if !@actions
      @actions = @actions + action
end
alias action Action
```

Code Sample 5

Finally, we come to the meat of the problem, the partial-order planning algorithm. The algorithm itself follows a fairly simple path:

1) From the list of open preconditions, choose one.
2) Find an action whose effect is the same as the precondition we chose and add it to the plan.
3) Add to the list of preconditions any requirements for that action.
4) Remove from the list of preconditions any that match the effects for the chosen action.
5) Repeat steps 1-4 until the set of open preconditions is empty, or no action that satisfies a precondition can be found.
6) Remove any preconditions from the open list that match the starting state.
7) If the set of open preconditions is empty, return the plan. Otherwise, fail.

The algorithm in Ruby follows:

```
def make_plan
      action_plan = []
      fail = false
      while (@open_preconditions.size > 0 && !fail)
            #randomize the open_preconditions and actions to show order
            #doesn't matter
            @open_preconditions=@open_preconditions.sort_by { rand }

            #find an action that solves it the first open precondition
            attempted_precondition = @open_preconditions.shift
            action_to_take = find_action_for attempted_precondition

            if (action_to_take != nil)
                  add_preconditions_for action_to_take
                  remove_preconditions_fulfilled_by action_to_take
                  #add the action to the plan
                  action_plan.push(action_to_take["name"])
            else
                  #put the precondition back on the open_preconditions, since
                  #it wasn't fulfilled by an action
                  fail = true if @open_preconditions.size == 0
                  @open_preconditions.push attempted_precondition
                  remove_preconditions_matching_start_state
                  fail = false if @open_preconditions.size == 0
            end
      end
      if @open_preconditions.size > 0 || fail
            puts "There appears to be no plan that satisfies the problem."
            puts "Open preconditions: "
            puts @open_preconditions
            action_plan = []
      end
      sanitize_plan(action_plan.reverse)
end
```

Most of the code is aptly named where there are functions (see the appendix for the complete code), but two issues in this algorithm immediately jump to the forefront. The first is: why aren't we also randomizing the list of actions? Clearly, if there are two actions that satisfy the same precondition, the first one encountered will always win. This was done because randomizing the list of actions (if two or more satisfy the same precondition) has the potential to cause a loop of preconditions/effects, and thus cause incorrect plans to be generated. Since no attempt was made at finding the optimal plan, I didn't want to clutter the code by fixing this and make it harder to follow. Correct plans are still generated, and a future version meant for more demanding environments would indeed allow a random action to be chosen.

The second issue that is not immediately clear begs the question: "just what is that `sanitize_plan` method doing there?" Some actions may add duplicate preconditions to the set of open preconditions. The algorithm as it stands allows this to happen for readability purposes, and simply cleans up the plan later with the `sanitize_plan` function.

Finally, it is also clear that a more "elegant" solution may have been to take actions as functions, which receive preconditions as their parameters, and whose output are effects. The thought of such an implementation is interesting and worthy of exploration, though time constraints prevented me from doing so in this case.

As mentioned above, a complete version of the code and three tests can be found in the appendix.

Conclusion

First a definition of planning was introduced. In a nutshell, it is "the task of coming up with a sequence of actions that will achieve a goal" (Russell, 375). That much was obvious. We then described two forms of linear, or totally ordered, planning - progression and regression planning through forward and backward state-space search, respectively. Having something to contrast with, the definition of partial order planning became clear, as did its advantages: it is able to exploit problem decomposition, and in doing so, work on several subgoals before arriving at a complete plan.

Then, a long selected history of partial order planning was described. First, Sussman's anomaly was introduced, as it was the impetus behind the design of the first non-linear planners. Sussman's anomaly showed how a simple problem could not be solved by traditional total-order planners, while Austin Tate's INTERPLAN and Sacerdoti's NOAH system were two of the first to do so, using non-linear plans.

Tate recognized some flaws in NOAH, in that it could not backtrack (similar to my own implementation) in case of following an incorrect path, and to solve that problem, he created NONLIN. David Chapman's TWEAK came in 1985, and it was the first formalization of a partial-order planner, which led to SNLP, the first complete and readable one. UCPOP soon followed, and it broke out of the realm of STRIPS and into the richer, more descriptive language ADL.

After UCPOP, research in the field seemed to die off until Nguyen (et al) showed how their system, RePOP, could defeat Graphplan by using many of the heuristics that graph planners were using. It was important because that seemed to be considered undoable by a partial order planner at the time.

Finally, we saw how a simple partial order planning DSL could be implemented in Ruby.

Appendix A - Complete code of rubypop.rb and rubypop_test.rb

rubypop.rb

```
module Logic
	def ^(condition)
		[self, condition]
	end
end

#modify the symbol class to include the ^ operation
class Symbol
	include Logic
end
#modify the array class to include the ^ operation
class Array
	include Logic
end

#modify the string class to include the ^ operation
class String
	include Logic
end

