pyPlanGenius

Generating Total Order Plans and Extracting Partial Order Plans from them.

16.412 Final Project

May 15, 2013

James Paterson

[redacted]@mit.edu

Enrique Fernandez

[redacted]@mit.edu

Contents

1 Introduction and Motivation

Robots are becoming cheap and more common in the work place and at home. They can perform tasks that humans can't or don't want to do and they can work continuously without having to stop for lunch or sleep. However, most robots can only to programmed to perform tasks by trained experts. Moreover, there is still a large separation between humans and robots. In order for us non-experts to program robots and work side-by-side with them, we need an easier way to command them to perform a task.

One way of doing this is to teach the robot a plan through demonstration. Once taught, we do not want the robot to simply repeat the trajectory demonstrated (as a motion capture approach would do), but we want the robot to generalize the plan, so that it can adapt to different situations and still complete the task.

An example of a robot that can learn a plan is the Baxter robot from ReThink Robotics. It is revolutionary because it is cheap (at $22,000) and it can be taught to perform a task by a human demonstrator without any programming experience. Plan learning is an important part of making robots like this easy to use and thus help them gain popularity.

2 Project Overview

The idea for this project came from a lecture we gave on plan learning [Fernandez and Paterson 2013]. In order to give a background for the project, this section will give a brief recap of the lecture and then will define the scope of the project.

2.1 Lecture Recap

The lecture showed a method to learn a reactive task specific plan from robot behaviors and object types (see figure 1). The lecture was split into three main sections:

1. **Learning action definitions:**
   First, a teacher demonstrates which robot behavior and object type to use. These are combined by the robot into a task specific action. Task specific actions are performed by the robot and the state of the world before and after execution is recorded in a list. From this list, the robot extracts PDDL action definitions with preconditions and effects. Timed actions are not considered.

2. **Learning action sequences:**
   An action sequence is a sequence of actions from the initial to the goal state. In this section we showed how a human can demonstrate actions to a robot in a specific order to form an action sequence. The generated action sequence is a totally ordered plan from the initial to the end state.

3. **Learning plans:**
   From the action sequence, which is a totally ordered plan, we showed how to find causal links and extract a partially ordered plan. Threats needed to be resolved to form a valid partially ordered plan. We then showed how to detect and merge loops in the plan, before detecting repeat-until loops that would add robustness to the plan. A general task specific plan was then extracted from the above.

2.2 Project Scope

This project follows the lecture method described above closely. However we assume that step 1 has already been performed. In other words, we start off with non-timed PDDL actions containing preconditions and effects. We then follow the method described above until we have extracted partial order plans from total

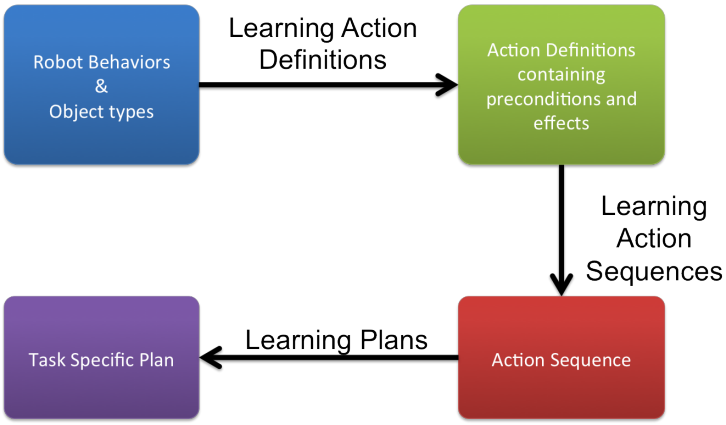


Figure 1: Overview over lecture on Plan Learning.

order plans and resolved threats. Partial order plans are useful as the allow for more flexibility to the robot than total order plans w.r.t. the order of performing actions.

In order to parse the PDDL actions into classes that we could use, we had to write a parser to parse two PDDL files (a *domain* and *problem* file). Apart from parsing, we added many useful classes and methods. Next, in order to learn an action sequence from demonstration, we wrote a user interface so the teacher could interact with the robot. This user interface includes error detection, text completion and colors. It also only allows the user to select from actions and objects that are allowed given the state of the world. After this, we implemented an algorithm to extract a partial order plan from a total order plan by detecting causal links and resolving threats.

The structure of the project is as follows:

We start with an introduction to plan learning and a motivation for it.Then we discuss how to use PDDL action representations and demonstrations from a human teacher to learn an action sequence to get from the initial state to the goal. This action sequence is a total order plan and in the next section, we find a partial order plan from it by extracting causal links and resolving threats. We also discuss limitations of our algorithm for extracting partial order plans. Lastly, we conclude and discuss lessons learnt from the project. Appendices show references and how the code can be used.

3 Generating Action Sequences

3.1 Section Overview

An action sequence is a sequence of actions from the initial to the goal state. In this section we start with pddl actions and show how a human can demonstrate actions to a robot in a specific order to form an action sequence. The generated action sequence is a totally ordered plan from the initial to the goal state. It is important to note that no planning algorithm is used in this approach.

3.2 Method

In learning the action sequence, the robot starts with ungrounded actions in the PDDL representation, then through demonstration by a human teacher, grounds the action type and parameters. The robot then

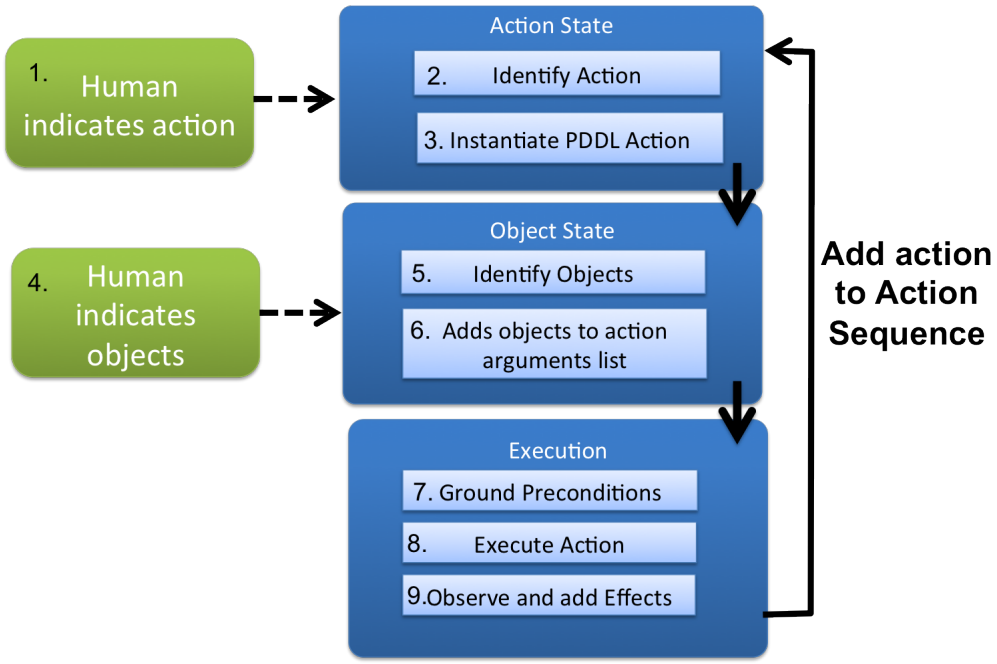


Figure 2: Interaction between a human teacher (green) and robot (blue) to form an action sequence.

performs this action and adds the grounded action with grounded preconditions and effects to the action sequence. This process is repeated until the goal state is reached. This method comes from [Veeraraghavan and Veloso 2008].

