Planning and Optimization

M. Helmert, T. Keller
S. Eriksson, F. Pommerening, S. Sievers

University of Basel
Fall Semester 2019

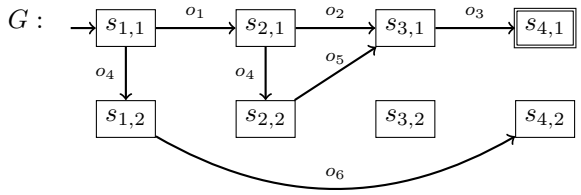
Exercise Sheet D

Due: November 10, 2019

The files required for this exercise are in the directory `exercise-d` *of the course repository (*`https://bitbucket.org/aibasel/planopt-hs19`*). All paths are relative to this directory. Update your clone of the repository with* `hg pull -u` *to see the files. For the runs with Fast Downward, set a time limit of 1 minute and a memory limit of 2 GB. Using Linux, such limits can be set with* `ulimit -t 60` *and* `ulimit -v 2000000`*, respectively.*

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

(a) Consider the following graph G depicting a simple transition system. The cost function is $cost = \{o_1 \mapsto 1, o_2 \mapsto 5, o_3 \mapsto 7, o_4 \mapsto 1, o_5 \mapsto 2, o_6 \mapsto 9\}$. As usual, an incoming arrow indicates the initial state, and goal states are marked by a double rectangle.



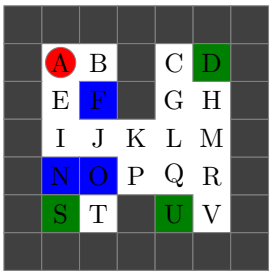
Provide the following graphs:

- a graph G_1 which is isomorphic to G but not the same.
- a graph G_2 which is graph equivalent to G but not isomorphic to it.
- a graph G_3 which is a strict homomorphism of G but not graph equivalent to it.
- a graph G_4 which is a non-strict homomorphism of G but not graph equivalent to it.
- a graph G_5 which is the transition system induced by the abstraction α with $\alpha(s_{x,y}) = x$.
- a graph G_6 which is the induced transition system of a non-trivial coarsening of α.
- a graph G_7 which is the induced transition system of a non-trivial refinement of α.

In *all graphs*, highlight an optimal path and compute its cost. For graphs G_1–G_4, justify (one sentence is enough) why they don't have the property they are not supposed to have, for example, why G_2 is not isomorphic to G.

Bonus exercise: You will recieve one additional mark for this exercise if 6 out of your 7 graphs $G_1, \ldots, G_7$ have a pairwise different shortest path cost.

(b) 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 red dot denotes the initial position of the worker, the blue cells denote the initial positions of the boxes, and the green cells denote the goal positions of the boxes, where it does not matter which box is finally located at which goal position. The letters (A – V) are only shown to indicate the cells.



Point out the problems with the following ideas for abstraction mappings in this domain:

- α_1: Each state is mapped to the number of boxes that are on a goal location.
- α_2: Each state is mapped to an abstract state by ignoring the position of the agent.
- α_3: Each state s is mapped to $f(s)$ modulo n where f is the perfect hash function for all states (see lecture slides D3) and $n = 10^6$ is used to limit the number of abstract states.
- 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.

(c) Let Π be a SAS$^+$ planning task and let P be a pattern for Π. Prove that $\mathcal{T}(\Pi|_P) \overset{\mathrm{G}}{\sim} \mathcal{T}(\Pi)^{\pi_P}$, i.e., $\mathcal{T}(\Pi|_P)$ is graph-equivalent to $\mathcal{T}(\Pi)^{\pi_P}$.

As there are three involved transition systems ($\mathcal{T}(\Pi)$, $\mathcal{T}(\Pi)^{\pi_P}$, $\mathcal{T}(\Pi|_P)$), a clear and consistent notation (variable names, etc.) is important. For full marks, the proof should not only be correct but also easy to follow.

