Planning and Optimization

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

University of Basel
Fall Semester 2019

Exercise Sheet C

Due: October 27, 2019

The exercise sheets can be submitted in groups of three students. Please submit one single copy of the exercises per group (only one member of the group does the submission), and provide all student names on the submission. The files required for this exercise are in the directory `exercise-c` *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 C.1 (3+2+2 marks)

(a) The delete relaxation of SAS$^+$ is different from propositional tasks because there is no notion of a *negative* effect. Instead of ignoring negative effects, we can consider the variables to have sets of values where new values can only be added. For example, a variable describing the position of an agent will have the value $\{A\}$ if the agent is at A. Moving from A to B in the delete relaxation *adds* the value B to this set, so afterwards the variable has the value $\{A, B\}$ and the agent is considered to be at A and B.

Define this idea formally for arbitrary SAS$^+$ tasks Π. This should include a definition for Π^+ and o^+; for *satisfying a formula* ($s \models \chi$); for *domination* of states (s dominates s'); and a definition of *applying a relaxed operator* ($s[\![o^+]\!]$) in such tasks. Prove that the following results also hold with your definitions:

- Domination Lemma: Let s and s' be states and χ a formula without negation symbols. If $s \models \chi$ and s' dominates s then $s' \models \chi$.
- Monotonicity lemma: Let s be a state and o an operator. Then $s[\![o^+]\!]$ dominates s.

Refer to the proofs presented in the lecture for parts of the proofs that remain the same.

(b) Provide unit-cost planning tasks in STRIPS with the following characteristics. In cases where this is not possible, explain why and use positive normal form instead of STRIPS. If this is also not possible, justify why no such task can exist:

(i) A task Π_1 with 2 operators, such that Π_1^+ has an optimal plan cost of 3.

(ii) A task Π_2 with 2 variables, such that Π_2^+ has an optimal plan cost of 3.

(iii) A task Π_3 with at least one plan with cost 3, where Π_3^+ is unsolvable.

(iv) An infinite family of planning tasks $P = \{P_1, P_2, \ldots\}$ (the definition of P_i is parametrized by the value of the integer parameter i) such that the optimal plan cost of P_i is twice the cost of an optimal plan for P_i^+ for all i.

(c) Take the simple instance of the *Visitall* domain (from the International Planning Competition) in directory `visitall-untyped`, and make sure you understand the problem. What is the optimal solution value $h^*(I)$? What is the value of $h^+(I)$? Draw the full Relaxed Task Graph corresponding to the instance, and label each node with the final cost that results from (manually) applying the algorithm seen in class for computing $h^{\max}$. What is the value of $h^{\max}(I)$? Finally, label the graph again, but with the costs that result from h^{add} instead of $h^{\max}$.

Exercise C.2 (3+3+4+2+3+2 marks)

(a) The files `fast-downward/src/search/planopt_heuristics/and_or_graph.*` contain an implementation of an AND/OR graph. Implement the so-called *generalized Dijkstra's algorithm* in the method `most_conservative_valuation` to find the most conservative valuation of a given AND/OR graph by following the approach outlined in the code comments.

The example graphs from the lecture are implemented in the method `test_and_or_graphs`*. You can use them to test and debug your implementation by calling Fast Downward as* `./fast-downward.py --test-and-or-graphs`.

(b) The files `fast-downward/src/search/planopt_heuristics/relaxed_task_graph.*` contain a partial implementation of a relaxed task graph for STRIPS tasks. Complete it by constructing the appropriate AND/OR nodes and edges between them in the constructor. Also complete the method `is_goal_relaxed_reachable` by querying the AND/OR graph.

You can use the heuristic `planopt_relaxed_task_graph()` *(which prunes states that are not relaxed solvable) to test your implementation.*

(c) Modify the construction of the relaxed task graph by setting the variable `direct_cost` of each operator effect node you create to the actual cost of the operator.

Then implement the method `weighted_most_conservative_valuation` for AND/OR graphs to compute h^{add} by following the approach outlined in the code comments. Use a comment to point out the change you would have to make to turn this into a computation for $h^{\max}$.

Finally, implement the method `additive_cost_of_goal` of the relaxed task graph class to return the h^{add} value of the task based on the implementation above.

(d) The heuristic `planopt_add()` uses your implementation from exercise (d) as heuristic values. Use it in an eager greedy search on the instances in the directory `sokoban`. Which of the instances are relaxed solvable? Which ones can you solve with this heuristic within the resource limits? Compare the heuristic values of the initial state with the cost of an optimal relaxed plan, the discovered plan and an optimal plan.

You can compute optimal relaxed plans by explicitly creating the delete relaxation of the task and solving it with an optimal search algorithm. This can be done with the Fast Downward options `--search "astar(lmcut())" --translate-options --relaxed`*. This is not the ideal way of computing optimal relaxed plans, so it will not complete on all instances. If the search does not complete, the last reached f-layer is a lower bound to the optimal relaxed solution cost.*

The values of `planopt_add()` *and the built-in implementation of Fast Downward (*`add()`*) should match, so you can use the built-in implementation for debugging exercise 2(d).*

(e) Modify your solution of exercise (d) so that every time you reduce the cost of an OR node, the ID of the responsible successor is stored in the `achiever` field of the OR node.

Then implement the method `ff_cost_of_goal` by collecting all best achievers. Start from the goal node and recursively collect all successors of each encountered AND node and the stored best achiever from each encountered OR node. Return the sum of direct costs of all collected nodes.

(f) The heuristic `planopt_ff()` uses your implementation from exercise (f) as heuristic values. Use it in an eager greedy search on the instances in the directory `sokoban` and compare the heuristic values of the initial state with the cost of an optimal relaxed plan, the discovered plan and an optimal plan. Also compare the results to the results of exercise 2(e).