The algorithm is shown in graphical form in figure 2.

Let's walk through this algorithm step by step.

1. In the first step, the human teacher indicates which action he/she wants the robot to perform. In this project, this is done by entering the action name in the user interface. For example, the user might enter:
   hammer-nail
2. The robot identifies which action the human had indicated.
3. The robot collects this action from its known list of PDDL operators. For example:
   :action hammer-nail
   :parameters (?h - hammer ?n - nail)
   :precondition (and (has ?h)(has ?n))
   :effect (and (hammered ?n)(not(has ?n)))
4. Next, the teacher indicates which objects to perform the action on e.g.
   hammer and **nail**
5. The robot identifies the objects.
6. The robot adds the objects to the action arguments list:
   :action hammer-nail

:parameters (hammer - hammer nail - nail)
:precondition (and (has ?h)(has ?n))
:effect (and (hammered ?n)(not(has ?n)))

7. The robot grounds the action preconditions:
   :action hammer-nail
   :parameters (hammer - hammer nail - nail)
   :precondition (and (has hammer)(has nail))
   :effect (and (hammered ?n)(not(has ?n)))

8. Robot executes the action.

9. The robot observes and grounds the effects of the action:
   :action hammer-nail
   :parameters (hammer - hammer nail - nail)
   :precondition (and (has hammer)(has nail))
   :effect (and (hammered nail)(not(has nail)))

10. The grounded action is added to the action sequence.
    Back to Step 1.

At the end of this process, we should have an action sequence going from the initial state to the goal state, provided the user has taught the robot the correct sequence. An example of an action sequence is shown in figure 3.

Two major pieces are needed in order to demonstrate an action sequence to the robot in our project. First, a parser is needed to parse the pddl-domain and the pddl-world files into python classes that are used extensively in the project. Then a user interface is needed so that the human teacher can interact with the robot. These two pieces are described in more detail in the sections that follow.

3.3 Parser and Classes

We wrote a parser to parse non-timed actions in PDDL and provide the classes and many useful methods we needed for the project.

Two files are parsed by our parser: a *domain* and *problem* file. The *domain* file defines actions and predicates, while the *problem* file defines the objects, object types, initial and goal conditions.

While all classes generated by the parser can be seen in the code, the most important ones are summarized below:

- **Operator Schema:** An *OperatorSchema* is the most general form of a PDDL action containing the action name and ungrounded parameters, preconditions and effects.

  A string representation of an operator schema might look like:

  Operator Schema: find-object
  Parameters: ?o-object
  Preconds: (stored ?o)
  Add Eff: (located ?o)

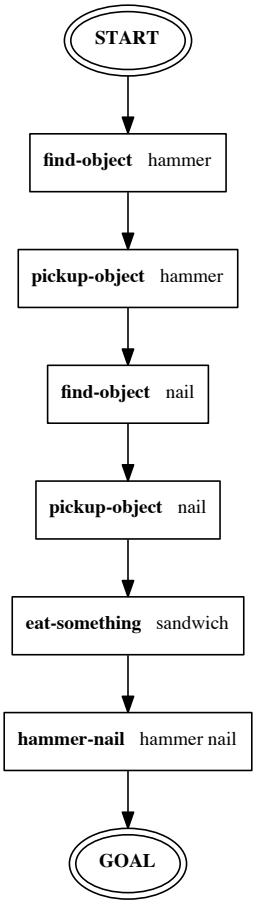


Figure 3: Generated action sequence.

Del Eff: (stored ?o)

- **Parameter:** A *Parameter* contains a tag and a parameter type. In the string representation of a parameter: (?o - object),
  ?o is the tag and it is of type object.
- **Predicate:** A *Predicate* has a property and an argument. In the three examples of *Predicates* below,the properties will correspond to *has*, *stored* and *hammered*, while the arguments correspond to ?*o*, ?*o* and ?*n*.

  (has ?o)
  (stored ?o)
  (hammered ?n)

  Predicates can occur in a *State*, as well as in the preconditions and effects of *OperatorSchemas*. *Predicates* are grounded when an *OperatorSchema* is grounded to an *ActionStep* (see ActionStep below).
- **State:** A *State* is a set of grounded predicates that are true at some time step in the world. Initial and goal conditions are a set of ground *Predicates* that from a *State*. For example:

  Current State: (stored nail) (located hammer)
- **Action Step:** An *ActionStep* is formed when we ground an *OperatorSchema* with arguments. The preconditions and effects of the *ActionStep* are also grounded in this process.
  For example, $ActionStep(find - object, [hammer])$ will yield:

  Action Step: (find-object hammer)
  Preconds: (stored hammer)
  Add Eff: (located hammer)
  Del Eff: (stored hammer)
- **Action Sequence:** An *ActionSequence* is an ordered list of *ActionSteps*. It also contains the initial state and end state.
  An example of a string representation of an action sequence can be seen below:

  Action Sequence:
  Initial Conditions: (stored hammer) (stored nail)
  Step 0: (find-object hammer)
  Step 1: (pickup-object hammer)
  Step 2: (find-object nail)
  Step 3: (pickup-object nail)
  Step 4: (hammer-nail hammer nail)
  Goal Conditions: (hammered nail)

  Keep in mind that each step in the action sequence is an Action Step, and thus we can find the grounded preconditions and effects of each step in the action sequence.

A number of functions are contained in the classes defined above. The most interesting of them will be explained when they are relevant in the following sections.

3.4 Important Methods

3.4.1 findValidActions()

When interacting with the robot, we would like to present the user with the option to choose from only the actions that are valid at a particular time step. To find the possible valid ungrounded actions at a timestep, we pass the current state, all operator schemas from the PDDL domain and objects from the problem file into the findValidActions() function that is located in the State class.

The algorithm for this method is described in algorithm 1. For every operator schema in the domain, this function performs depth-first search (method SEARCH_IF_VALID(), section 3.4.2) to ground the parameters in operator schema and check for validity given the current state. A list of all valid operators is returned.