(d) Discuss why the theorem from exercise (c) is relevant. Why would we need to define $\Pi|_P$, if we already saw that π_P is a valid abstraction of $\mathcal{T}(\Pi)$, and hence we could use h^{π_P} as our heuristic?

Exercise D.2 (3+3+1 marks)

Consider the SAS$^+$ representation of the Sokoban problem depicted in exercise 1(b) with variables pos_w, pos_{b1}, pos_{b2}, pos_{b3} (which denote the positions of the worker and the three boxes), $atgoal_{b1}$, $atgoal_{b2}$, $atgoal_{b3}$ (which indicate whether the boxes are at goal positions), and $content_A$, …, $content_V$ (which denote the content of the individual cells). Formally, the variable domains are defined as follows:

- $dom(pos_w) = dom(pos_{b1}) = dom(pos_{b2}) = dom(pos_{b3}) = \{A, \ldots, V\}$
- $dom(atgoal_{b1}) = dom(atgoal_{b2}) = dom(atgoal_{b3}) = \{\mathbf{T}, \mathbf{F}\}$
- $dom(content_A) = \cdots = dom(content_V) = \{empty, w, b1, b2, b3\}$

The initial state is defined by the set consisting of the following mappings:

- $pos_w \mapsto A$, $pos_{b1} \mapsto F$, $pos_{b2} \mapsto O$, $pos_{b3} \mapsto N$, $atgoal_{b1} \mapsto \mathbf{F}$, $atgoal_{b2} \mapsto \mathbf{F}$, $atgoal_{b3} \mapsto \mathbf{F}$
- $content_F \mapsto b1$, $content_O \mapsto b2$, $content_N \mapsto b3$, $content_A \mapsto w$
- $content_x \mapsto empty$ for all $x \in \{A, \ldots, V\} \setminus \{A, F, N, O\}$

The goal is given by the formula $atgoal_{b1} = \mathbf{T} \wedge atgoal_{b2} = \mathbf{T} \wedge atgoal_{b3} = \mathbf{T}$. The operators (*move* and *push*) are defined as usual (recall that it is not allowed to pull boxes). We call cells c and c' *adjacent* if c' is either above, below, left or right to c (i.e., diagonal cells are *not* adjacent).

- *move* operators: For adjacent cells c and c', the worker can move from c to c' if the worker is currently at c and c' is empty. After moving, c is empty and the worker is at c'.

- *push* operators: For cells c, c', c'' such that c is adjacent to c' in direction X iff c' is adjacent to c'' in direction X for $X \in \{\text{above}, \text{below}, \text{left}, \text{right}\}$, the worker can push a box bi from c' to c'' if the worker is at c, the box is at c' and c'' is empty. After pushing, c is empty, the worker is at c', and the box is at c''. $atgoal_{bi}$ is set depending on whether the box moved from a nongoal to a goal position or vice versa.

Consider the pattern collection $\mathcal{C}$ that consists of exactly the following patterns:

- $P_1 = \{atgoal_{b1}, pos_{b1}\}$
- $P_2 = \{atgoal_{b1}, content_H, content_M\}$
- $P_3 = \{atgoal_{b3}, pos_w\}$
- $P_4 = \{atgoal_{b3}, content_A\}$
- $P_5 = \{atgoal_{b2}, pos_{b2}, atgoal_{b3}\}$
- $P_6 = \{pos_{b1}, content_D, content_E\}$
- $P_7 = \{atgoal_{b2}, pos_{b2}\}$

(a) Simplify the collection by removing trivial patterns and causally irrelevant variables from patterns.

(b) Construct the compatibility graph for $\mathcal{C}$ and determine the maximal cliques.

(c) Provide the canonical heuristic $h^{\mathcal{C}}$ and simplify it with help of the Dominated Sum Theorem if possible.

