SAARLAND UNIVERSITY
Prof. Jörg Hoffmann, Dr. Daniel Fišer
Thorsten Klößner, Pascal Lauer

AI Programming Project Description — Project 2

In this project, your task is to implement the pattern database heuristic and the cliques heuristic. For a planning task $\Pi = (V, A, I, G)$ and a pattern (subset of variables) $P \subseteq V$, the pattern database (PDB) heuristic is the abstraction heuristic $h^P = h^{\pi_P}$, induced by the abstraction function $\pi_P(s) := s|_P$ on Θ_Π, where $s|_P$ is the restriction of s to P. The cliques heuristic for a set of patterns $\mathcal{C}$ is defined as $h^{\mathcal{C}}(s) := \max_{D \in cliques(\mathcal{C})} \sum_{P \in D} h^P$, where $cliques(\mathcal{C})$ is the set of all maximal cliques in the combinability graph of $\mathcal{C}$.

Implementing the PDB heuristic will give you 20 points, while implementing the cliques heuristic is worth 5 points.

1 Obtaining the Project

To obtain the project, please pull the `main` branch of the fork repository by running `git pull --no-edit fork main`.

2 Relevant Files

For this project, only the classical planning module `downward` is relevant. Recall that the headers and sources of all modules are located in the sub-folders `include` (headers) and `src` (sources). The entities relevant for this project are the same as for project 0 and 1, with the following additional ones.

Representing $\Pi|_P$ The class `SyntacticProjection` implements a `ClassicalTask` that is the syntactic projection $\Pi|_P$ of another `ClassicalTask` Π with respect to a pattern P. Beyond implementing the `ClassicalTask` interface, it provides the methods `compute_index` and `get_state_for_index`, which can be used to map the states of the syntactic projection to unique indices in $\{0, \ldots, |S| - 1\}$ and vice versa. This is useful for storing states of the projection as indices. The method `get_num_states` also provides access to the number of states in the projection.

Computing maximum cliques The function `max_cliques::compute_max_cliques` computes the maximal cliques in a graph. It is only relevant for the cliques heuristic $h^{\mathcal{C}}$.

General Utility There are two utility functions which you can use to check if an action is applicable in a state, or if a state is a goal state.

3 Implementation

To implement the heuristics, you need to complete the implementations of the classes `PDBHeuristic`, which represents h^P, and `CliquesHeuristic`, which represents $h^{\mathcal{C}}$. These classes are defined in the `heuristics` directory of the classical planning module, namely inside the files `pdb_heuristic.{h,cc}` and in `cliques_heuristic.{h,cc}`, respectively.

To solve the tasks, you may edit the header and source files `pdb_heuristic.{h,cc}` and `cliques_heuristic.{h,cc}` how you see fit, as long as the implementation classes `PDBHeuristic` and `CliquesHeuristic` remain and already provided members are not removed and their accessibility is not changed. You may additionally add your own files to the `students` module, in case this is needed (see main page of the software documentation for details). **All other changes will be discarded on our test server.**

3.1 Implementation of h^P (20 Points)

Before implementing $h^{\mathcal{C}}$, consider h^P first. The class `PDBHeuristic` receives the pattern P in its constructor. In the constructor (!), you shall compute a lookup table that associates each abstract state s of the syntactic projection $\Pi|_P$ with its perfect heuristic value $h^*(s)$ in the syntactic projection. To enumerate the abstract states of $\Pi|_P$, the helper class `SyntacticProjection` will prove useful. To compute the lookup table, proceed in two steps.

As a first step, reduce the problem of computing h^* for $\Pi|_P$ to a regression search. The idea is to start the regression from the abstract goal states and compute their minimum distance to all other abstract states. To this end, construct a weighted directed graph $\mathcal{G} = (V, E)$ similar to the transition system of $\Pi|_P$, but with edges that go into the opposite direction. We call this graph the *regression graph*. The vertices V of this graph are the abstract states S of the syntactic projection. The edge relation is defined by $E = \{(s[\![a]\!], s) \mid s \in S, a \in A(s)\}$, and the edge weights are specified by $w(s[\![a]\!], s) = c(a)$.