Algorithm 1: FIND_VALID_ACTIONS finds all ungrounded actions that are valid given the current state

Input: All operator schemas *Ops*, all objects $\mathcal{O}$, and the current state $\mathcal{S}$.
Output: A list of valid Operator Schemas (a.k.a ungrounded Actions)

```
validOps = []
for op in Ops do
    paramtypes = get_param_types(op)
    if SEARCH_IF_VALID(op,S,O,paramtypes) then
        validOps.append(op)
return validOps
```

3.4.2 searchIfValid()

This method checks if there is at least one combination of objects that make an operator schema valid given the current state. The search method SEARCH_IF_VALID() can be visualized in figure 4. A new Node is instantiated for every parameter type in the operator. In this example, the $hammer - nail$ operator takes two types of parameters as arguments: hammer and nail. The depth of each search tree is dependent on the number of parameter types in the operator schema.

Each Node contains a list of Objects of that parameter type. Objects a and b are of type *hammer* and objects 1,2 and 3 are of type *nail*.

The search then finds all possible combinations between Objects of different types. For each combination, it grounds the operator schema by creating an *ActionStep* and then checks for validity given the current state using $isFeasibleActionStep()$ (see section 3.4.3). If $isFeasibleActionStep()$ returns $True$, it means that there is at least one combination of that operator schema and available objects that the teacher can enter that would be correct given the current state.

3.4.3 isFeasibleActionStep()

The method $isFeasibleActionStep()$ checks whether an *ActionStep* can be performed given the current state of the world. An *ActionAtep* is a grounded action with grounded preconditions and effects. This method return $True$ if the set of predicates in the precondition of the action step is a subset of the set of predicates in the current state of the world.

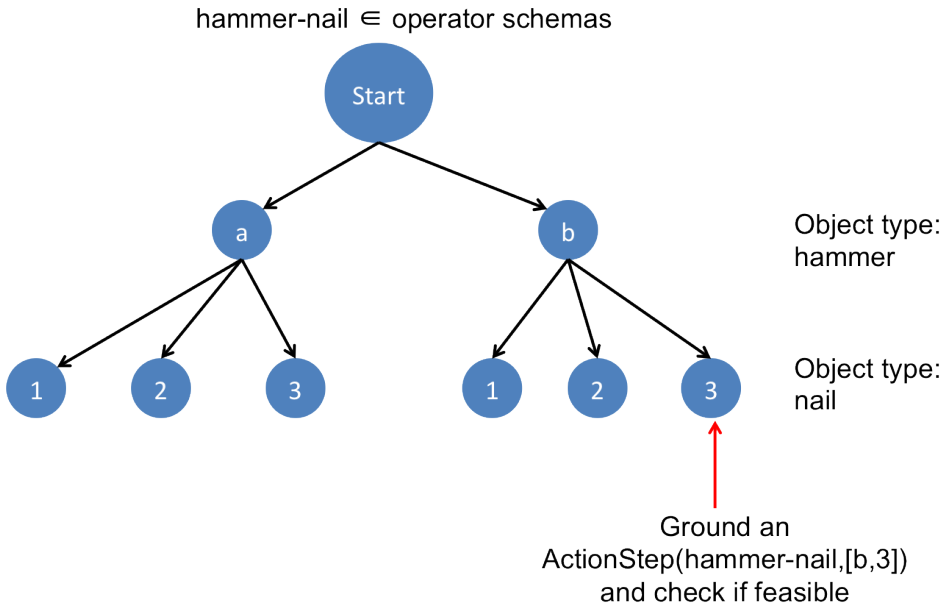


Figure 4: Visualization of the search method used to check for valid operating schemas.

3.5 User Interface

The user interface is needed for the human demonstrator to communicate with the robot and tell it which action to perform, which objects to perform the action on and the order of performing actions.

The interface between the human and the robot takes place in the terminal. An example of the user entering one action in the action sequence can be seen in figure 5.

In order to make the interaction between human and robot more fluid, the user is only presented with the ungrounded actions (operator schemas) that are possible given the current state of the world. This is done by using the $findValidActions()$ method in the State class (section 3.4.1). In the screenshot of the interface below, the only valid actions given the current state of $((hashammer) \wedge (hasnail))$ are $hammer - nail$, $eat - something$, and $store - object$. These are presented as options to the user:

```
Current State:
  (has hammer) (has nail)

Please choose action5 to perform from the list below:
 hammer-nail
 eat-something
 store-object
>>hammer-nail
```

After the user has chosen the operator schema ($hammer - nail$ in this example), we only want to present objects that are of the correct type for the operator schema chosen. Since each parameter has a tag and a type, we present the user with the objects associated with that tag's object type. In the example below, we see that the user is only presented with the hammer object, when it is the only object of type hammer that is relevant for the $(?h - hammer)$ parameter.

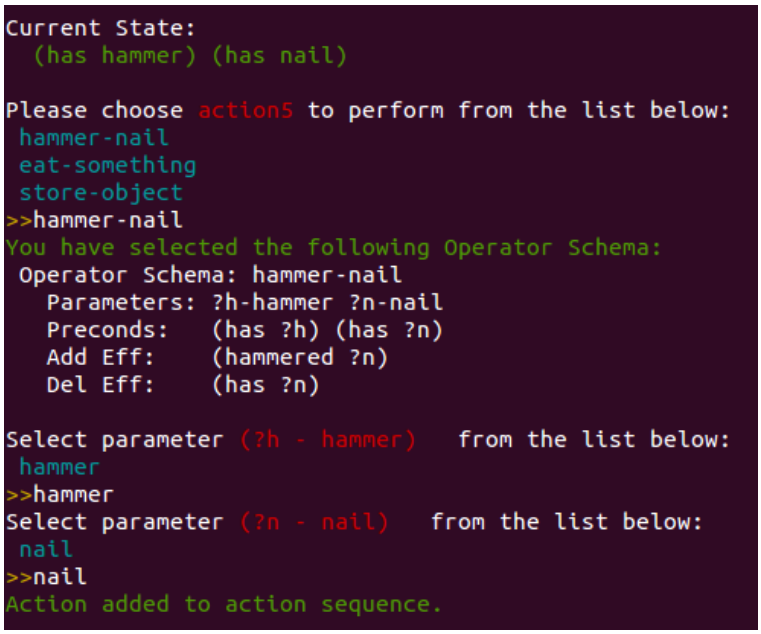
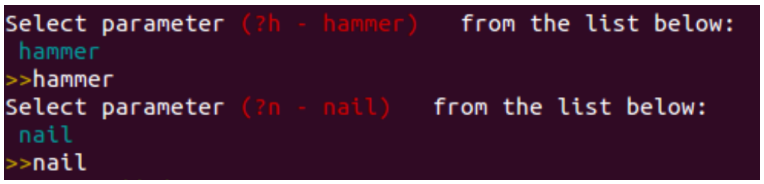


Figure 5: User interface used by a human teacher to interact with a robot.



