Automated Planning (TDDD48)

Jendrik Seipp
Mika Skjelnes

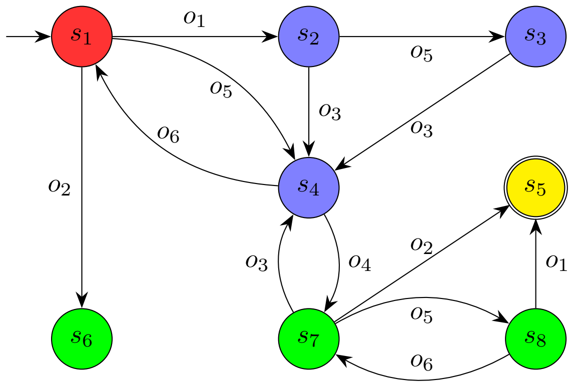
Linköping University

Lab 5

Important: For submission, consult the rules at the end of this document. Non-adherence to these rules might lead to a penalty in the form of a deduction of points. Some points are *bonus points*. These can help you reach the point quota per lab (4/12 points) and the overall point quota ($50\% \cdot 7 \cdot 12 = 42$ points).

Exercise 5.1 (1+1+1 points)

Consider the transition system $\mathcal{T} = \langle S, L, c, T, s_0, S_\star \rangle$ with S, L, T, s_0 and $S_\star$ as depicted below and with $c(o_i) = i$ for all $1 \leq i \leq 6$. Note that this is not a unit-cost task. Instead, each operator costs as much as its index.



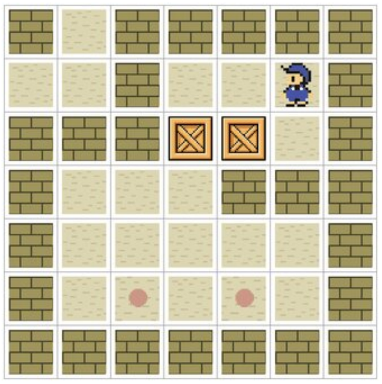
(a) Consider the abstraction α that maps all states depicted in the same color to the same abstract state, i.e., $\alpha(s_1) = s_r$, $\alpha(s_2) = \alpha(s_3) = \alpha(s_4) = s_b$, $\alpha(s_5) = s_y$, and $\alpha(s_6) = \alpha(s_7) = \alpha(s_8) = s_g$. Graphically provide $\mathcal{T}^\alpha$ and give h^α.

(b) Assume you may change the abstraction α from part (a) by mapping one concrete state to another (already existing) abstract state. If you care about having some positive effect on the heuristic quality, which change do you make? Justify your answer. (There are multiple reasonable options.)

(c) Provide an abstraction β of $\mathcal{T}$ such that $|S^\beta| = 4$ and such that there is no abstraction $\beta' \neq \beta$ with $|S^{\beta'}| = 4$ and $h^{\beta'}(s_1) > h^\beta(s_1)$. Graphically provide the transition system $\mathcal{T}^\beta$.

Bonus Exercise 5.2 (0.5+0.5+0.5+0.5=2 bonus points)

In the *Sokoban* domain, a worker has to push boxes to goal positions, but cannot pull them. The figure below illustrates an example problem. The goal is to push one box to each tile indicated by a red dot. (It does not matter which box is located at which position.) In any given state, the worker may move to an empty tile adjacent to its current location, where empty means that tile is neither a wall nor there is a box. If there is a box, the worker may still move there if the tile behind the box (from the worker's perspective) is empty, pushing the box to that empty tile.



In the following, we suggest four abstraction functions for Sokoban problems. While they might seem reasonable at a first glance, all of them come with different practical limitations. Point out and explain the problems with these suggestions in 2–3 sentences each.

(a) α_1: Each state is mapped to the number of boxes that are on a goal location.

(b) α_2: Each state is mapped to an abstract state by ignoring the position of the agent.

(c) α_3: Each state s is mapped to $f(s) \mod n$ where f is a bijection from the set of states S to the natural numbers from 1 to $|S|$ (i.e., $f\colon S \to \{1, \ldots, |S|\}$) and $n = 10^6$ is used to limit the number of abstract states.

