Planning and Optimization

M. Helmert, G. Röger F. Pommerening University of Basel Fall Semester 2017

Exercise Sheet C Due: November 5, 2017

The files required for this exercise are in the directory exercise-c 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. 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+1+2+2 marks)

- (a) Define a suitable notion of delete relaxation for SAS⁺ tasks where variables accumulate values. This should include a definition for *states* in the delete relaxation; for *satisfying a formula* (s ⊨ χ); 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.
 - Relaxation lemma: Let s be a state, and let s' be a state that dominates s.
 - 1. If o is an operator applicable in s, then o^+ is applicable in s' and $s' \llbracket o^+ \rrbracket$ dominates $s \llbracket o \rrbracket$.
 - 2. If π is an operator sequence applicable in s, then π^+ is applicable in s' and $s'[\pi^+]$ dominates $s[\pi]$.
 - If additionally π leads to a goal state from s, then π⁺ leads to a goal state from state s'.

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

- (b) Provide a planning task Π in positive normal form where Π is unsolvable and Π^+ is solvable.
- (c) In the files fast-downward/src/search/planopt_heuristics/h_greedy_relaxed_plan.* you can find an incomplete implementation of a heuristic that estimates the goal distance as the cost of a greedily computed relaxed plan. Complete the implementation by applying operators that are applicable in the relaxed task until the goal is reached or until there are no more changes. Do not use operators that do not add anything new.
- (d) The heuristic from exercise (c) can be used in a greedy search with the option --search "eager_greedy([planopt_greedy_relaxed()])". Run Fast Downward on the instances in the directory castle. 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.

Exercise C.2 (4+4+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.

The heuristic planopt_relaxed_task_graph() uses your implementation to prune states that are not relaxed solvable. Use it in an A*-search on the instances in the directory castle and compare it to blind search by the number and speed of expansions.

- (c) Let Π^+ be a relaxed planning task and let $N_{\mathbf{T}}$ be the forced true nodes of $\operatorname{RTG}(\Pi^+)$. Prove that for all formulas over state variables φ that occur in the definition of Π^+ : φ is true in some reachable state of Π^+ iff $n_{\varphi} \in N_{\mathbf{T}}$.
- (d) Modify the construction of the relaxed task graph by setting the cost of each operator as the direct_cost of its effect node.

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.

(e) 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 castle 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 1(d).

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).

(f) 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.

(g) 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 castle 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 exercises 1(d) and 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.

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