Colors and auto-completion were also added to the interface to make the interface easy to use and error detection ensures that the user cannot add incorrect actions to the action sequence. An example of error detection in the *balls* domain is shown below, where the combination of action and arguments is not feasible given the current state of the world:

```
Current State:
  (empty-gripper pr2) (found pr2 yellowball)

Please choose action2 to perform from the list below:
 find-ball
 pickup-ball
>>pickup-ball
You have selected the following action:
 Operator Schema: pickup-ball
   Parameters: ?r-robot ?b-ball
   Preconds:   (found ?r ?b) (empty-gripper ?r)
   Add Eff:    (holding ?r ?b)
   Del Eff:    (empty-gripper ?r)

Select parameter ( ?r - robot )   from the list below:
 pr2
>>pr2
Select parameter ( ?b - ball )   from the list below:
 yellowball
 greenball
>>greenball
ERROR: Combination between action and arguments not feasible.
Action NOT added to action sequence.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Current State:
  (empty-gripper pr2) (found pr2 yellowball)

Please choose action2 to perform from the list below:
 find-ball
 pickup-ball
>>
```

4 Extracting Partial Order Plans

4.1 Overview

In section 3 we have seen how the user can teach the robot how to perform a task by building an **Action Sequence** or *total order plan*. The goal of our lecture (Fernandez and Paterson 2013) was to then use this action sequence to learn a *domain specific* plan that the robot can apply in new situations to solve the same type of problems. In our lecture, we described that, in order to learn a domain specific plan, we had to do three steps: 1) Extract the annotated partial order plan, 2) Detect and merge loops and 3) Detect repeat until loops (Figure 6).

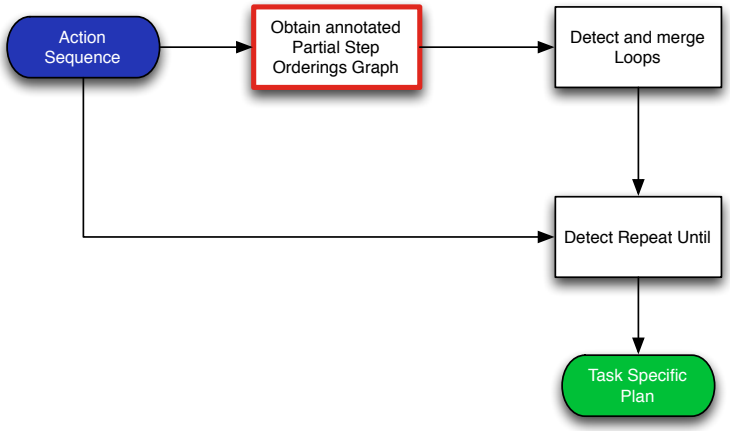


Figure 6: Process for learning a domain specific plan from an action sequence

For this project we have decided to focus on the first step of the process highlighted in figure 6 : extracting annotated partial order plans.

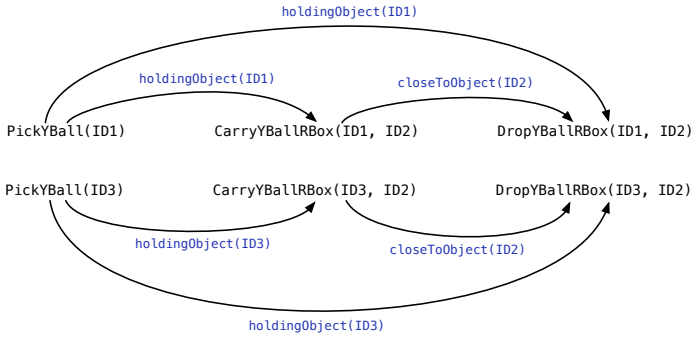


Figure 7: Example of a partial order plan

Intuitively, an annotated partial order plan is a directed graph that makes ordering constraints between actions explicit. Figure 7 shows an example of a partial order plan in which the objective is to pick two balls (`YBall1` and `YBall2`), carry them to the box and drop them there. We can see in the graph that there are to edges that link action `DropYBallRBox(ID1, ID2)` with actions `CarryYBallRBox(ID1, ID2)` and `PickYBall(ID1)`. These edges indicate that in order to drop `YBall1` in the box we first need to pick it

up and carry it to the box and we will refer to them in the future as *causal links*. It is important to note that there are not any causal links between the actions applied on `YBall1` and the actions applied on `YBall2`. This indicates that the order in which we process `YBall1` and `YBall2` is not relevant for the success of this plan.

In this section we first discuss an algorithm for extracting annotated partial order plans from total order plans presented in Winner and Veloso 2002 and show several examples. We then introduce a modification to the algorithm needed to ensure that the generated partial order plans are consistent even in the presence of threats. Finally we discuss some limitations of this algorithm and direct the reader to references that deal with these problems.

4.2 Algorithm for finding MACPOs

Algorithm 2: EXTRACT_ANY_MACPO finds a minimal annotated partial order plan.

Input: A totally ordered plan consisting of a sequence of action steps $\mathcal{T} = S_1, ..., S_n$, the initial conditions and the goals.

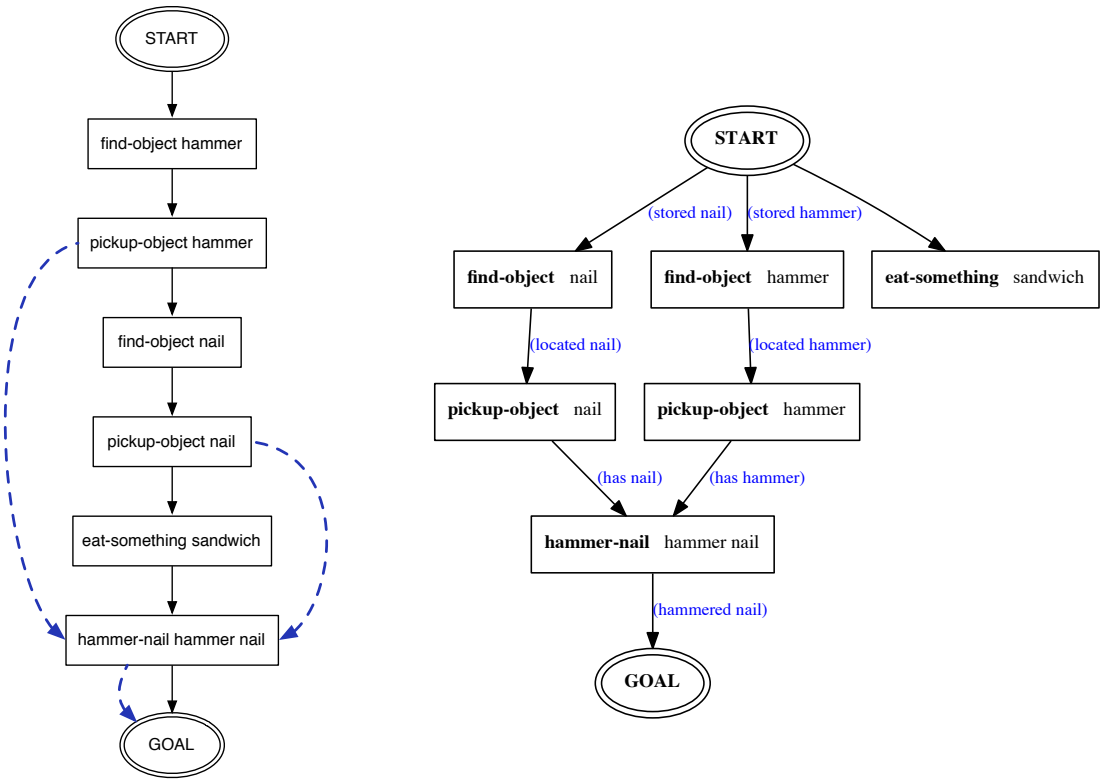
Output: A minimal annotated partial order plan given as a directed graph in which edges are causal links.