Hint: You may assume that it is possible to evaluate $f(s)$ for a given state s efficiently. In fact, we will discuss a specific function of this kind in lecture E6 (perfect hash function for Pattern Database heuristics).

(d) α_4: A state is mapped to s_1 if 5 or fewer moves are necessary to move all boxes to a goal location; it is mapped to s_2 if between 5 and 10 moves are necessary; and so on.

Exercise 5.3 (2+3+1 points)

Note: to simplify implementation details, for the exercises in this part you can assume that the planning tasks that you have to deal with possess a simplified structure. In particular, you can assume that they are SAS$^+$ *tasks with the additional restrictions that (i) for every operator o, the set of state variables that appear in $pre(o)$ is the same as the set of state variables that appear in $eff(o)$, and (ii) the goal formula mentions all the state variables of the problem, which implies that there is one single goal state. The tasks are converted to this simplified form automatically without you having to do anything about it, so you can safely assume that conditions (i) and (ii) always hold. This simplified form, by the way, is called* Transition Normal Form *(TNF), and is useful to make the proofs of theorems and implementation of algorithms easier. You can find more details about the way TNF tasks are represented in the code in file `fast-downward/src/search/planopt_heuristics/tnf_task.h`.*

(a) In the files `fast-downward/src/search/planopt_heuristics/projection.*` you can find an incomplete implementation of a class projecting a TNF task to a given pattern. Complete the implementation by projecting the initial state, the goal state and the operators.

The example task from the lecture and two of its projections are implemented in the files `projection_test.*`. You can use them to test and debug your implementation by calling Fast Downward as `./fast-downward.py --test-projections`. Of course, the tests will fail until you have added your code.

(b) In the files `fast-downward/src/search/planopt_heuristics/pdb.*` you can find an incomplete implementation of a pattern database. Complete the implementation by computing the distances for all abstract states as described in the code comments. You can run your implementation using the heuristic `planopt_pdb(pattern=greedy(1000))` in an A* search. It uses a greedy pattern generation algorithm which results in an abstract state space with at most 1000 states.

You can debug your code by comparing it to the built-in PDB implementation of Fast Downward. The according heuristic is `pdb(pattern=greedy(1000))` which should find the same pattern. Make sure the patterns are identical (printed as "Greedy generator pattern") and that both implementations result in the same amount of expanded states before the last f-layer (printed as "Expanded until last jump").

Examples from the `castle` directory:

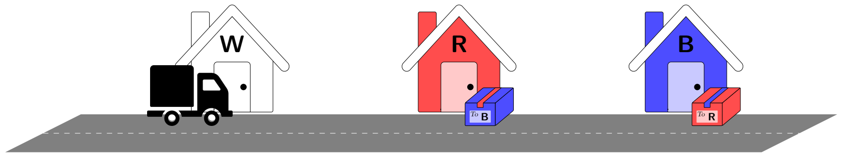
instance	pattern	expanded until last jump
castle-8-5-4-cards	[56, 57, 58, 59, 60, 61, 62, 63, 64]	3718
castle-8-5-10-cards	[57, 58, 59, 60, 61, 62, 63, 64, 65]	744
castle-8-5-13-cards	[56, 57, 58, 59, 60, 61, 62, 63, 64]	6742

(c) Analyze what effect the abstraction size has on the search performance. To do so, run your implementation of the PDB heuristic on all problem instances of the `castle` domain with an abstract state spaces of at most 1000 states as well as at most 100000 states. Then, compare the preprocessing time (i.e., difference between total time and search time), search time, and expanded states before the last f-layer.

The `run-experiment.sh` script can be used as a starting point to run the experiment.

Bonus Exercise 5.4 (2 bonus points)

Consider a logistics problem similar to the running example from the lecture. A single truck needs to pick up and deliver two packages between three locations. The figure below illustrates the initial state of the problem. The color of each package indicates its destination.



