Real-time Action Planning with Preconditions and Effects

by Stavros Vassos

In this article I will be covering an artificial intelligence (AI) technique for decision making that can be used in various parts of game development to account for "thinking before acting". The technique is called Classical Planning in academic AI research, and is one of the most basic approaches for deliberating about the effects of actions and the way the properties of a given domain change under these effects.

Variants of this technique have been used successfully in game development under the name Goal Oriented Action Planning (GOAP), each time focusing on a different aspect of this technique, adopting it for the particular needs of the game. In this article I will present a case for classical planning that follows closely the academic STRIPS paradigm which is based on a convenient way of specifying actions with preconditions and effects. In what follows I will be simply using the term planning to refer to this approach.

Running Example

As a running example we will consider the scenario of a real-time strategy game where opposing players command virtual armies in battle against each other. The game involves gathering resources, training units, and engaging in combat in the usual way that is common to games in this genre. We will focus on the way a specific unit is controlled by the human player by issuing commands, and show how planning can be used to help a peasant think and plan his course of action in order to follow orders that involve reaching simple goals.

We will look at a very simple implementation of a peasant that can be instructed to find some food or defend the castle he serves, each of which may require him to perform a series of actions to achieve the given goal. We will assume that the castle is located in some forest area and buildings can be built inside the castle walls. In particular, an armory is a building where units can pick up and store spears, a kitchen is where raw food is cooked and converted to food rations, and a farmhouse is where rice can be harvested and converted to food rations.

The example is intended to demonstrate the basic ingredients needed in order to employ planning in the decision making process of a character in a video game. After going over these ingredients and describing some code that implements the planning decision procedure, we will discuss how this technique can be integrated in a real game setting in more exciting ways.

What is a Planning Problem?

In a planning problem we face the following challenge: given i) a description of the current state of the world, ii) a set of actions that describe how the world changes in terms of preconditions and effects, and iii) a goal condition, we have to find a sequence of actions such that when applied one after the other in the current state, they transform the state description into one that satisfies the goal condition.

This may sound a little confusing as in a video game we normally do not have a *description* of the current state of the game world. Instead, the state of the game is maintained internally by the game engine and we can get information about it using available functions. Similarly, a description of how this state changes when some action is executed seems unnecessary as the code describes it really well, implementing the changes right away.

Nonetheless, this is the type of information that will enable a character *deliberate* about the outcome of his actions *without them actually being executed* in the game world. By allowing the character think about the description of the current state and how it is transformed after a sequence of actions is executed, the character can look into his options and find out which course of action can achieve a given goal. We now proceed to see a simple yet convenient way to handle the description of the current state of the world, the preconditions and effects of actions, as well as the goal condition that we want our character to achieve.

A World Model as a List of String Labels

Going back to our running example and as far our peasant is concerned, a description of the current state of the world may be simply the information about what the peasant is holding in his hands and carrying in his backpack, and what his current location is. Let's assume that the peasant is located at the gates of the castle, holding nothing and carrying nothing, awaiting orders from the human player. This information is stored by the game engine in some appropriate way but let's also assume that it is also written down in the "brain" of the peasant as a list of string labels of the form:

```
["empty-hands", "empty-backpack", "at-gates"]
```

This reflects the current *world model* from the point of view of the peasant. Now let's see the available actions that the peasant can perform, and how each of them affects the world model.

Positive and Negative Effects of Actions as String Labels

Assuming that the castle has an armory built already then the peasant can go there and pick up a spear that he may use for hunting or defending the castle in emergency situations. So, one available action for the peasant is moving to the armory and another one is picking up a spear.

Let's name the first action `"moveto-armory"` and assume that whenever this action is executed by the peasant, he walks toward the armory until he gets there. In a sense, the effect of this action to the world model of the peasant as represented above by the string labels is that it removes the `"at-gates"` and adds the `"at-armory"` instead. We say that the first label is a p*ositive effect* of the action and the second label is a *negative effect* of the action. So, if the action `"moveto-armory"` is applied to the world model above, the peasant will have the following updated world model:

```
["empty-hands", "empty-backpack", "at-armory"]
```

Note that this update takes place only in the mental level of the peasant as no real action is executed in the game. This kind of deliberation is the basis for planning as it allows the character to play with these lists of string labels and figure out what his possibilities are before actually acting in the game.