$START\_step \leftarrow build\_action\_step(None, initial\_conditions)$
$GOAL\_step \leftarrow build\_action\_step(goals, None)$
Insert $START\_step$ at the beginning of $\mathcal{T}$
Insert $GOAL\_step$ at the end of $\mathcal{T}$
for $S_i \leftarrow S_{n+1}$ ***downto*** S_1 **do**
    for P_{ij} *in preconditions of* S_i **do**
        $S_j \leftarrow$ find_last_producer(P_{ij})
        Add_Causal_Link($\mathcal{P}$, S_j, S_i, P_{ij})
return $\mathcal{P}$

Algorithm 2 shows the algorithm that finds an annotated partial order plan that was implemented in this project and that is presented in Winner and Veloso 2002. The algorithm takes as input a totally ordered plan or action secuence as defined in section 3 ($\mathcal{T} = S_1, ..., S_n$), the initial conditions and the goals. The output of the algorithm is a minimal annotated partial order plan described as a directed graph in which edges represent causal links and nodes represent plan actions.

Intuitively, the algorithm iterates through each action in the action sequence starting from the goal and through each precondition of that action sequence and looks for the *last* action that produces that precondition. Then it adds a *causal link* from the action that produces the precondition to the action that consumes it to the graph. This causal link indicates that the *consumer* action needs to happen after the *producer* action.

As an example, consider figure 8. In 8a in black we can see the action sequence whose partial order plan we want to extract. On the right we can see the partial order plan that was extracted from this sequence. Note that the algorithm first creates a START action whose effects are the starting conditions of the demonstration and adds it to the beginning of the action sequence (3). In the same way, a GOAL action whose preconditions are the goals is added to the end of the sequence. Then, the preconditions of the last action in the sequence (the GOAL action initially) are extracted and the sequence is traversed back from the end in order to find the last action that generated those preconditions. In this case, the goal is `hammered nail`, that is generated by the action sequence `(hammer-nail hammer nail)`. Therefore, a `Causal_Link((hammer-nail hammer nail), GOAL, (hammered nail) )` is added to the graph. In the same way, the algorithm now extracts the preconditions of the previous action and discovers that in



(a) In black, action sequence or total order plan generated from the user demonstration. In blue, the first three causal links discovered by algorithm 2

(b) Extracted partial order plan (output of algorithm 2)

Figure 8: Example of a total order plan and its correspondent partial order plan.

order to hammer the nail we first need to have the nail and the hammer and therefore two new causal links are added to the graph. These first three causal links are displayed in blue in 8a. After extracting all causal links, the resulting partial order plan can be seen in 8b.

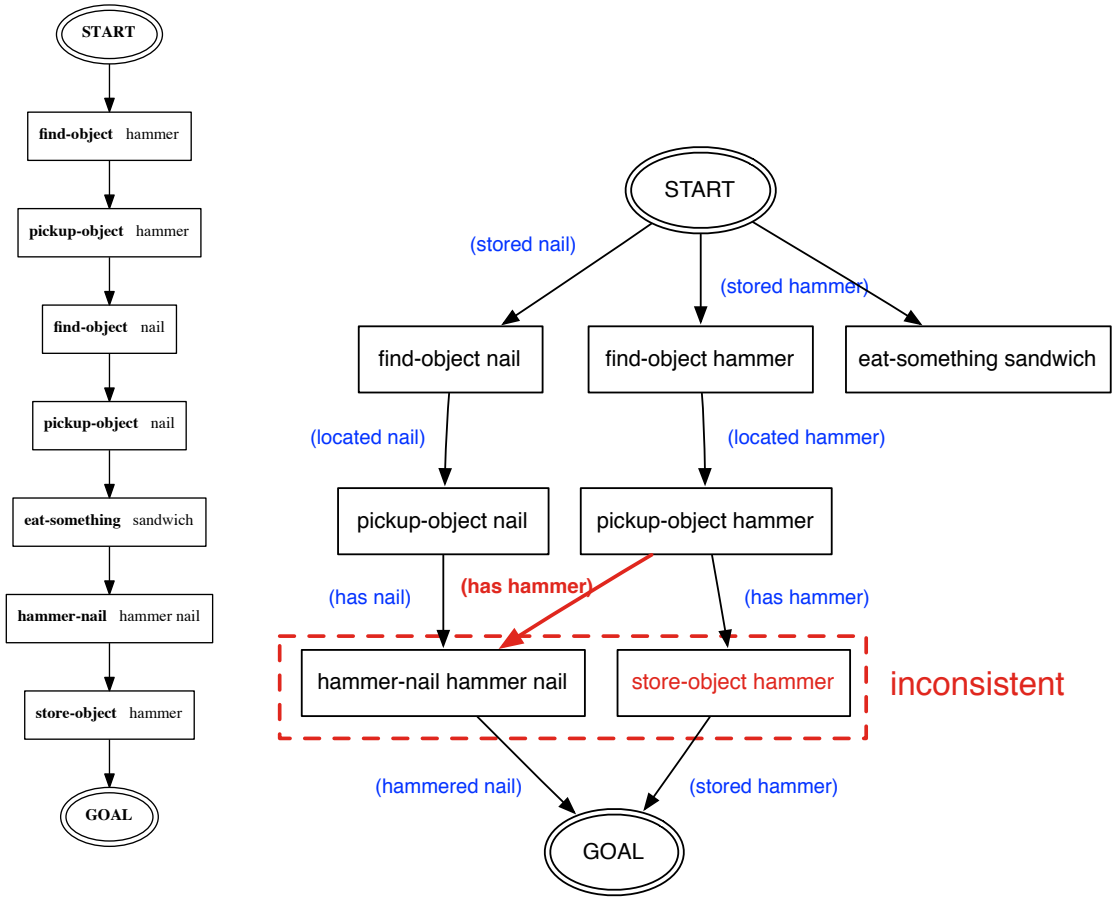
We can observe that only the necessary ordering constrains are drawn in the graph. For example, the partial order plan in 8b shows that in order to do action `(pickup-object hammer)`, we need to find the hammer first. However, although in the demonstrated action sequence in 8a the hammer was found and picked up before the nail, the partial order plan states that the order is not importan and that the nail could be picked up before the hammer and we would be able to reach the same goal.