# the pop module
module RubyPOP
	#constants to use to store hash for precondition and effect
	#(only for purposes of keeping the DSL looking close to the original)
	PRECOND = :precondition
	EFFECT = :effect

	#store the start-state conditions
	def Init(conditions)
		@start_state = conditions
	end
	alias init Init

	# store actions defined by the user
	def Action(name, precondition_effect)
		action= ["name" => name,
		            "precondition" => precondition_effect[PRECOND],
		          "effect" => precondition_effect[EFFECT]]
		@actions = [] if !@actions
		@actions = @actions + action
	end
	alias action Action

	#store the goal defined by the user
	def Goal(conditions)
		@goal = conditions
		@open_preconditions = @goal
	end
	alias goal Goal

	def PlanName(name)
		@plan_name = name
```

```
	end

	def output_actions
		@actions.each do |x|
			puts
			puts "name: " + x["name"].to_s
			puts "precondition: " + x["precondition"].to_s
			puts "effect: " + x["effect"].to_s
		end
	end

	def clear
		@actions = []
		@goal = nil
		@open_preconditions = nil
		@plan_name = nil
	end

	# when the user enters a function, turn it into an action
	def method_missing(method_id, *args)
		symbol_name = "#{method_id}("
		args.each { |arg| symbol_name += arg.to_s + "," }
		symbol_name[0,symbol_name.length-1] + ")"
	end

	def print_plan
		puts "One possible plan for #{@plan_name}: "
		puts get_plan
	end

	def get_plan
		return sanitize_plan(make_plan)
	end

	private
	def find_action_for(cond)
		@actions.each do |action|
			if action["effect"].class == Array
				action["effect"].each { |effect| return action if
effect.to_s == cond.to_s	}
			else
				return action if action["effect"].to_s == cond.to_s
			end
		end
		return nil
	end

	def remove_preconditions_matching_start_state
		@open_preconditions.each do |cond|
			@open_preconditions.delete(cond) if
@start_state.index(cond)
		end
	end

	#if there were some actions that duplicated precondition, it will cause
a loop in plan.
	#this function cleans that up by analyzing the current state and
```

```
removing unnecessary actions
      #a better implementation might make a graph of the actions and check
that before putting them in
      def sanitize_plan(plan)
            current_state = []
            #should be examining the effects individually, since it may end
up choosing two with the same effect
            #but not currently doing so
            plan.each { |action| current_state.push(action) if
!current_state.index(action) }
            return current_state
      end

      def make_plan
            action_plan = []
            fail = false
            while (@open_preconditions.size > 0 && !fail)
                  #randomize the open_preconditions and actions to show order
doesn't matter
                  @open_preconditions=@open_preconditions.sort_by { rand }
                  @actions = @actions.sort_by { rand }   #---- causes bugs
right now?

                  #find an action that solves it the first open precondition
                  attempted_precondition = @open_preconditions.shift
                  action_to_take = find_action_for attempted_precondition

                  if (action_to_take != nil)
                        add_preconditions_for action_to_take
                        remove_preconditions_fulfilled_by action_to_take
                        #add the action to the plan
                        action_plan.push(action_to_take["name"])
                  else
                        #put the precondition back on the open_preconditions,
since it wasn't fulfilled by an action
                        fail = true if @open_preconditions.size == 0
                        @open_preconditions.push attempted_precondition
                        remove_preconditions_matching_start_state
                        fail = false if @open_preconditions.size == 0
                  end
            end
            if @open_preconditions.size > 0 || fail
                  puts "There appears to be no plan that satisfies the
problem."
                  puts "Open preconditions: "
                  puts @open_preconditions
                  action_plan = []
            end
            sanitize_plan(action_plan.reverse)
      end

      #add the preconditions for this action if they don't already exist
      def add_preconditions_for(action)
            preconditions = action["precondition"]
            if preconditions.class == Array
                  preconditions.each { |precondition|
@open_preconditions.push(precondition) if (precondition != nil &&
```

```
!@open_preconditions.index(precondition)) }
            else
                  @open_preconditions.push(preconditions) if (preconditions
!= nil && !@open_preconditions.index(preconditions))
            end
      end

      # remove any open preconditions which the action fulfilled
      def remove_preconditions_fulfilled_by action
            action["effect"].each do |effect|
                  @open_preconditions.each { |precon|
@open_preconditions.delete(precon) if precon.to_s == effect.to_s }
            end
      end

end
```

rubypop_test.rb

```
require 'rubypop'
include RubyPOP
####################################################
PlanName("Put on Shoes")
Goal(:RightShoeOn ^ :LeftShoeOn)

Action(:RightShoe, EFFECT => :RightShoeOn, PRECOND => :RightSockOn)
Action(:RightSock, EFFECT => :RightSockOn)
Action(:LeftShoe, EFFECT => :LeftShoeOn, PRECOND => :LeftSockOn)
Action(:LeftSock, EFFECT => :LeftSockOn)

print_plan
clear
puts

####################################################
PlanName("Change Tire")
Init(At(:Flat,:Axle) ^ At(:Spare,:Trunk))
Goal(At(:Spare,:Axle))