The values of `planopt_ff()` *and the built-in implementation of Fast Downward (*`ff()`*) are not guaranteed to match, but should lead to similar results on these benchmarks.*

Exercise C.3 (6 marks)

We consider four variants of the problem from exercise A.1. All of them are defined based on a directed graph $G = \langle N, E \rangle$, start and target location $s, t \in N$ for the agent, a set of locked doors $L \subseteq E$, and a set of keys K. Each key $k \in K$ has an initial location *initial-location*$(k) \in N$ and fits in a subset of doors *fits*$(k) \subseteq L$. Note that in this general form one key can unlock multiple doors but we will restrict that to a single door in some of the variants. In all variants, the set of variables is $V = \{\texttt{at-}x \mid x \in N\} \cup \{\texttt{unlocked-}x\texttt{-}y \mid \langle x, y \rangle \in E\} \cup \{k\texttt{-at-}x \mid k \in K, x \in N\} \cup \{\texttt{holding-}k \mid k \in K\} \cup \{\texttt{usable-}k \mid k \in K\}$. The initial state sets the following variables to true and all other variables to false: $\texttt{at-}s$; $\texttt{unlocked-}x\texttt{-}y$ for each $\langle x, y \rangle \in E \setminus L$; and $\texttt{usable-}k$ for each $k \in K$; $k\texttt{-at-}\textit{initial-location}(k)$ for each $k \in K$. The goal in all cases is $\texttt{at-}t$. We assume that all tasks are solvable and use the following operators to define the variants. We define operators in terms of three sets *pre*(o), *add*(o), and *del*(o) as mentioned in lecture A6 (slide 23).

Operator o	*pre*(o)	*add*(o)	*del*(o)
$\texttt{move-}x\texttt{-}y$	$\{\texttt{at-}x, \texttt{unlocked-}x\texttt{-}y\}$	$\{\texttt{at-}y\}$	$\{\texttt{at-}x\}$
$\texttt{move-back-}x\texttt{-}y$	$\{\texttt{at-}y, \texttt{unlocked-}x\texttt{-}y\}$	$\{\texttt{at-}x\}$	$\{\texttt{at-}y\}$
$\texttt{pick-up-}x\texttt{-}k$	$\{\texttt{at-}x, k\texttt{-at-}x\}$	$\{\texttt{holding-}k\}$	$\{k\texttt{-at-}x\}$
$\texttt{drop-}x\texttt{-}k$	$\{\texttt{at-}x, \texttt{holding-}k\}$	$\{k\texttt{-at-}x\}$	$\{\texttt{holding-}k\}$
$\texttt{unlock-}x\texttt{-}y\texttt{-}k$	$\{\texttt{at-}x, \texttt{holding-}k\}$	$\{\texttt{unlocked-}x\texttt{-}y\}$	$\{\}$
$\texttt{lock-}x\texttt{-}y\texttt{-}k$	$\{\texttt{at-}x, \texttt{holding-}k\}$	$\{\}$	$\{\texttt{unlocked-}x\texttt{-}y\}$
$\texttt{unlock-otk-}x\texttt{-}y\texttt{-}k$	$\{\texttt{at-}x, \texttt{holding-}k, \texttt{usable-}k\}$	$\{\texttt{unlocked-}x\texttt{-}y\}$	$\{\texttt{usable-}k\}$

We group the operators as follows:

$$\begin{aligned}
O_M &= \{\texttt{move-}x\texttt{-}y \mid \langle x, y \rangle \in E\} \\
O_{M^{-1}} &= \{\texttt{move-back-}y\texttt{-}x \mid \langle x, y \rangle \in E\} \\
O_P &= \{\texttt{pick-up-}x\texttt{-}k \mid x \in N, k \in K\} \\
O_D &= \{\texttt{drop-}x\texttt{-}k \mid x \in N, k \in K\} \\
O_U &= \{\texttt{unlock-}x\texttt{-}y\texttt{-}k \mid k \in K, \langle x, y \rangle \in \textit{fits}(k)\} \\
O_L &= \{\texttt{lock-}x\texttt{-}y\texttt{-}k \mid k \in K, \langle x, y \rangle \in \textit{fits}(k)\} \\
O_{UO} &= \{\texttt{unlock-otk-}x\texttt{-}y\texttt{-}k \mid k \in K, \langle x, y \rangle \in \textit{fits}(k)\}
\end{aligned}$$

The variants are distinguished by the allowed actions and whether keys can fit multiple doors.

Variant 1: $|K| = |L| = 0$ and $O_1 = O_M$.

Variant 2: $|\textit{fits}(k)| \geq 1$ for all $k \in K$ and $O_2 = O_M \cup O_P \cup O_{UO}$.

Variant 3: $|\textit{fits}(k)| = 1$ for all $k \in K$ and $O_3 = O_M \cup O_{M^{-1}} \cup O_P \cup O_U$.

Variant 4: $|\textit{fits}(k)| = 1$ for all $k \in K$ and $O_4 = O_M \cup O_{M^{-1}} \cup O_P \cup O_U \cup O_D \cup O_L$.

How well would you expect enforced hill-climbing with the optimal delete relaxation heuristic (h^+) to work in each of these cases? Justify your answer by discussing the heuristic error of h^+ in each of these cases and the properties of the search spaces as defined in the following paper.

- Hoffmann, J. (2001). Local Search Topology in Planning Benchmarks: An Empirical Analysis. In Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), 453–458.

A good answer can be written in 0.5–1 *page and does not require formal proofs.*