Exercise D.3 (10 marks)

Write a tutorial about *constrained pattern databases* based on the two papers listed below. Your tutorial should enable someone familiar with the planning and optimization course (up to this point) to understand what constrained pattern databases are and how they relate to their non-constrained version in terms of heuristic strength. Use the terminology from the lecture, and use an illustrative example from a domain different than blocksworld. Also explain why variables that are not causally relevant in a pattern might still be interesting to include in patterns for constrained PDBs.
You can include theoretical results from papers directly as long as you cite their origin. But when doing so, add an intuitive explanation of the result and why it holds in your own words. A good answer can be written in 0.75 *pages.*

- Haslum, P., Bonet, B., Geffner, H. (2005). New Admissible Heuristics for Domain-Independent Planning. In Proceedings of AAAI, 1163–1168.
- Haslum, P., Botea, A., Helmert, M., Bonet, B., Koenig, S. (2007). Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In Proceedings of AAAI, 1007–1012.

Exercise D.4 (3+3+4 marks)

In this exercise, you are asked to implement and evaluate a *shrink strategy* in Fast Downward. In the lecture, we have seen that merge-and-shrink is a powerful framework for computing abstractions through the means of applying transformations to factored transition systems. Shrinking is one type of such transformations, and it means to apply an abstraction to a single transition system of the factored transition system. In practice, given a transition system and a size limit imposed on it, the question is *how* to come up with a good abstraction of the factor. This is what a shrink strategy does: it is an algorithm that computes an abstraction of a given transition system so that its new size is guaranteed to obey a given limit.

In Fast Downward, a shrink strategy is given a transition system and an object that allows to retrieve distances for states of the transition system. The strategy then has to compute an equivalence relation over the states, i.e., a partitioning over states. All states of the same equivalence class (or partition) are then mapped to the same new abstract state. Your task will be to implement the logic of the strategy for computing the state equivalence relation (the state partitioning).

(a) In `fast-downward/src/search/merge-and-shrink/shrink_h_preserving.cc` you find an incomplete implementation of a h-preserving shrink strategy that aims at abstracting all states with the same h-value to the same abstract state. It works as follows: first, partition all states of the transition system according to their h-value. Then, iterate over all partitions in increasing order of their h-value and simply assign all states of a partition to the same equivalence class in the result, as long this does not violate the given size limit. If the size of the resulting equivalence relation reaches the given size limit, then the strategy cannot turn each partition into a separate equivalence class anymore, but instead, it assigns all states of all remaining partitions to the last equivalence class created (i.e., the last equivalence class possibly holds states that have different h-values due to the size limit).

You can test your strategy using
`./fast-downward/fast-downward.py --alias mas-h-preserving-x blocks/p1.pddl`
where $x \in \{1, 10, 100, 1000\}$ denotes the size limit imposed on transition systems.

(b) In `fast-downward/src/search/merge-and-shrink/shrink_random.cc` you find an incomplete implementation of a random shrink strategy that abstracts states uniformly at random. This can be implemented as follows: Iterate over all states in a random order. As long as the size of the resulting equivalence relation has not reached the imposed size limit, assign the state to its own equivalence class, thus increasing the size of the resulting equivalence relation by 1. Once the size limit is reached, assign the state to a random equivalence class (in this case, there are exactly as many equivalence classes as the size limit allows).

You can test your strategy using
`./fast-downward/fast-downward.py --alias mas-random-x blocks/p1.pddl`
where $x \in \{1, 10, 100, 1000\}$ denotes the size limit imposed on transition systems.

(c) Evaluate both strategies on the tasks in the directories `gripper` and `blocks`, using all four size limits ($\{1, 10, 100, 1000\}$). For each run, report the runtime of the merge-and-shrink algorithm ("Merge-and-shrink algorithm runtime: "), the total runtime ("Total time: "), and the number of expanded states ("Expanded until last jump: "), which denotes the number of expanded states excluding the last f-layer of the A* search. (On the last f-layer, the number of expanded states only depends on tie-breaking which we don't want to include in our evaluation.) Discuss the results. (In particular, *explain* the results. Note that this can, to some extent, also be done without a working implementation.)

Please structure your table as follows:

Shrink strategy	$h-1$	$h-10$	$h-100$	$h-1000$	$r-1$	$r-10$	$r-100$	$r-1000$
blocks 6-0, M&S:								
blocks 6-0, Exp:								
blocks 6-0, Total:								