Planning and Optimization

M. Helmert, G. Röger — University of Basel
P. Ferber, T. Keller, S. Sievers — Fall Semester 2020

Exercise Sheet A

Due: October 04, 2020

Important: for submission, consult the rules at the end of the exercise. Non-adherence to the rules will lead to your submission not being corrected.

The files required for this exercise are in the directory `exercise-a` *of the course repository (*`https://github.com/aibasel-teaching/planopt-hs20`*). All paths are relative to this directory. Update your clone of the repository with* `git pull` *to see the files. In the virtual machine,* `/vagrant/plan-opt-hs20` *is the repository.*

Exercise A.1 (4 marks)(Lecture A4)

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, $2\clubsuit$ can be moved on $3\heartsuit$, $10\clubsuit$ on $J\spadesuit$, or $Q\diamondsuit$ on $K\diamondsuit$.
- 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 $7\heartsuit$ was discarded last, then $8\heartsuit$ 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 $\mathit{Piles} = \mathit{Tableaus} \cup \mathit{Discards}$.
Model Beleaguered Castle as a family of propositional planning tasks. Use the following state variables:

- c-*on*-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-*on*-c_2 should be true iff c_1 is directly on top of c_2; For a pile p the variable c_1-*on*-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-*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-*discarded* for all $c \in \mathit{Cards}$
  A card is discarded iff it is on the discard pile.

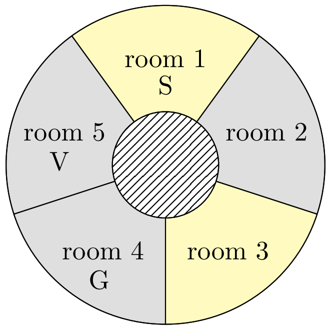
Exercise A.2 (3+2+3+2 marks)(Lecture A5, Exercise Session 1)

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 (V). Luckily, there also is a vampire slayer (S) 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: whenever the light in the vampire's room is switched on, it moves to a neighboring room. If one of the neighboring 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 (G) 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. Your objective is to toggle the lights so that the slayer can win the fight.

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.

You can use INVAL for debugging your PDDL code: ***`inval <domain.pddl> <problem.pddl> <plan>`*** *where problem and plan file are optional.*

(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

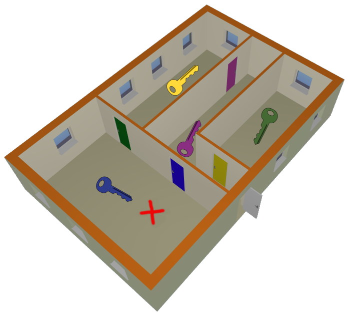
```
./preprocess/transform.py vampire/domain_grounded_for_p01.pddl
```

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.

Exercise A.3 (4+2+2 marks)(Lecture A6 for (a) and (b), Lecture A7 for (c))

Consider the following planning domain: an agent is moving on a map and is trying to reach a specific target. However, there are locked doors between some locations that can only be unlocked by first picking up their corresponding key from some other location. The agent can hold only one key at a time.
Example instance with four rooms:



Let $G = \langle N, E \rangle$ represent the map, let $s, t \in N$ be the start and target location of the agent, let $L \subseteq E$ be the set of locked doors, let *Keys* be the set of keys and let the function *unlocks* : *Keys* $\to L$ denote which door is unlocked by a key and the function *initial-location* : *Keys* $\to N$ denote where each key is located in the beginning.
The domain can then be modelled as a family of FDR tasks $\Pi = \langle V, I, O, \gamma \rangle$ where:

- V consists of the following variables:
  - Variable `at` with domain N denotes the position of the agent.
  - Variable $\langle a, b \rangle$`-locked` for each $\langle a, b \rangle \in E$ with domain $\{\top, \bot\}$ denotes whether or not an edge is locked.
  - Variable `holding` with domain $K \cup \{\text{none}\}$ denotes which key the agent currently holds.
- I sets above variables as follows:
  - $I($`at`$) = s$,
  - $I(\langle a, b \rangle$`-locked`$) = \top$ for all $\langle a, b \rangle \in L$ ($\bot$ otherwise),
  - $I($`holding`$) = \text{none}$.
- O contains the following actions:
  - `move-a-b` and `move-b-a` for all $\langle a, b \rangle \in E$, where `move-`$x$`-`$y$ has precondition `at` $= x \land \langle x, y \rangle$`-locked` $= \bot$ and effect `at` $:= y$.

  - `unlock-`$\langle a, b\rangle$ for all $\langle x, y\rangle \in L$, with precondition $(\texttt{at} = x \lor \texttt{at} = y) \land \texttt{holding} = k \land \langle a, b\rangle\texttt{-locked} = \top$ where *unlocks*$(k) = \langle a, b\rangle$, and effects $\langle a, b\rangle\texttt{-locked} := \bot$ and $\texttt{holding} = \text{none}$.
  - `pick-k` for all $k \in \mathit{Keys}$ with precondition $\texttt{at} = x \land \texttt{holding} = \text{none}$ where *initial-location*$(k) = x$, and effect $\texttt{holding} := k$.

- $\gamma = (\texttt{at} = t)$

(a) Transform the model into an equivalent family of STRIPS planning tasks.

(b) Consider the example instance above in the given FDR model and your STRIPS model. Compare the state spaces of the two formulations. How many states do they have? How many of those states are reachable? Are the state spaces the same? Are they equivalent? What remains the same, what changes?

(c) List the mutex groups over the state variables of your STRIPS task in part (a) that would induce the variables in the given FDR task.

Exercise A.4 (8 marks)(Lecture A8)
This exercise is a literature research question. We might repeat this kind of exercise from time to time. The goal of such exercises is to find information in research papers. We will provide you with starting points for your search. We don't expect you to fully read all papers. Instead, try to extract the relevant information to answer the question. This time, you will find all required information in the given paper, but in the future, you might need to follow references.

Provide an overview of how different syntactic restrictions alter the complexity of PlanEx and BCPlanEx for STRIPS planning tasks. Elaborate on either the first or the second case where the complexity is polynomial. Include a summary of the corresponding formal proofs of Theorem 3.7 or Theorem 3.8, respectively.
Your answer should be in the form of a written text, not only bullet points and tables. Between half a page and one page should be sufficient to answer this question in detail.
You'll find the necessary information in the following paper:

> Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2), 165–204.

Submission rules:

- Exercise sheets must be submitted in groups of two or three students. Please submit one single copy of the exercises per group (only one member of the group does the submission).
- Create a single PDF file (ending .pdf) for all non-programming exercises. If you want to submit handwritten parts, include their scans in the single PDF in a reasonable resolution, so that they are readable but the PDF size is not astronomically large. Put the names of all group members on top of the first page. Use page numbers or put your names on each page. Make sure your PDF has size A4 (fits the page size if printed on A4).
- For programming exercises, only create and submit those code textfiles required by the exercise. Put your names in a comment on top of each file. Make sure your code compiles and test it!
- For the submission, you can either upload the single PDF or prepare a ZIP file (ending .zip, .tar.gz or .tgz; not .rar or anything else) containing the single PDF and the code textfile(s) and nothing else. Please do not use directories within the ZIP, i.e., zip the files directly.
- Name all files without spaces.
- Only upload one submission per group. Do not upload several versions, i.e., if you need to resubmit, use the same file name again so that the previous submission is overwritten.