It is also very interesting to observe that, because this algorithm is considering that we know the goal conditions, we can use the resulting partial order plan to identify the actions in the demonstrated action sequence that are irrelevant to reach the goal. In effect, an action is 'useless' if there is no direct path that from the action to the goal. In the example in 8b we can see that the action `(eat-something sandwich)` is irrelevant.

4.3 Threat resolution

Algorithm 2 presented in section 4.2 properly extracts partial order plans from action sequences in simple cases. However, this algorithm doesn't always provide consistent partial order plans because it doesn't

consider how actions can threat the causal links extracted by the algorithm.



(a) Same action sequence as in 8a with an extra action `(store-object hammer)`

(b) Inconsistent partial order plan extracted by algorithm 2

Figure 9: Example of a situation in which algorithm 2 fails to provide a consistent partial order plan.

In effect, figure 9 shows that algorithm 2 can provide inconsistent partial order plans. This example is the same one presented in figure 8 with one difference: there's a new additional goal condition that indicates that the hammer needs to be stored at the end. In order to satisfy this new requirement, the action sequence was extended with one additional action `(store-object hammer)`, that stores the object after hammering the nail. Running our implementation of algorithm 2 on this action sequence generated the graph shown in figure 9b. We can see that this graph is not imposing any ordering constraints between actions `(hammer-nail hammer nail)` and `(store-object hammer)`. Therefore, we this partial plan is suggesting that we could first do action `(store-object hammer)` and then `(hammer-nail hammer nail)`, but we know that that would be infeasible since we need to have the hammer in order to hammer the nail. In order to fix algorithm 2 to generate the correct partial order plan for this example, we need to **handle threats**.

We now present in this part of the project a revised version of algorithm 2, algorithm 3 that can resolve

Algorithm 3: EXTRACT_ANY_MACPO finds a minimal annotated partial order plan.

Input: A totally ordered plan consisting of a sequence of action steps $\mathcal{T} = S_1, ..., S_n$, the initial conditions and the goals.
Output: A minimal annotated partial order plan given as a directed graph in which edges are causal links.

$START\_step \leftarrow$ *build_action_step*$(None, initial\_conditions)$
$GOAL\_step \leftarrow$ *build_action_step*$(goals, None)$
Insert $START\_step$ at the beginning of $\mathcal{T}$
Insert $GOAL\_step$ at the end of $\mathcal{T}$
for $S_i \leftarrow S_{n+1}$ ***downto*** S_1 **do**
  for P_{ij} *in preconditions of* S_i **do**
    $S_j \leftarrow$ find_last_producer(P_{ij})
    Add_Causal_Link$(\mathcal{P}, S_j, S_i, P_{ij})$
return *Resolve_Threats*$(\mathcal{P})$

Algorithm 4: RESOLVE_THREATS adds the necessary causal links to ensure the partial order plan is consistent.

Input: A partial order plan $\mathcal{P}$ consisting of a directed tree made out of causal links.
Output: A partial order plan $\mathcal{P}$ with resolved threats.

for c *in causal links of* $\mathcal{P}$ **do**
  $S_i \leftarrow$ producer(c)
  $S_j \leftarrow$ consumer(c)
  for $k \leftarrow 1$ ***upto*** $i-1$ **do**
    if *Threatens*$(S_k, S_i \rightarrow S_j)$ **then**
      `// Previous step threatens consumer.`
      demote: Add_Causal_Link$(\mathcal{P}, S_k, S_i)$
  for $k \leftarrow j$ ***upto*** n **do**
    if *Threatens*$(S_k, S_i \rightarrow S_j)$ **then**
      `// Posterior step threatens producer.`
      promote: Add_Causal_Link$(\mathcal{P}, S_j, S_k)$
return $\mathcal{P}$

threats and always extracts consistent partial order plans. This algorithm operates in exactly the same way as algorithm 2 in order to extract the causal links, but then uses algorithm 4 to ensure that threatened causal links are protected.

Intuitively, algorithm 4 resolves threats by iterating through all causal links in the partial order plan in order to find all actions that threaten either the consumer or the producer of that causal link. An action that threatens the causal link and that occurred in the demonstrated action sequence earlier than the producer of the causal link is said to threaten the consumer, because if the action was executed after the producer action, the consumer action would not be feasible any more, as the threatening action would have compromised its preconditions. Conversely, a threatening action that occurred after the consumer in the demonstrated sequence is said to threaten the producer, because if the action was executed before the consumer, the effect of the producer that the consumer needs would be compromised by this action. This threatened actions are then resolved in the same way that were resolved in the demonstration sequence. That is, actions that threaten the consumer are explicitly *demoted* (placed before the producer action) by adding a causal link from the threatening action to the producer action. In the same way actions that threaten the producer are promoted (placed after the consumer action).

Using the improved partial plan extraction algorithm (algorithm 3), the consistent partial plan that corresponds to the action sequence shown in figure 9a is drawn in figure 10

4.4 Extracting partial order plans when the goal conditions are unknown

It is interesting to note that in the algorithms presented up to this section, we are assuming that we know what the goal conditions were for the given action sequence. While this assumption was very convenient in order to design a user interface that would stop asking for new actions as soon as the goal was reached, there's not any reason why the partial order plan extraction algorithm (algorithm 3) wouldn't work if the

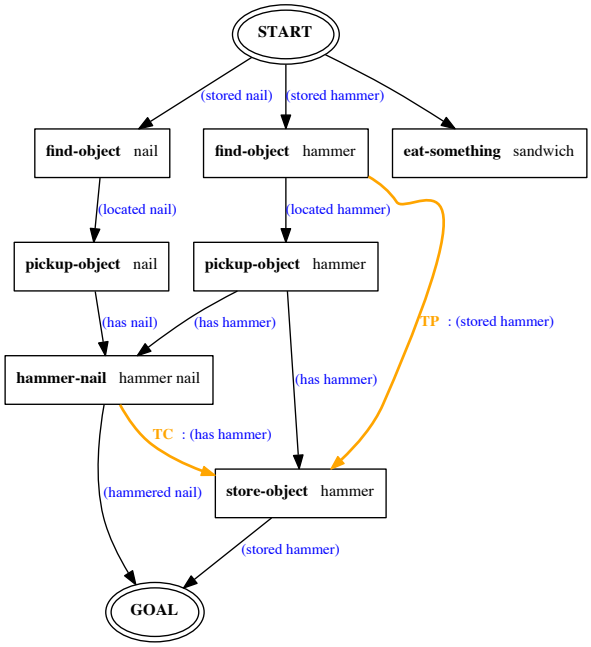


Figure 10: Consistent partial order plan extracted from the action sequence in figure 9a

goal conditions weren't known.

