Autonomous Systems LAB Session 3: Planning

Arnau Colom, Emma Fraxanet

I. EXERCISE C.1. MONKEY AND BANANAS

A. Description of the model

In our description of the problem there is a monkey that has to eat all the bananas in the room, but needs a chair to get to them. Once the monkey has the chair and is in one of the banana's location, it can proceed to leave the chair on the floor and grab and eat the banana. We separate four actions then: moving, picking up a chair, leaving a chair on the floor and, finally, grabbing and eating the banana in one single action.

In this case we also allow having different chairs throughout the room, since it makes the problem more interesting.

States: we have N possible locations for the monkey, the N_b banana(s) and the N_c chair(s). At the same time, this bananas can be eaten and the chairs can be carried.

The goal states will be those that include both bananas set as eaten.

We attach the *pddl* files that encode the domain and problem inside the folder $lab3/pddl_files$.

B. Problem defined as a STRIPS problem

Problem: $P = \langle F, I, O, G \rangle$
Encoded variables (types): location, banana, chair
For our specific instance:
$Banana = B = \{b1, b2\}$,
$Chair = C = \{c1, c2\}$,
$Location = L = \{loc1, loc2, loc3, loc4, loc5, loc6, loc7, loc8, loc9\}$
Atoms (F):
$F = \{at - monkeyloc(?l), at - chloc(?c, ?l), atloc(?b, ?l), ChairCarried(?c),$
$BananaEaten(?b) | ?b \in B, ?c \in C, ?b \in L\}$
Initial situation (I):
$I = \{at - monkeyloc(loc1), at - chloc(c1, loc9), at - chloc(c2, loc5), atloc(b1, loc6), atloc(b2, loc8)\}$
Goal situation (G):
$G = \{bananaEatenb1, bananaEatenb2\}$
Operators(O) or Action schemas:

1. $move(?l1, ?l2 \in L)$: $\boldsymbol{P}$: $\{at - monkeyloc(?l1)\}, \boldsymbol{A}$: $\{at - monkeyloc(?l2)\}, \boldsymbol{D}$: $\{at - monkeyloc(?l1)\}$
2. $pickchair(?b \in B, ?l \in L, ?c \in C)$: $\boldsymbol{P}$: $\{not(at(b, l)), at - ch(c, l), at - monkey(l), not(ChairCarried(c))\}, \boldsymbol{A} : \{ChairCarried(c)\}, \boldsymbol{D} : \{not(at - ch(c, l))\}\}$
3. $leavechair(?b \in B, ?l \in L, ?c \in C)$: $\boldsymbol{P}$: $\{at(b, l), at - monkey(l), ChairCarried(c)\}, \boldsymbol{A}$: $\{at - ch(c, l)\}, \boldsymbol{D} : \{not(ChairCarried(c))\}\}$
4. $grabeat(?b \in B, ?l \in L, ?c \in C)$: $\boldsymbol{P}$: $\{at(b, l), at - monkey(l), not(ChairCarried(c)), at - ch(c, l), not(bananaEaten(b))\}, \boldsymbol{A} : \{bananaEaten(b)\}, \boldsymbol{D} : \{at(b, l)\}\}$

II. EXERCISE C.2. TELEPORTING SOKOBAN

A. Description of the model

The Sokoban problem describes the following puzzle: A player is initially confined inside a board, and he can only move vertically and horizontally. If the player moves towards a box, and there is no other box or wall in the square beyond the box, the player can move to the box position and push the box to the position beyond that. The objective of the problem is to locate the boxes in the storage locations. In addition, in this case the player is allowed to teleport to any empty square only once.

