# Comp3620/Comp6320 Artificial Intelligence
# Assignment 3: Planning via Forward Heuristic Search

May 4, 2017

## Automated Planning

Automated Planning is the model-based, domain-independent approach to the problem of action selection. Given an initial state, a planning problem consists in finding a sequence of actions that brings the agent into a state in which the goal is satisfied. To the execution of each action is often associated a cost, and in optimal planning we are interested in the cheapest plan.

In this assignment you will implement a heuristic search planner based on forward exploration of the state space. The planner will be able to take as an input a description of the planning problem in the PDDL language (Planning Domain Description Language), and give as an output a total-ordered sequence of actions. PDDL is the standard de-facto language to express planning problems. For a brief introduction to PDDL, have a look at the following PDDL Tutorial: `http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html`.

In this assignment we target the *typed STRIPS* fragment of this language which was presented in the lectures. Recall that in the STRIPS representation:

- Actions are instances of operators (also known as action schemas)
- Actions have (positive) preconditions and add/delete effects, represented as sets (conjunctions) of ground atoms (facts)
- The initial state is represented as a set of ground atoms (facts) under the Closed World Assumption (the facts that are not stated are false)
- The goal is represented as a set of facts, under the Open World Assumption (the facts that are not stated may be either true or false)

PDDL Planning problems come with two parts (often split in two files), respectively representing the domain description (predicates, operators) and the problem instance description (objects, initial state and goal). There are a number of PDDL planning domains together with a set of problem instances for each under the "benchmarks" folder. Please have a look at them.

To make your life easier, we provide you with a Python infrastructure for a planner which implements the backbone of your planning system, and includes parsing PDDL, grounding predicates and operators, and basic data-structures for your implementation. Your task is to complete some of the interesting aspects of this planning system. At the end of the assignment, you will have a PDDL planner that given a PDDL description of your problem, automatically computes sub-optimal or optimal solutions.

The PDDL planner can be run using the following command from the code folder:

```
./pddl_planner.py [domain-file] problem-file -H <heuristic> -s <search-strategy>
```

The only compulsory file to be provided to the planner is the problem-file, through which the planner tries to retrieve the location of the domain file.

In this assignment, **the operators in your problem description have been already grounded into a set of actions.** The grounding operation is supplied with the basic implementation.

This assignment focuses on four different aspects:

1. Implementing the Action Model
2. Search for Automated Planning
3. Domain-Independent Inadmissible Heuristics for Planning
4. Domain-Independent Admissible Heuristics for Planning

These aspects are split in a number of exercises (from Q1-Q4). Note that Q3 and Q4 can be done independently of each other.

# 1 Q1 - Implementing the Action Model (10 marks)

When opening the file `planning_task.py`, you will find a number of methods that need to be completed (currently the exception "notImplementedError" is raised). Use your knowledge of the STRIPS representation studied during the lectures to implement the following methods:

- `applicable(self,state)` returns true iff an action is applicable in a state (i.e. iff its preconditions are satisfied in that state)
- `apply(self,state)` which, if the action is applicable, returns the resulting state, computed from the action add/delete-effects using the strips rule
- `goal_reached(self,state)` which states whether a state satisfies the goal
- `get_successor_states(self,state)` which returns the set of successor states of a given state, taking into account all applicable actions.

Remember that these operations are crucial to obtain plans that are actually correct with respect to the action model. Any mistake may cause the agent to find plans that are not valid.

Full marks are granted whenever the provided implementation is correct, i.e., it correctly implements the semantics of STRIPS.

# 2 Q2 - Implementing the Search Engine of a Forward State Space Planner (30 Marks)

In this part of the assignment you are asked to implement two search strategies for planning via forward search: A* and Greedy Search (GS). These two strategies are to be used for solving planning optimally (A*, as long as the algorithm is run with an admissible heuristic), and sub-optimally (GS, here there is no guarantee of optimality). Optimal planning is of course harder, but provides guarantees on the obtained solution; sub-optimal planning - also called satisficing planning - is way faster but no guarantees of optimality are provided. For this assignment, optimality corresponds to finding shortest plans; every action is assumed to have a cost equal to one. The algorithms you need to implement here are adaptations of the algorithms you have seen during the Search assignment. Though, you need to be careful in how you structure the planning search space. Both strategies need to explore the search space in a forward state space manner. Yet they differ on which portions of this search space are explored during the search.