In fact, we can easily modify algorithm 3 to extract partial order plans without knowing the initial conditions as shown in the new algorithm 5. Note that the only differences compared with algorithm 3 are in lines 2 and 4: instead of adding the goal conditions at the end, we add all the end state predicates from the state sequence as if they were the goals.

Note that one mayor limitation of this approach is that we are no longer able to detect 'useless' actions that do not contribute to the goal since we don't precisely know what the goals are. Remember that in the examples that we have been showing until now, we were able to determine that the action `(eat-something sandwich)` was irrelevant since there was not any path from that action to the goal. We are now no longer able to do this, since the goals are unknown and it could be possible that 'eating something' was a desired outcome of the demonstrated action sequence.

4.5 Algorithm Limitations

Although the examples shown previously prove that the algorithm presented here works and is able to provide good results, there are situations in which the algorithm is not able to provide a partial plan that doesn't contain ordering constraints that a human would consider unnecessary.

Consider, for example, the demonstrated action sequence represented in figure 11a. In this example, the PR2 robot finds, picks up and drops in a basket the yellowball and the greenball. However, in order to pick up a ball, its gripper needs to be empty (it can only carry one ball at a time). Although we as humans understand that if the goal is to drop both balls in the basket the order in which the balls are found, picked up and stored is irrelevant, the algorithm presented in this project is unable to reach the same conclusion. As we can see in figure 11b there is a causal link between `(drop-ball yellowball)` and `(pickup-ball`

Algorithm 5: EXTRACT_ANY_MACPO finds a minimal annotated partial order plan.

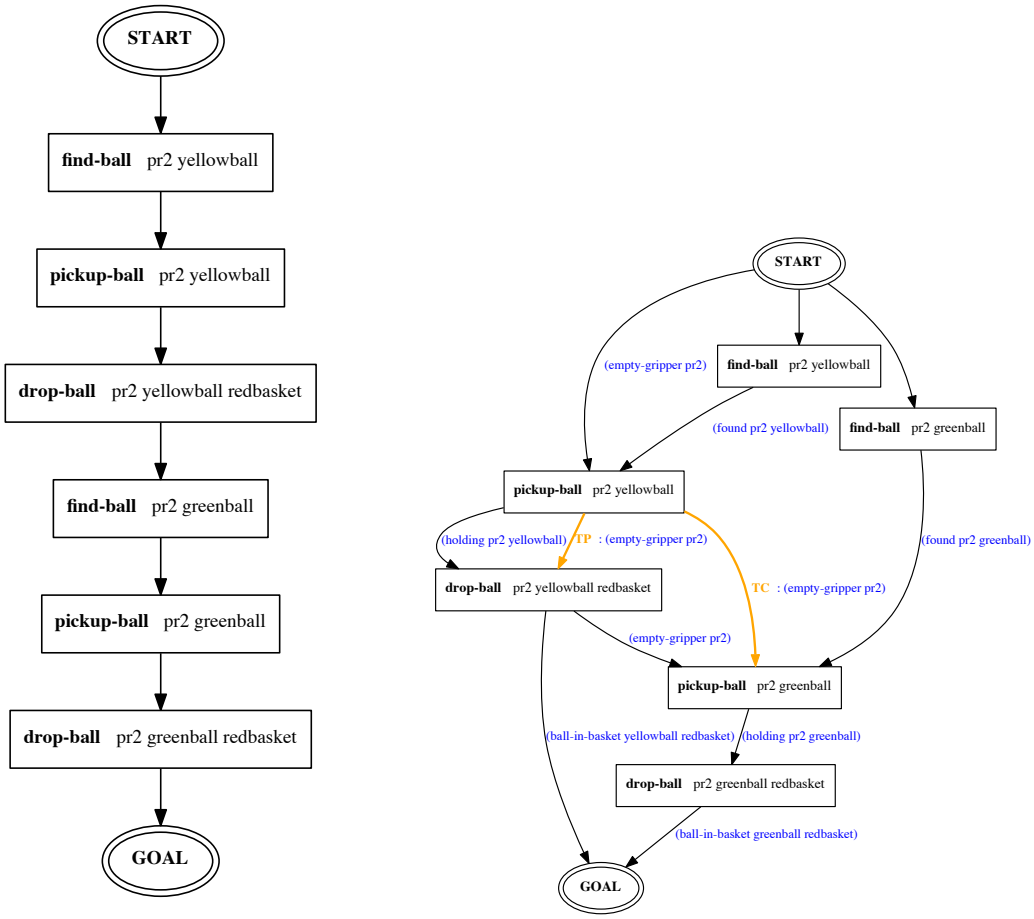
Input: A totally ordered plan consisting of a sequence of action steps $\mathcal{T} = S_1, ..., S_n$, the initial conditions and the state sequence $T = T_1, ..., T_{n+1}$

Output: A minimal annotated partial order plan given as a directed graph in which edges are causal links.

```
START_step ← build_action_step(None, initial_conditions)
END_step ← build_action_step(T_{n+1}, None)
Insert START_step at the beginning of 𝒯
Insert END_step at the end of 𝒯
for S_i ← S_{n+1} downto S_1 do
    for P_ij in preconditions of S_i do
        S_j ← find_last_producer(P_ij)
        Add_Causal_Link(𝒫, S_j, S_i, P_ij)
return Resolve_Threats(𝒫)
```

`greenball)` that imposes an ordering constraint between these two actions. The reason why the algorithm is adding this causal link is that one of the preconditions of `(pickup-ball greenball)` is that the gripper is empty and the algorithm is finding the last producer action of this predicate, which is the action `(drop-ball yellowball)`.

Note that ideally we would like to come up with a partial order plan that represents the fact that the yellow and the green ball could be stored in any order such as the ideal one shown in figure 12. Note, in this case, that the actions are considered in blocks. That is, it doesn't matter which block the robot does first (the yellow or the green ball) as long as it completes the whole block before proceeding with the other block. Siddiqui and Haslum 2012 explore this idea and present an algorithm for block decomposition that seems to work well for this type of situations.