Similarly, let's name the second action `"pickup-spear"` and assume that whenever this action is executed by the peasant, he picks up a spear provided he is located at the armory. In this case, a positive effect of the action is the string label `"holding-spear"` and a negative effect of the action is the string label `"empty-hands"`. So, if the action `"pickup-spear"` is applied to the world model above (that is, the one we get after applying the `"moveto-armory"` action), the peasant will have the following updated world model:

```
["holding-spear", "empty-backpack", "at-armory"]
```

Preconditions as String Labels

Note that the `"pickup-spear"` action is only executable in the game world if the peasant is actually located at the armory. This should also be reflected in the description of the action that we will use for planning purposes. This information can be easily handled using string labels that we will call *preconditions*. In order for an action to be applicable to a world model we will require that all the preconditions of the actions are present in the world model. For action `"pickup-spear"` the only precondition is the string label `"at-armory"` meaning that the peasant can pick up a spear only in those world models that he is located at the armory.

An Action Description as Three Lists of String Labels

In the general case an action may have more than one precondition or more than one positive and negative effect. In order to describe the preconditions and effects of actions in a simple and uniform way, we use three lists of string labels, one for the preconditions, one for the positive effects, and one for the negative effects. A description for the action `"pickup-spear"` would be something like the following:

```
Action name: "pickup-spear"
Precondition: ["at-armory"]
Positive effects: ["holding-spear"]
Negative effects: ["empty-hands"]
```

Similarly for any action we want the peasant to consider, we need to write down three lists that essentially describe when this action can be applied in a world model and what are the effects in the world model after the action is applied.

In order to check if an arbitrary action A can be applied to an arbitrary world model S, we simply need to go over the list of preconditions of A (each of which is a string label) and see if they are all included in the world model (which is also a list of string labels).

If A can be applied to S, in order then to generate the description of the world model S' that results from applying A on S, we just need to take the list of negative effects of A and remove all those labels from S, and then take the list of positive effects of A and add all those labels in S.

A Goal Condition (also) as a List of String Labels

There is only one more thing we need to specify for a planning problem, that is, the goal we want our character to achieve. In our running example the goal could be that the peasant successfully produces some food rations to be used for feeding the people living in the castle.

In terms of the world model that consists of string labels, satisfying the goal amounts to one or more string labels being present in the resulting goal model after a series of actions have been applied to the starting world model. In our case the goal condition might be just the singleton list `["carry-food"]`.

Planning: Thinking About Sequences of Actions and the Resulting World Model Before Acting in the Game World

A variety of actions are available to the peasant each of which transforms a world model in a particular way. Assuming that `"carry-food"` is a positive effect of one or more actions, then the peasant can look into the available actions and identify a sequence of actions such that when applied to the starting world model one after the other, they lead to a world model that includes `"carry-food"` (therefore satisfying the goal condition). As we will see shortly, a sequence of actions that achieves the goal is one that involves the peasant moving to the armory, picking up a spear, going to the forest, hunting for deer, and coming back to the castle to cook the raw meat and make food rations.

Note again that this process of deliberation takes place in the mental level of the peasant and no action actually takes place in the game. After the sequence of actions that achieves the goal is found, then the peasant can start executing each of these actions in the game one after the other until they are over or some new command is issued by the human player interrupting this plan.

Observe that this sequence of actions is more like a *high-level plan* rather than a sequence of atomic actions that is executed by the game engine. Each of these actions we consider in our running example may require implementing a simple strategy for acting accordingly based on scripting or the common approaches for reactive behavior such as a version of Finite State Machines or Behavior Trees.

Note also that the starting world model and the available actions for the peasant can be either provided by the game engine every time a command is issued by the human player or can be maintained for the peasant using appropriate data structures that are updated by the game engine automatically.