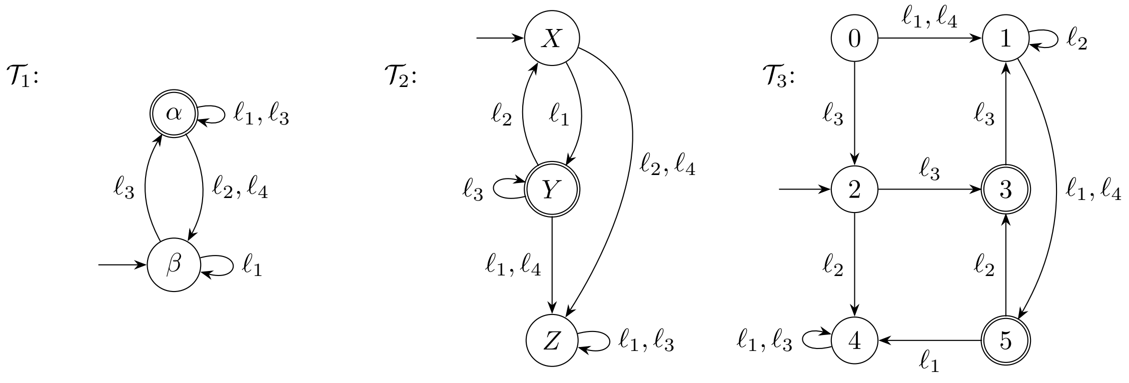
Formally, we can describe this problem as a SAS$^+$planning task $\Pi = \langle V, I, O, \gamma \rangle$ with

- $V = \{t, p_B, p_R\}$ where $dom(t) = \{W, R, B\}$ and $dom(p_B) = dom(p_R) = \{W, R, B, T\}$;
- $I = \{t \mapsto W, p_B \mapsto R, p_R \mapsto B\}$;
- $O = \{move_{o,d} \mid o, d \in \{W, R, B\}, o \neq d\} \cup \{load_{p,l} \mid p \in \{p_B, p_R\}, l \in \{W, R, B\}\} \cup \{unload_{p,l} \mid p \in \{p_B, p_R\}, l \in \{W, R, B\}\}$ where
  - $move_{o,d} = \langle t = o, t := d, 1 \rangle$,
  - $load_{p,l} = \langle t = l \wedge p = l, p := T, 1 \rangle$, and
  - $unload_{p,l} = \langle t = l \wedge p = T, p := l, 1 \rangle$; and
- $\gamma = p_B = B \wedge p_R = R$.

Visualize the factored transition system induced by the atomic projections of Π. Make sure to indicate initial states, goal states, and transition labels in all factors.

Exercise 5.5 (1+1+0.5+0.5 points)

Consider the factored transition system $F = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3\}$ with label set $L = \{\ell_1, \ell_2, \ell_3, \ell_4\}$ and cost function $c(\ell_1) = 1$ and $c(\ell_2) = c(\ell_3) = c(\ell_4) = 2$. The transition systems look as follows:



(a) Graphically provide the synchronized product $\mathcal{T}_1 \otimes \mathcal{T}_2$.

(b) We discussed the f-preserving shrinking strategy in the lecture. It repeatedly combines two states that have the same g- and h-values. Now consider instead h-preserving shrinking which repeatedly combines two states with the same goal distance. Apply h-preserving shrinking to $\mathcal{T}_3$ until no additional states have the same h-value and provide the resulting transition system $\mathcal{T}_3'$.

(c) For all transition systems $\mathcal{T} \in F$, enumerate all pairs $\ell, \ell' \in L$ such that ℓ locally subsumes ℓ' in $\mathcal{T}$.

(d) Based on your results for the previous part: Is there an exact label reduction for F? If yes, provide the functions λ and c'. If no, explain why not.

Submission rules:

- Lab sheets must be submitted in groups of 2–3 students. Clone the labs repo (`https://github.com/mrlab-ai/tddd48-labs`) and push it to a repo at the University GitLab instance `https://gitlab.liu.se`. Make sure the repo is **private** and give read access to Mika Skjelnes (`mika.skjelnes@liu.se`).
- For non-programming exercises, create a single PDF file at the location `labX/solution.pdf`. If you want to submit handwritten solutions, include their scans in the single PDF. Make sure it is in a reasonable resolution so that it is readable. Put the names of all group members on top of the first page. Either use page numbers on all pages 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, directly edit the code in the cloned repository and only create those code text file(s) required by the lab. Put your names in a comment on top of each file. Make sure your code compiles and test it. Code that does not compile or which we cannot successfully execute will not be graded.