For the implementation of these algorithms, you need to use the functions that are in `planning_task.py`; some of the methods in this class are the ones you have implemented for Q1. You may find some of the tools in `code/search/searchspace.py` useful.

The current infrastructure of the planner is provided with a very simple heuristic assigning 1 and 0 respectively to non-goal and goal states. This is the default heuristic called by both place-holder methods. The heuristic produces an estimate by just calling it with its constructor `BlindHeuristic(self,Node)`.

We expect you to provide the two implementations of the algorithms in the following files:

- `search/a_star.py`
- `search/gbfs.py`

Full marks are given for correct implementations of these strategies.

At this point of the assignment, you have a planner that can solve the PDDL benchmarks provided with blind optimal or suboptimal search.

# 3 Q3 - Implementing the FF Heuristic (30 marks)

Albeit quite restrictive (single agent, deterministic actions and complete observability), the fragment of planning we are studying is very hard in general - finding a plan for a grounded STRIPS representation is a PSPACE-complete problem. Yet, this complexity does not arise in many practical problems, as planners are capable of exploiting the problem structure in an automatic way and can still scale up. One such ways is computing sophisticated heuristics obtained automatically from a relaxed version of the problem.

In this part of the assignment you are asked to implement (a part of) a well known heuristic for STRIPS Planning: the FF heuristic, introduced in the lectures.

The heuristic is computed in two stages; the first does a reachability step using a relaxed version of the problem; the second extracts a relaxed solution for that relaxed problem. This solution is used to guide the search for a solution to the "real" planning task. In this planning system the heuristic is to be called to evaluate each state, so a reasonably efficient implementation of the heuristic is important.

We provide you with the reachability step (which is quite complicated), and leave you the task of completing the relaxed plan extraction part of the heuristic. In order to implement such a heuristic you should have very clear what the reachability analysis (within the reachability procedure) is computing. For this purpose, we highly suggest you to have look at that part of the code, including at its comments. The implementation of the heuristic has to be reasonably consistent with what presented during the lecture.

The implementation goes in the `calc_goal_h` method within `rel_based_heuristics.py` in the heuristics folder.

Full marks are granted to a correct implementation of the heuristic, and substantial evidence of its positive impact on the search strategy. For this purpose, choose two domains from the benchmarks folder (the ones you like the most), and study experimentally the impact of the heuristic for the first 15 instances. During these experiments you need to run the FF heuristic with GS. Run the experiments with and without the heuristic and study its impact. Provide a table of the results in a file, i.e. `hff_evaluation.md`, using markdown table formatting. The table needs to contain for each instance and for each domain the following information:

- Run-Time
- Node expansion (note that you should be able to print the node expansion in your algorithm)
- Plan Length

In the `hff_evaluation.md` file we also expect you to provide a thorough explanation reporting your interpretation of the experiments.

The FF heuristic can be activated through the option `-h hff` when invoking the planner. Your search strategy will call the heuristic for each state encountered during the search.

# 4 Q4 - Come up with your heuristic (30 marks)

In this part of the assignment you will implement an admissible heuristic of your choice for planning. You are not required that such a heuristic is consistent, as long as your search algorithm takes this into account[1].

The heuristic implementation is to be provided in the `__call__(self,node)` function in the hadm class, and can be called using option `-h hadm` when invoking the planner.

As for the previous exercise, once your implementation is done, run an experimental analysis over two domains of your choice using A* with and without hadm. We expect you to provide a table and thorough interpretation of the experiments in the file `hadm_evaluation.md`. We expect a reasonable reduction in the number of nodes with overall run time saving (compared with the blind heuristic) for the majority of the instances under consideration (especially the large ones).

It is up to you to decide whether or not to use the relaxation framework we have provided to build the FF heuristic.

[1] Remember that it suffices for the algorithm to reconsider states already in the explored-set, but for which a smaller path has been found.