Action(Remove(:Spare,:Trunk),
         PRECOND => At(:Spare,:Trunk),
       EFFECT => NotAt(:Spare,:Trunk) ^ At(:Spare,:Ground))
Action(Remove(:Flat,:Axle),
         PRECOND => At(:Flat,:Axle),
       EFFECT => NotAt(:Flat,:Axle) ^ At(:Flat,:Ground))
Action(PutOn(:Spare,:Axle),
         PRECOND => At(:Spare,:Ground) ^ NotAt(:Flat,:Axle),
       EFFECT => NotAt(:Spare,:Ground) ^ At(:Spare,:Axle))
Action(:LeaveOvernight,
        EFFECT => NotAt(:Spare, :Ground) ^ NotAt(:Spare,:Axle) ^
NotAt(:Spare,:Trunk) ^ NotAt(:Flat,:Ground) ^ NotAt(:Flat,:Axle))

print_plan
```

```
clear
puts

##################################################
PlanName("Watch Fulham beat Chelsea")
Init(Asleep(:Sam))
Goal(In(:Sam,:Pub) ^ Watching(:Sam,:FulhamBeatChelsea))
Action(WakeUp(:Sam),
       PRECOND => Asleep(:Sam),
       EFFECT => Awake(:Sam))
Action(Bathe(:Sam),
       PRECOND => Awake(:Sam),
       EFFECT => Clean(:Sam))
Action(Shower(:Sam),
       PRECOND => Awake(:Sam),
       EFFECT => Clean(:Sam))
Action(Dress(:Sam),
       PRECOND => Clean(:Sam),
       EFFECT => Dressed(:Sam))
Action(Work(:Sam),
       PRECOND => Clean(:Sam) ^ Dressed(:Sam),
       EFFECT => IsAbleToGoToPub(:Sam))
Action(GoToPub(:Sam),
       PRECOND => IsAbleToGoToPub(:Sam),
       EFFECT => In(:Sam,:Pub))
Action(Watch(:Sam,:Fulham),
       PRECOND => In(:Sam,:Pub),
       EFFECT => Watching(:Sam,:FulhamBeatChelsea))
print_plan
```

References

Chapman, David. "Planning for Conjunctive Goals." *Massachusetts Institute of Technology.* 1985. Available online from MIT at http://dspace.mit.edu/handle/1721.1/6947?mode=full

Dyer, C.R. "Partial-Order Planning: Chapter 11." CS 540 Lecture Notes. Retrieved on the World Wide Web on 4/24/2007 from http://www.cs.wisc.edu/~dyer/cs540/notes/pop.html.

Figure 1: Retrieved on the World Wide Web on 4/20/2007 from http://www.cs.cardiff.ac.uk/Dave/AI2/sussman.gif and verified in Russell and Sacerdoti.

Miksch, Silvia. "Plan management in the medical domain." *AI Communications 12,* pp 209-235. 1999.

Nguyen, XuanLong and Kambhampati, Subbarao. "Reviving Partial Order Planning." 2000. Retrieved on 4/20/2007 from http://rakaposhi.eas.asu.edu/ucpop-revive.pdf on the World Wide Web.

Penberthy, J. Scott and Weld, Daniel S. "UCPOP: A Sound, Complete, Partial Order Planner for ADL." *Third International Conference on Principles of Knowledge Representation and Reasoning*, pp. 189-197. Cambridge, MA.

Russell, Stuart and Norvig, Peter. *Artificial Intelligence: A Modern Approach 2nd Edition*. New Jersey: Pearson Education, 2003.

Sacerdoti, Earl D. "The Nonlinear Nature of Plans." 1975. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*. Retrieved on 4/20/2007 from http://dli.iiit.ac.in/ijcai/IJCAI-75-VOL-1&2/PDF/028.pdf on the World Wide Web.

Tate, Austin [A]. "INTERPLAN: a plan generation system which can deal with interactions between goals" Research Memorandum MIP-R-109, Edinburgh: Machine Intelligence Research Unit, December 1974. (Can be found online at http://www.aiai.ed.ac.uk/project/oplan/documents/1990-PRE/1974-mip-r109-tate-interplan.pdf)

Tate, Austin [B]. "Generating Project Networks", Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77) pp. 888-893, Boston, Mass. USA, August 1977. (Can be found online at http://www.aiai.ed.ac.uk/project/oplan/documents/1990-PRE/1977-ijcai-tate-generating-project-networks.pdf)

Weld, Dan and Penberthy, Scott. "The UCPOP Planner." Retrieved on the World Wide Web on 4/19/2007 from http://www.cs.washington.edu/ai/ucpop.html.