Planning and Optimization

M. Helmert, G. Röger
F. Pommerening

University of Basel
Fall Semester 2017

Exercise Sheet A

Due: October 15, 2017

The files required for this exercise are in the directory `exercise-a` *of the course repository (*`https://bitbucket.org/aibasel/planopt-hs17`*). All paths are relative to this directory. Update your clone of the repository with* `hg pull -u` *to see the files.*

Exercise A.1 (3+2+2+2+2+4+2+3 marks)

In this exercise we consider the solitaire game Beleaguered Castle (`http://justsolitaire.com/Beleaguered_Castle_Solitaire/`). It consists of a deck of cards stacked face-up in several tableau piles. For each suit in the deck there is a discard pile consisting only of the ace initially. There are three types of legal moves:

- The top card of a tableau pile can be moved on top of another tableau pile if the top card of the target pile has a value that is one higher. The suit of both cards does not matter for this move. For example, 2♣ can be moved on 3♥, 10♣ on J♠, or Q♦ on K♦.
- The top card of a tableau pile can be moved to an empty tableau pile. This is allowed for all cards (not just for kings as in other solitaire games).
- The top card of a tableau pile can be moved to the discard pile for the matching suit if the top card on the discard pile has a value one lower. For example, if 7♥ was discarded last, then 8♥ can be discarded next. Discarded cards can never be moved again.

We consider a parametrized version of the game with m tableau piles $\mathit{Tableaus} = \{t_1, \dots, t_m\}$ and any set of cards Cards. For a given card $c \in \mathit{Cards}$ we use $\mathit{suit}(c)$ and $\mathit{value}(c)$ to refer to its suit and numerical value. The set of discard piles contains one discard pile for each suit: $\mathit{Discards} = \{\mathit{discard}_s \mid s = \mathit{suit}(c) \text{ for a } c \in \mathit{Cards}\}$. The set of all piles is $\mathit{Piles} = \mathit{Tableaus} \cup \mathit{Discards}$.

(a) Model Beleaguered Castle as a family of propositional planning tasks. Use the following state variables:

- $c\text{-}\mathit{on}\text{-}x$ for all $c \in \mathit{Cards}$ and $x \in \mathit{Cards} \cup \mathit{Piles}$
  For cards c_1, c_2 the variable $c_1\text{-}\mathit{on}\text{-}c_2$ should be true iff c_1 is directly on top of c_2; For a pile p the variable $c_1\text{-}\mathit{on}\text{-}p$ should be true iff c_1 is directly on top of the pile, i.e., iff c_1 is the bottom-most card in the pile p.
- $x\text{-}\mathit{clear}$ for all $x \in \mathit{Cards} \cup \mathit{Tableaus}$
  An object should be clear iff there is no card on top of it.
- $c\text{-}\mathit{discarded}$ for all $c \in \mathit{Cards}$
  A card is discarded iff it is on the discard pile.

(b) Transform your model into an equivalent family of STRIPS tasks.

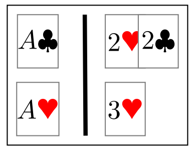
(c) Prove that in your model each card is always on top of exactly one other card or pile. That is, prove that the following formula is an invariant:

$$\varphi_c = \bigvee_{x \in X} c\text{-}\mathit{on}\text{-}x \wedge \bigwedge_{x_1 \in X} \bigwedge_{\substack{x_2 \in X \\ x_1 \neq x_2}} \neg(c\text{-}\mathit{on}\text{-}x_1 \wedge c\text{-}\mathit{on}\text{-}x_2) \qquad \text{for } X = \mathit{Piles} \cup \mathit{Cards} \setminus \{c\}.$$

(d) Define useful mutex groups and transform your model in an equivalent family of FDR planning tasks.

(e) Bring your FDR model into TNF.

(f) Consider the following example instance in each model of exercises (a), (b), (d), and (e):



Compare the state spaces of the different formulations. How many states do they have? Approximately how many of those states are reachable? How many states are visited by an optimal plan in each of the models? Are the state spaces the same? Are they equivalent? What remains the same, what changes?