Searching for sequences of actions that lead to a world model that satisfies the given goal can be a tricky process. We will present a simple approach based on a depth-first-search manner and sketch how this can be adapted to meet our needs in different cases. In order to see the details of how a planning problem is solved, we will now look into an implementation for the planning decision process.

Implementation and Code Listings

The basic ingredients for planning are world models, actions, and goals. We will first go over a simple implementation that maintains and updates this information and then we will look into the kind of thinking we have discussed about the outcome of actions and achieving the goal condition.

I will be using Python to demonstrate a straightforward implementation of planning using list structures that contain string objects. The reason I chose Python is that using just the basic functionality of Python we can have a clean and concise implementation that looks a lot like the pseudo-code that I would write to demonstrate the main implementation ideas – and at the same time you can run it and experiment with the ideas.

Note that the code is not at all optimized as the intention is to show the basic functionality. I assume that anyone who is fluent in C++ or Java should have no problem in translating the example code into their programming environment, and also optimize the implementation so that their efficiency requirements are met.

WorldModel Class and Action Class

In Listing 1, you can see the class *WorldModel* that essentially consists of a list of facts which for this example will be simple string objects. Note that the list is implemented so that adding a duplicate entry or removing a fact that is not in the list produces no error but leaves the list unchained. These methods are used to transform the world model when some action is applied at the mental level.

Listing 1: *A world model is described as a simple list of facts in class WorldModel*

```
class WorldModel:
  # A world model is initialized with a list of facts
  def __init__(self, facts):
    self.facts = facts

  # Adding a fact in a world model is simply adding it
  # in the internal list of facts unless already present
  def add_fact(self, fact):
    if fact not in self.facts:
      self.facts.append(fact)

  # Removing a fact in a world model is simply taking it
  # out of the internal list of facts if present
  def remove_fact(self, fact):
    if fact in self.facts:
      self.facts.remove(fact)
```

This is shown in Listing 2 where the first method updates the world model by adding the positive effects of a given action and removing the negative effects of the action.

The second method first creates a duplicate world model and applies the changes there. This is called the successor word model. Similarly, Listing 3 shows how to test whether a world model satisfies a goal. Finally, Listing 4 shows the class Action.

Running Example

Listings 5 and 6 show how our running example may be implemented using the classes we defined. We use simple strings to represent facts about the game from the perspective of the peasant, and create an initial world model where the peasant is not holding or carrying anything and he is ready to move. A goal condition is specified as a list that includes only the desired fact the peasant is carrying some food rations. A series of actions are then defined using four arguments, the first one being the name of the action, and the other three being the preconditions, positive effects, and negative effects of the action represented as lists of facts.

The peasant can move to five different areas (the gates, the kitchen, the armory, the farmhouse, and outside of the castle to the surrounding forest) using the corresponding command as long as the `"ready-to-move"` fact is present in world model he is considering. This precondition is added so the peasant first applies a `"new-destination"` action before moving that removes from the world model any fact describing his current position.

The available actions suggest that the agent can achieve the goal by cooking some raw meat, but a few actions need to be performed before cooking can take place. As the example is very simple, we can see that this involves picking up a

Listing 2: *The methods (also part of the class WorldModel) for updating a world model with respect to a given action and creating successor model*

```
# Checks if an action can be executed in the world model
def can_execute(self, action):
  # An action can be executed in a world model if the list of
  # facts called preconditions are present in the world model
  for fact in action.preconds:
    # If any fact in the action list of preconditions is
    # absent then return false
    if fact not in self.facts:
      return False
  # If all facts in the action list of precondition are
  # present in the world model then return true.
  return True

# Executes an action in the world model updating the facts
# of the model
def execute(self, action):
  # When an action is executed it alters the world model by
  # removing from the model the negative effects of the action
  for fact in action.neg_effects:
    self.remove_fact(fact)
  # and inserting the positive effects of the action.
  for fact in action.pos_effects:
    self.add_fact(fact)
```