To describe the problem we only need the following encoded variable:
$Coordinate_X = X = \{x_1, x_2, x_3, ...x_w\}$, *where* w *is the width of the board.*
$Coordinate_Y = Y = \{y_1, y_2, y_3, ...y_h\}$, *where* h *is the height of the board.*

The predicates that we need to keep track of the problem and describe its states are the following:

- ***at*** $?x - x$ $?y - y$*: Describes the position of the user.*
- ***iswall*** $?x - x$ $?y - y$*: Define the location of the walls.*
- ***hasbox*** $?x - x$ $?y - y$*: Track the location of the boxes.*
- ***hastp****: Is used to track the use of the teleport action.*
- ***incx*** $?x1$ $?x2 - x$ */* ***decx*** $?x1$ $?x2 - x$*: With this predicates we can know if the coordinate* x *is adjacent and increases/decreases.*
- ***incy*** $?y1$ $?y2 - y$ */* ***decy*** $?y1$ $?y2 - y$*:: With this predicates we can know if the coordinate* y *is adjacent and increases/decreases.*
- ***samex*** $?x1$ $?x2 - x$*: We can know if two x-coordinates are the same.*
- ***samey*** $?y1$ $?y2 - y$*: We can know if two y-coordinates are the same.*

Operators(O) or Action schemas*: These operators define the movement/teleportation of the agent to a given location, and the pushing of a box towards a location. We exemplify here the case of moving towards the up direction only. The other directions follow the same logic.*

1. ***move_up****:* $(?xfrom \quad ?xto \quad \in X \quad ?yfrom \quad ?yto \quad \in Y)$:
   P*:* $\{at(?xfrom \quad ?yfrom), \quad not(iswall(?xto \quad ?yto)), \quad incy(?yfrom \quad ?yto),$
   $samex(?xfrom \quad ?xto), \quad not(hasbox(?xto \quad ?yto))\}$
   $\boldsymbol{A} : \{at(?xto \quad ?yto)\}$
   $\boldsymbol{D} : \{at(?xfrom \quad ?yfrom)\}$
2. ***moveboxup****:* $(?xfrom \quad ?xto \quad ?xboxto \quad \in X \quad ?yfrom \quad ?yto \quad ?yboxto \quad \in Y)$:
   P*:* $\{at(?xfrom \quad ?yfrom), \quad not(iswall(?xto \quad ?yto)), \quad not(iswall(?xbotto \quad ?yboxto))$
   $incy(?yfrom \quad ?yto), incy(?yto \quad ?yboxto), \quad samex(?xfrom \quad ?xto),$
   $samex(?xto \quad ?xboxto), \quad (hasbox(?xto \quad ?yto)), \quad not(hasbox(?xboxto \quad ?yboxto))\}$
   $\boldsymbol{A} : \{at(?xto \quad ?yto), \quad hasbox(?xboxto \quad ?yboxto)\}$
   $\boldsymbol{D} : \{at(?xfrom \quad ?yfrom), \quad hasbox(?xto \quad ?yto)\}$

3. ***teleport***: $(?xfrom \quad ?xto \quad \in X \quad ?yfrom \quad ?yto \quad \in Y)$:
   P: $\{at(?xfrom \quad ?yfrom), \quad not(iswall(?xto \quad ?yto)), \quad not(hasbox(?xto \quad ?yto)) \quad not(hastp)$
   A : $\{at(?xto \quad ?yto)\}$
   D : $\{at(?xfrom \quad ?yfrom), \quad hastp\}$

The definition of the Goal will be defined using the ***hasBox*** *predicate. The goal situation is having all boxes in a goal location and the definition will have as many elements as boxes in our board.*

Goal situation (G): $G = \{hasbox \quad ?x \quad ?y, ...\}$

Initial situation (I): *The initial situation will be defined by the location of the walls, the boxes and the agent. It also needs a definition of which way does the index of the position variables increase. For example,* $x1$ *is smaller than* $x2$ *(incx* $?y1$ $?y2$ *, decx* $?y2$ $?y1$ *).*

B. Sokoban solver

We attach the domain.pddl and the code (sokoban.py) that generates an instance (probl.pddl) from a .sok file, calls the planner, finds the plan and translates the order of actions to obtain a successful outcome. All files for this part are in the folder lab3.