(g) Discuss advantages and disadvantages of each formulation. Look at different theoretical and practical angles in your discussion.

(h) Model the domain in PDDL. In the directory `castle` you can find a script to generate instances with different parameters. Generate a few instances with n cards per suit and $\lceil \frac{n}{2} \rceil + 1$ tableau piles for different values of n and different random seeds. Try to find a good configuration in Fast Downward to solve generated instances. What is the most complex game you are still able to solve within a time limit of 60 seconds and a memory limit of 2 GB? Which configuration worked best?

The limits can be set with `ulimit -t 60` *and* `ulimit -v 2000000`*. Refer to instances by their generator parameters (e.g.* `task-15-8-1-all`*) instead of including PDDL files.*

Exercise A.2 (3+2+3+2 marks)

Consider the following planning task:

- You are trapped in the cellar of a building with a switch board full of light switches. In the rooms above you there is a vampire. Luckily, there also is a vampire slayer in those rooms. To keep things simple, we consider only room layouts that are circular corridors where each room has a clockwise and an anti-clockwise neighbor.

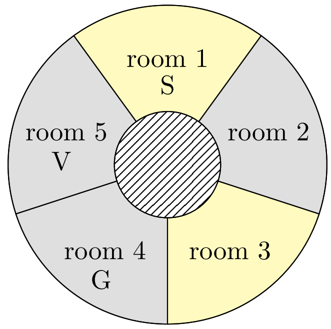
- The vampire avoids the light (yes, even if it is only artificial light). Whenever the light in the vampire's room is switched on, it moves to a neighboring room. If one of the rooms is dark, it will move there, preferring the anti-clockwise one if both are dark. If both neighboring rooms are bright, it will move clockwise.

- The slayer tries to stay in the light. If the light in her room is switched off, she moves to a neighboring room. She moves clockwise if that room is bright and anti-clockwise otherwise.

- If the two of them meet in a room they will fight. The vampire wins the fight in a dark room unless there is garlic in that room. In bright rooms or in rooms with garlic, the slayer wins.

- All you can do is use the switch board to toggle lights and watch the fight, when it happens.

Example instance with five rooms:



(a) There is a partial model of this domain in the directory `vampire`. Complete it by adding the effects of `toggle-light` and `watch-fight`. Do not add new actions or predicates.

The directory also contains instances which you can use for debugging. Their optimal plan costs are 6, 4, 7, 5, 4, 12, 11, 10, 13, and 8. A bug in VAL prevents it from parsing this domain, so use INVAL for debugging instead. You can find it in the directory `INVAL`.

(b) PDDL uses first-order predicate logic to model planning tasks. However, the models discussed in the lecture are all based on propositional logic. Most planners convert PDDL into one of the propositional models in a step called *grounding*. The directory `preprocess` contains a Python tool to do this step. The call

```
./preprocess/ground.py vampire/domain.pddl vampire/p01.pddl
```

will create a new domain file `vampire/domain_grounded_for_p01.pddl` and a new task file `vampire/p01_grounded.pddl`. Repeat this for all task files and describe the effect of the grounding procedure.

(c) In addition to `ground.py` there is an incomplete Python program called `transform.py` in the directory `preprocess` which should transform grounded domains into conflict-free effect normal form. Complete the missing parts and use it to transform your grounded domains from exercise (b) into conflict-free effect normal form. The call

```
./preprocess/transform.py vampire/domain_grounded_for_p01.pddl
```

will create the file `vampire/domain_grounded_for_p01_normalized.pddl`.

(d) Use Fast Downward to generate plans for all tasks using the domains you created in exercises (a)–(c). Then use INVAL to validate each plan against each domain formulation. In which combinations are the plans valid? Discuss the reason for that.

The exercise sheets can be submitted in groups of two students. Please provide both student names on the submission.