Listing 3: *The method (also part of the class WorldModel) for testing whether a world model satisfies a given goal*

```
# A goal is satisfied by the world model if all facts
# in the goal list are present in the world model
def satisfies_goal(self, goal):
  for fact in goal:
    if fact not in self.facts:
      # If any fact in the goal list is absent then
      # return false
      return False
  # If all facts in the goal list are present in the
  # world model then return true.
  return True
```

Listing 4: *An action is defined as three lists of facts describing the preconditions, the positive effects and the negative effects of the action*

```
class Action:
  # An action is defined by a name, and three lists of facts:
  # a list of preconditions, a list of positive, and a list
  # of negative effects
  def __init__(self, name, preconds, pos_effects, neg_effects):
    self.name = name
    self.preconds = preconds
    self.pos_effects = pos_effects
    self.neg_effects = neg_effects
```

Listing 5: *Initial world model, goal, and actions for our running example*

```
initial_model = WorldModel(["empty-hands",
                            "empty-backpack",
                            "ready-to-move"])
goal = ["carry-food"]

actions = []

actions.append( Action("pickup-spear",
                       ["at-armory", "empty-hands"],
                       ["hold-spear"],
                       ["empty-hands"]) )

actions.append( Action("store-spear",
                       ["at-armory", "hold-spear"],
                       ["empty-hands"],
                       ["hold-spear"]) )

actions.append( Action("hunt-deer",
                       ["at-forest", "hold-spear", "empty-backpack"],
                       ["carry-rawmeat"],
                       ["empty-backpack"]) )

actions.append( Action("cook-rawmeat",
                       ["at-kitchen", "carry-rawmeat"],
                       ["carry-food"],
                       ["carry-rawmeat"]) )
```

Listing 6: *Additional moving action for our running example*

```
actions.append( Action("new-destination",
                       [],
                       ["ready-to-move"],
                       ["at-forest", "at-gates", "at-armory",
                        "at-kitchen" "at-farmhouse"]) )

actions.append( Action("moveto-armory",
                       ["ready-to-move"],
                       ["at-armory"],
                       ["ready-to-move"]) )

actions.append( Action("moveto-kitchen",
                       ["ready-to-move"],
                       ["at-kitchen"],
                       ["ready-to-move"]) )

actions.append( Action("moveto-gates",
                       ["ready-to-move"],
                       ["at-gates"],
                       ["ready-to-move"]) )

actions.append( Action("moveto-forest",
                       ["ready-to-move"],
                       ["at-forest"],
                       ["ready-to-move"]) )

actions.append( Action("moveto-farmhouse",
                       ["ready-to-move"],
                       ["at-farmhouse"],
                       ["ready-to-move"]) )
```

spear and hunting for deer. The question, though, is how the peasant can find such a solution for himself, and more importantly how he can do this for any kind of goal and any kind of situation he may be.

Finding a Solution to a Planning Problem Using Depth-first Search

Solving a planning problem essentially involves searching among the potential sequences of actions that can be performed in order to find one that results in a world model that satisfies the goal. The easiest way to do this (but not the most efficient) is to do an exhaustive search among all possibilities. One way to do this is as follows:

- We start from the world model that reflects the current state of the game (which we will call the initial world model).
- We check if the world model already satisfies the goal. If it does we are done, otherwise we continue.
- We look into all the available actions and see which ones can be applied to the world model, and for each one that can be applied we produce the successor world model.
- We pick one of these successor models and loop back to the beginning.

This is very similar to AI depth-first search, only slightly more complicated since the successor steps are not given (e.g., by some graph) but we need to produce them at each iteration by looking into the preconditions and effects of the available actions.

Listing 7 shows an implementation that relies on an open list which keeps track of the possibilities that we haven't checked yet in terms of partial plans, and also uses an upper-bound on the depth of the search so that we don't end up

searching forever for a goal that cannot be achieved.