After constructing the regression graph, use Dijkstra's algorithm on $\mathcal{G}$ to compute the minimum distance from the abstract goal states to every abstract state of the problem, which will effectively compute a lookup table for h^* of $\Pi|_P$. While Dijkstra's algorithm usually assumes a single source state, you can extend it to multiple source states by initializing the search queue with the full set of abstract goal states and initializing the distance table entry for all abstract goal states states to 0.

After computing the lookup table in the constructor, store it so you can access the h^* values of the abstract states during computation of the heuristic. Next, you must implement the inherited method `int Heuristic::compute_heuristic(const State& s)`, which shall return $h^P(s) = h^*(s|_P)$ for the input state s. Here, you must first translate s to the

abstract state $s|_P$, before you look up the perfect heuristic value for $s|_P$ in the previously constructed lookup table.

Remarks We require you to implement the lookup table computation for h^* of the projection in the constructor of the `PDBHeuristic`, as the lookup table shall be computed only once. Constructing the lookup table in the `compute_heuristic` method will re-compute the lookup table each time a new state is evaluated, which is extraordinarily inefficient and will lead to timeouts in our tests.

3.2 Implementation of $h^{\mathcal{C}}$ (5 Points)

After implementing h^P, turn to $h^{\mathcal{C}}$. The class `CliquesHeuristic` has a constructor that receives the set of patterns $\mathcal{C}$ as a parameter. In the constructor (!), you should first compute the lookup tables for every pattern and store them for the heuristic computation. To this end, you can reuse your code from the previous implementation. Furthermore, compute the maximal cliques $cliques(\mathcal{C})$ of the combinability graph of $\mathcal{C}$ and save them for later. To this end, use the helper function `max_cliques::compute_max_cliques`, which receives a graph in a specific format as input and computes the maximal cliques in this graph.

Afterwards, implement the inherited method `int Heuristic::compute_heuristic(const State& s)`, which shall return $h^{\mathcal{C}}(s) := \max_{D \in cliques(\mathcal{C})} \sum_{P \in D} h^P(s)$ for the input state s.

Remarks Once again, we require you to implement the lookup table computations, as well as the maximal clique computation in the constructor of the class, so that everything is precomputetd once. Construction in the `compute_heuristic` method will be extraordinarily inefficient and will lead to timeouts in our tests.

4 Grading

The implementations are evaluated using public tests, which you may execute and inspect at any time, and daily tests which run at least once per day when you push a change to your repository. In order to obtain points for a heuristic, you need to pass all public tests for it. If that is the case, you will receive points for your implementation according to the ratio of daily tests you passed, i.e. if you pass 20% of the daily tests for h^P (and pass all public tests), you receive $20\% * 20 = 4$ points.

The deadline for this project is **December 22nd, 23:59**. We will use the last revision that was committed before the deadline in the main branch of your repository as a basis for grading.

5 Running the Public Tests

All public tests of all projects reside in the `tests/public` subfolder of the repository. The tests of this specific project are located in the subfolder `heuristic_tests`, and are implemented in the files `pdb_tests.cc` and `cliques_heuristic_tests.cc`.

All public tests are located in the `tests` module, which is split up into an `include` and `src` directory, like all other modules. The tests of this specific project are located in the subfolder `src/tests/heuristic_tests`, specifically in the files `pdb_tests.cc` and `cliques_heuristic_tests.cc`. The public tests for both heuristics evaluate your heuristic implementations on example states for a sample planning task and check whether the result matches the correct heuristic value for that state. The five sample problems used in the public tests are documented on our documentation page and reside in the `tasks` subfolder of the `tests` module.

Inside Visual Studio Code, all public tests defined by the project should be listed in the test panel after refreshing the panel. From here, you can run individual tests by clicking on the run button of a test, or run multiple tests at once. You can also debug specific tests from here, so that the program halts at the breakpoints you set in your implementation or in the public tests.

Alternatively, you can run the public tests from the command line via the test script `run_tests.py`. To this end, run either `.\run_tests.py -f PDBHeuristic` for the h^P tests, or `.\run_tests.py -f CliquesHeuristic` for the $h^\mathcal{C}$ tests from a terminal which has the project root directory as the current working directory. To list tests instead of running them, append the `-l` option. To run only the test *test_name*, run `.\run_tests.py -f` *test_name*. To show verbose output, including printouts for tests that pass, specify the option `-v`.