C. Sasquatch problems results

In the following tables we present the solved problems from the folder benchmarks/sasquatch in less than 1 minute for different options of algorithms and heuristics available for Fast Downward. It is clear to see that these problems are hard for our pddl approach, since with most techniques we barely solve a quarter of them, and could be made easier by changing some rules to the game. For example, if the teleport action could be made more than once, the paths would be easier to find. The sasquatch problems also tend to have narrow spaces, and this makes the moving of the boxes considerably more complex.

To choose the different algorithms and heuristics to solve our Sokoban problem we first took a quick look to the planners that provides Fast Downward using the command line `python fastdownward.py --showa-aliases`.

We can see from the tables that some techniques deliver better results for less solved levels while others can solve more levels with lower efficiency. In particular we find that the most efficient technique, that also obtains solutions for three of the levels, is using A search with an iPDB heuristic. This is an optimal technique. At the same time, the BJOLP technique is also optimal because it results in the shortest path for level two. The FF heuristic and the context-enhanced additive heuristic with Lazy greedy search are satisfying, even though they are not admissible heuristics. Finally the LAMA algorithm uses a combination of an inadmissible heuristic (FF) with landmark based estimates and is satisfying.*

LAMA FIRST	
Level Solved	**Steps**
1	160
2	48

Table I: `seq-sat-lama-2011`

BJOLP	
Level Solved	**Steps**
2	34

Table II: `seq-opt-bjolp`

As we can appreciate from the tables, the LAMA heuristic is capable to generate more solutions but, in exchange, the number of steps is greater.

FF Heuristic and Lazy Greedy	
Level Solved	**Steps**
1	190
2	72
14	76

Table III: FF Heuristic and Lazy Greedy

Context-Enhanced Additive and Lazy Greedy	
Level Solved	**Steps**
2	56

Table IV: Context-Enhanced Additive and Lazy Greedy

A* with iPDB heuristic	
Level Solved	**Steps**
1	103
2	34
14	45

Table V: A* with iPDB heuristic

FF Heuristic and Context-Enhanced	
Level Solved	**Steps**
2	62
14	73

Table VI: FF Heuristic and Context-Enhanced

D. Search algorithms and heuristics

***Landmark heuristics**:*
Landmarks are propositional formulas that must be true in every solution of a planning task, like implicit problem subgoals. The use landmarks helps direct search towards those states where many landmarks have already been achieved. This eventually helps estimating heuristics better.

LAMA FIRST is a planner system with multi-heuristic search (it uses landmark heuristic and a variant of FF-heuristic). It also uses a weighted A search with iteratively decreasing weights, so that the planner continues to search for plans of better quality until the search is terminated.*
[Richter, S., Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. Journal of Artificial Intelligence Research, 39, 127-177.]

BJOLP also uses landmark heuristics.
[BJOLP: The Big Joint Optimal Landmarks Planner [Domshlak, C. Helmert, M. Karpas, E. Keyder, E. Richter, S. Roger, G. Seipp, J. Westphal, M.]]

***Fast-Forward Heuristics**:*
FF relies on forward search in the state space, guided by a non-admissible heuristic that estimates goal distances by ignoring delete lists. Used with a greedy search algorithm. Since it uses a non-admissible heuristic it may overestimates the cost of reaching the goal and it may result in an not optimal solution.

***Context-Enhanced Additive Heuristic**:*
Inadmissible heuristic where fact costs are evaluated relative to context states that arise from achieving first a pivot condition of each operator.
[Helmert, M., Geffner, H. (2008, September). Unifying the Causal Graph and Additive Heuristics. In ICAPS (pp. 140-147).]

***A* with iPDB heuristic**:*

iPDB is a pattern database heuristic. Admissible heuristic. Due to the domain specific properties of the goal states a normal PDB is innefective. But the use of iPDB is the currently best admissible heuristic function for Sokoban, it effectively detects deadblocks. This could be the reason why using this heuristic generates better solutions. [André G.Pereira, MarcusRit, tLuciana S.Buriol (2015, October)Optimal Sokoban solving using pattern databases with specific domain knowledge. In Artificial Intelligence Volume 227 (pp. 52-70)]