A partial plan (the corresponding class is shown in Listing 8) is simply a list of actions that we have considered so far along with the world model that we get after applying them on the initial world model. So, at any given point we keep track both of the sequences of actions we have considered and the world model that they take us after executing them. In this way, when we reach a world model that satisfies the goal we will also know the sequence of actions that led there and report this as a solution to the planning problem.

Listing 9 shows the result of running the search procedure for our running problem using an upper-bound of 8 actions. The search procedure finds that there is an appropriate sequence of actions that achieves the given goal.

Beyond Depth-first Search

Using an exhaustive search procedure to search for a given goal can be problematic as the number of actions and facts increase in a real implementation. Moreover, as this approach does not aim for finding the shorter action sequence that achieves the goal, we may encounter unintuitive solutions where repeated steps and unnecessary moves take place. One solution to this is to appeal to a search method similar to A*.

A straightforward approach is to use the number of actions in the partial plan as the cost of the plan, and the simple heuristic of counting the remaining sub-goals (that is, facts in the goal condition that are not in the model of the partial plan) as the estimation of the remaining cost. The search procedure can then keep a list of partial plans that are sorted based on the estimated total cost for the plan, and always pick the most promising one to proceed.

Listing 7: *Solving a planning task using a depth-first search method with a bounded depth*

```
class PlanningTask:
  def __init__(self, initial_model, available_actions, goal):
    self.initial_model = initial_model
    self.available_actions = available_actions
    self.goal = goal

  def depth_first_search(self, bound):
    # Initialize search
    node = PartialPlan([], self.initial_model)
    open_nodes = [node]

    # As long as there are nodes in the list of open nodes
    # proceed with searching for a solution to the planning task
    while open_nodes:
      # Take the last node from the list of open search nodes
      node = open_nodes.pop()

      # If the model of the partial plan satisfies the goal then
      # the corresponding actions that led to this world model is
      # a solution to the planning task
      if node.model.satisfies_goal(self.goal):
        return node.actions

      # If the actions of the partial plan have already reached
      # the maximum number of actions specified by the bound
      # then do not do any more search for this partial plan
      if len(node.actions) == bound:
        continue

      # Otherwise generate all the successor world models using
      # those actions that can be executed in the model of the
      # partial plan and add them at the end of the open list of
      # nodes as new partial plans
      for action in self.available_actions:
        if node.model.can_execute(action):
          successor = node.model.create_successor(action)
          actions = list(node.actions)
          actions.append(action.name)
          open_nodes.append(PartialPlan(actions,successor))

      # That's all, now loop back to check the list of open nodes

    # If the list of open nodes is empty then there is no
    # solution for the planning task with a number of actions
    # that does not exceed the given bound.
    return False
```

Listing 8: *A partial plan is simply a list of actions and the resulting world model after action execution*

```
class PartialPlan:
  # A partial plan holds a list of actions that have been
  # executed so far and the resulting world model
  def __init__(self, actions, model):
    self.actions = actions
    self.model = model
```

Listing 9: *Using the PlanningTask search functionality to find a solution for the planning problem of our running example*

```
task = PlanningTask(initial_model, actions, goal)
plan = task.depth_first_search(8)
print "Plan: ",
print plan

>>>

Plan: ['moveto-armory', 'pickup-spear',
       'new-destination', 'moveto-forest',
       'hunt-deer', 'new-destination',
       'moveto-kitchen', 'cook-rawmeat']
```

Nonetheless, there are many other ways that this search procedure and the planning decision process can be adapted in order to account for the particular needs of each case. For example, one might want to search for that particular sequence of actions that maximizes some utility metric (e.g., the happiness of the character in the game) instead of planning for a particular goal. In this case we could define a function that calculates the value of the intended utility metric for any arbitrary state and use this as a guide for the search procedure.

Similarly, depending on the game and the intended result we could combine all of the above ideas to make the search procedure work for us and look for sequences of actions with the intended qualities.

Why is Planning Useful?

We have looked into planning assuming that searching for sequences of actions and planning for a goal condition is a useful tool as a decision making procedure. In the running example that we saw, we ended up finding a solution to the planning problem that is straightforward and that could have been addressed using a more common approach for decision making. In fact, it seems like an overkill to use a search procedure in order to find simple solutions like this.

