PDDL Planner for HRC Task

Abhinav Dahiya

***Abstract*—PDDL Planner is an A^* search based solver made for planning problems written in PDDL-like format.**

***Index Terms*—PDDL, Planner, STRIPS, A^* search**

I. Introduction

The planner implements an A^* search algorithm, which takes in the domain and problem instances defined in a PDDL-like format (*in a manner that saves trouble for parsing the actual PDDL files*). The planner is extended to include various features of PDDL specification, such as conjunction, disjunction, conditional propositions, negation preconditions, numerical preconditions, ADL definitions (*forall, exists*), and any possible combination of all of these.

The implementation is being optimized as we go but still the current version is quite slow to handle larger problems like the one we are tackling right now.

Following sections cover the details of various components of the planning problem and working of the planner.

II. Problem definition

We are solving for a *kitting* problem which starts with a number of *containers* stacked together each containing a certain number of various *components*. The goal is to arrange the components in a specified way on a *kit tray*. The whole task includes bringing the containers to an *unloading location*, *unloading* the component inside that container and placing it on the tray. There are certain rules to the game regarding how and when can any *object* be moved or acted upon. There is also a limit on total number of components one can unload at a given time and number of *platforms* on which the containers can be stacked.

Following are some terminology that is necessary to define various concepts and attributes of a PDDL problem definition:

- State, Action, Predicates, Literals, Atoms, Fluents, Functions

As in standard PDDL, the problem is defined using a domain file and a problem instance file.

A. Domain

The domain file defines the objects present in the domain and the actions allowed. In the kitting domain, containers, components, grid etc. constitute the *objects*.

An action in PDDL format is defined by

- Name
- Parameters; the variables representing objects from the problem instance on which the action is applied. **e.g.**
- Preconditions; a set of conditions that need to be *True* in a given *state* for that action to be applicable in that state. These can be positive preconditions (that should be true), negative preconditions (should be false) or numerical ones (specifying numerical constraints over state predicates(*check word usage*??))**e.g.**
- Effects; a set of changes that the action will bring upon the state it is applied to. Again, there can be *add effects* specifying the new predicates to be added to the state, *delete effects* specifying the predicates to be removed from the state and numerical effects which modify the values of certain state predicates **e.g.**.

There are 10 different actions in this domain *movebox, move_to_station, move_from_station, place_component, unlock_component, return_component, grid_to_grid, grid_to_display, error_action, redundant_action*.

B. Problem Instances

Defined by: Number of containers, number of components in each container, number of extra platforms for stacking, number of maximum allowed unlocked / unloaded components, order of stack of containers and target arrangement of components in the tray.

III. Problem parsing and interpretation

A. Grounding

Based on problem instance, the actions defined in PDDL domain file are *grounded*, i.e., the variables used are replaced with objects of that problem.

B. Applicability

During the search, expanding a node (state) requires knowledge of all the grounded actions which can be taken in that particular state. This is done using a function which checks for

- presence of positive action preconditions in the state predicates
- absence of negative preconditions
- agreement of numerical preconditions

- agreement of all these conditions subjected to any propositional logic that may be present in action preconditions

Once we know of the applicable actions given a state we expand the node corresponding to that state in our search tree by applying these actions on the node.

C. Applying actions

It simply includes doing the following tasks:

- adding positive action effects to the state predicates
- deleting negative effects
- updating numerical values of state functions
- implementing these effects based on any propositional logic that may be present in action effect definition

IV. Working of Planner

As mentioned earlier, the planner implements a *Fringe Search* algorithm [**?**], which is ....

1) Given a node, find all children nodes by applying all actions possible on that node,
2) Calculate **f-value** of those children nodes:

$$f(child) = g(parent, child) + heuristics(child) \quad (1)$$

where $g(parent, child)$ is the cost of getting from parent to child node and $heuristics(child)$ is the estimated cost to go from child node to the goal.

A. Heuristic

A common approach to calculate heuristics for task planning problems is to compare the current state with the goal state (and count the number of predicates in goal state that are true in the current state etc.) to get an estimate over the *cost to go* from that state to the goal. Another approach is to plan for goal without considering any delete effects of all the actions and then the plan length obtained is used as heuristic in the actual planning.

However, these approaches are not quite useful for the kitting problem because they do not provide much information about the *quality* of a state and which action is going to be better than other (WHY?, because of large action and state space, existence of multiple optimal plans etc.). Therefore, we implemented a domain-specific heuristic which decompose the problem in various sub-tasks (??, moving containers, unloading and placing components) and take into account the various features of the problem.

Explanation of the heuristic

B. Search

Select node with lowest f-value, expand to get all children, repeat until goal is reached.

V. Extension of planner

Other functions for convenience and speed:

A. Multiple shortest paths

For storing information for future planning or to assess complexity of a problem instance.

B. Multi-player game (turn-based, not concurrent)

Taking turns to select an action to play. Same shared goal and no private information.

C. Analyzing a plan

Calculating cost to go... Number of mistakes / number of redundant actions

D. Storing optimal actions for a given state

...for a particular problem instance

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.