(a) Example of an action sequence that doesn't provide the expected results

(b) Partial order plan with apparently more constrains than needed

Figure 11: Example of a situation in which the results of the algorithm are not as good as expected

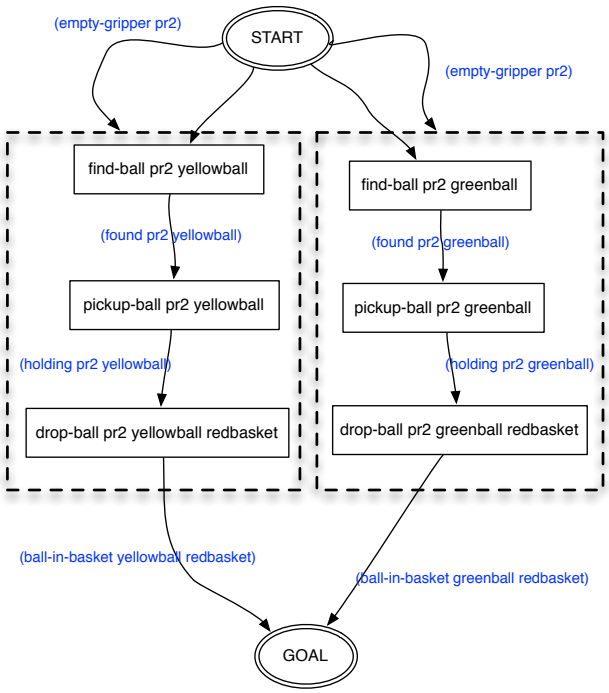


Figure 12: Ideal block partial order plan that better captures the action sequence in figure 11a

5 Conclusion and Lessons Learned

We chose this project on the topic of plan learning because of its large impact on robotics, allowing non-experts to program robots to perform tasks through demonstration. We gave a lecture on this topic, where we discussed a method for learning task specific plans from demonstrations, starting at robot behaviors and object types.

For this project we decided to focus on a subset of the method from the lecture for the project, starting with PDDL actions, demonstrating an action sequence and, finally, extracting a partial order plan.

In doing so, we wrote a parser to parse PDDL actions into classes, wrote a user interface for interacting with the robot and implemented an algorithm to extract partial order plans from total order plans.

We showed how the algorithm in section 4 can provide consistent partial order plans even in the presence of threats, and we showed that the generated partial order plans provide more flexibility during the execution (least commitment plans). We also showed how in the case of knowing the goals we were able to identify 'useless' actions that are not necessary to reach the goal and how, even without knowing the goals, we are still able to find partial order plans. Finally, we exposed an important limitation of the presented algorithm (section 4.5) and pointed at recent research that aims to solve the problem by doing block decomposition[Siddiqui and Haslum 2012].

We would like to remind the reader that, although the implemented algorithm has the advantage of running in polynomial time, it is not optimal. The reader can, however, find an optimal partial order extraction algorithm in [Winner and Veloso 2002].

Lastly we would like to say that we are glad to have chosen this interesting topic and we enjoyed implementing the project. Being relatively unfamiliar with the topic of planning, it was a great experience for us to understand the differences between total and partial order plans and how we can learn plans for robotic applications. We not only enjoyed doing this project, but we also found out that this topic is relevant for our research and will help us going forward.

References

[1] Brenna D Argall et al. “A survey of robot learning from demonstration”. In: *Robotics and Autonomous Systems* 57.5 (May 2009), pp. 469–483.

[2] Enrique Fernandez and James Paterson. “Plan Learning Lecture”. In: *Cognitive Robotics Advanced Lecture Series*. Apr. 2013.

[3] Fazlul Hasan Siddiqui and Patrik Haslum. “Block-Structured Plan Deordering”. In: *AI 2012: Advances in Artificial Intelligence*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 803–814.

[4] H Veeraraghavan, R Vaculin, and M Veloso. “Learning Task Specific Web Services Compositions with Loops and Conditional Branches from Example Executions”. In: *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*. 2010, pp. 581–588.

[5] Harini Veeraraghavan and Manuela Veloso. *Teaching sequential tasks with repetition through demonstration*. International Foundation for Autonomous Agents and Multiagent Systems, May 2008.

[6] M Veloso and Harini Veeraraghavan. “Learning task specific plans through sound and visually interpretable demonstrations”. In: *Chen, Liming* (2008), pp. 2599–2604.

[7] E Winner and M Veloso. “Analyzing plans with conditional effects”. In: *Proceedings of the Sixth International Conference on ...* 2002.

[8] Elly Winner and Manuela Veloso. “LoopDISTILL: Learning looping domain-specific planners from example plans”. In: (2007).

Appendices

A Using the provided source code

A.1 Overview

The code developed for this project consists on a python package called **pyplangenius** that is distributed together with this report. This package provides two main capabilities:

1. Generating action sequences from user inputs and PDDL files
2. Generating partial order plans (with or without threat resolution) using the previously generated action sequence

Inside the package we have included a program called pyplangenius.py that chains items 1 and 2 above. That is, it will read and parse the specified domain and problem PDDL files, generate an action sequence from user input, extract the partial order plan using the algorithms in section 4 and generate and save to disk PDF graphs of the total order plan and the partial order plan using graphviz.

A.2 Using the code

The python program that runs the demos is called pyplangenius.py and is placed in the src folder.

The easiest way to test the code is by running:

```
./projectdemo
```

inside the src folder. This program is just a bash script that calls

```
python pyplangenius.py hammerworld-domain.pddl hammerworld-problem-nothreat.pddl
```

This will start the user interface to build an action sequence. Note that the user interface supports TAB completion that makes it easier to enter action sequences. After entering the required actions to reach the goal (`hammered nail`), the program will automatically generate the partial order plan for the action sequence. The program will also create PDF graphs of the action sequence and partial order plan and store them in the output_graphs folder (`total_order`).

pyplangenius.py needs to be called with the following options:

```
pyplangenius.py [--args] [pddl_domain] [pddl_problem]
```

where args denotes the following optional arguments:

- `--no-poplan` Only generate the action sequence, but not the partial order plan.
- `--no-threat-resolution` Generate partial plans without considering threats, such as in algorithm 2

Apart from using `./projectdemo`, you may want to test the code by running:

```
python pyplangenius.py hammerworld-domain.pddl hammerworld-problem-threat.pddl
--no-threat-resolution
```

to see a the resulting partial order plan in which threats are not considered, and

```
python pyplangenius.py hammerworld-domain.pddl hammerworld-problem-threat.pddl
```

to see the partial plan in which threats are resolved using algorithm 3.

Remember that in any case the generated graphs are stored in the `output_graphs` folder.

A.3 Requirements

The main requirement for this python package is to have graphviz installed in your system. The `dot` program should be in the `PATH`. There are other python third party dependencies, but are included in the src/third_party folder and should be imported automatically by the package.

A.4 Troubleshooting

This code has been tested on OSX 10.8.3 and on Ubuntu 12.04 and should work on these systems out of the box, but your mileage may vary, as the code depends on some external programs such as Graphviz that are outside of the authors control. Please do not hesitate to contact the authors in case the code does not run as expected.