First note that planning is not some magic technique that can achieve types of behavior that cannot be expressed otherwise. That is, for any sequence of actions that a planning decision procedure comes up, a game developer could use some of the common decision making techniques for achieving the same behavior. So, what do we gain with planning? The answer is *flexibility*.

As the game elements increase to large numbers and available interactions in the game become more complex, there is a trade-off to be considered. On the one side we have a *planning decision making* approach where we only need to specify the way things interact and the intended goals that a character should pursue (which also requires costly search in order to realize the behavior). On the other side we have all the common *reactive decision making* approaches that require more effort in order to cover all the interesting interactions that the character should be able to handle (which require no searching and work lightning fast).

So far the reactive side of the trade-off has been working perfectly well for most games, but there are also cases where moving toward the planning side has some benefits too. In any case, planning is just another tool that can be used in any combination is beneficial for a game developer at a particular time. A useful rule of a thumb is that normally planning is better used for tactical decision making or what I have been referring to as "high-level plans" instead of time-critical responses (even though there have been some notable exceptions).

Listing 10: *Using the PlanningTask search functionality to find a solution for the planning problem of our running example after a new action is available to the peasant*

```
actions.append( Action("harvest-rice",
                       ["at-farmhouse", "empty-backpack"],
                       ["carry-food"],
                       ["empty-backpack"]) )

task = PlanningTask(initial_model, actions, goal)
plan = task.depth_first_search(8)
print "Plan: ",
print plan

>>>

Plan:  ['moveto-farmhouse', 'harvest-rice']
```

A Case for "Smart" Peasants Using Planning

Closing, I would like to sketch how planning could be used to extend the running example to a real game setting where smart peasants are instructed to do more demanding tasks like finding food in any of the possible ways, instead of the usual "get this type of resource" or "repair this building".

Imagine for instance that the game of our running example allows the human player to buy all sorts of items to be used by the units of the castle in order to carry out the commands of the human player. As a result there may be many ways that a peasant can find food, including harvesting rice using the appropriate equipment, processing wheat to make flour and then baking bread by taking each resource to the appropriate building, hunting for animals, and many others depending on the number of available items and buildings and the complexity of the interactions allowed. As a quick example, Listing 10 shows how adding another available action for the peasant leads to an alternative plan for finding food.

In this scenario, a combination of reactive decision making and planning decision making could be beneficial for a game developer. One way to go would be to write down how each item and each building can be used in terms of high-level actions with preconditions and effects (such as "use mill" or "bake bread"). When a command is given to the peasant then planning can be performed

in order to find a sequence of actions that achieve the given goal, and then smaller-scale reactive decision making can be performed in order to successfully execute each of these actions.

Now consider a case where *more than one peasant* can be instructed to pursue a *common* goal. Planning can be used in this case as well with only minor modifications, allowing the discovery of plans where each peasant takes care of a different task. Now consider building upgrades like a college and a university that essentially upgrade the upper-bound allowed in planning in order for the peasants to be able to perform more complex plans. Now consider this type of behavior in all types of units of the game. Now consider assigning commands to multiple types of units with different capabilities…

Examples and Code

The code and examples that I mentioned in this article can be found at my webpage: http://stavros.lostre.org/pyPlan/

Stavros Vassos

Stavros Vassos received his B.Sc. degree in Electrical and Computer Engineering from the National Technical University of Athens in Greece, and his M.Sc. and Ph.D. degree in Computer Science from the University of Toronto in Canada. His research interests lie mainly in the area of Artificial Intelligence, in particular logic-based approaches for Knowledge Representation and Reasoning, Reasoning about Action and Change, and Intelligent Agent Design. During his studies in the University of Toronto he was involved in the development of IndiGolog, a high-level agent programming language for cognitive robotics. Recently he has turned his attention to how similar AI techniques can be applied to the challenging application domain of video games.

Webpage: http://stavros.lostre.org

Twitter: @StavrosVassos