Université Grenoble Alpes

PATIA M1 INFO – 2022-2023

Responsible : Humbert Fiorino

Daana ROC
Joan Besante

Date : April 2023

You can find the Git repository for Project 1 (Graph Coloring) at the following link: https://github.com/daana27/projects_patia.git.

Project 1 (Graph Coloring)

After cloning the repository, you can access the domain and problem files for the graph coloring project in the GraphColoring directory located at the root of the repository.

Project 2 (Sokoban)

This project consists of two submodules: "pddl4j" and a forked version of "sokoban". "pddl4j" can be found on GitHub at https://github.com/pellierd/pddl4j.git, while "sokoban" is available at https://github.com/daana27/sokoban.git

Additionally, the project includes a bash script (sokoban.sh) that accepts a JSON file as input. This file will first be parsed and encoded as a PDDL problem using the ProblemParser project. Subsequently, the submodule "pddl4j" will be used to generate a solution for the given problem. The output of the "pddl4j" project, along with the original JSON file, will then be passed to the "sokoban" project for further processing. To facilitate this, we modified the "sokoban" project code to accept a file as an argument.

Dependencies: Maven, Gradle, Java 17

The folder test at the root of the project contains a list of tests to be passed to the project.

To run the script, follow these steps:

- Navigate to the project directory and execute the following command: "bash sokoban.sh test/test1.json"

This will run the script with the "test1.json" file as input. You can replace "test1.json" with the name of any other test file you wish to use. The script will parse the input file, generate a solution using the "pddl4j" submodule, and encode it using the "sokoban" submodule. The output will be displayed at http://localhost:8888/test.html

Projet 3 : (SAT Planner)

The project is divided into two classes: SATMain and SATEncoding.

Unfortunately, we were unable to complete the project because the generated plans were not always valid, and we spent a lot of time debugging without success. For the basic problem of a robot moving from room A to room B(p1 and d1), the solver generated a good plan. We make another problem, the same but the robot needs to move the box from A to B, but the solver doesn't make a valid plan. We believe that the bug is in our SATEncoding class.

The program is structured as follows:

The **SATEncoding** class generates the SAT encoding of the PDDL problem. The **SATMain** class uses SATEncoding to encode the problem and then solve it using SAT4J. We then decode the result generated by sat4j to a pddl4j plan that we display at runtime.

To encode the problem, the program uses org.sat4j.core.VecInt to represent clauses and org.sat4j.minisat.SolverFactory to create an instance of the SAT solver. The program then adds the clauses to the solver using the addClause method and checks if the problem is satisfiable using the isSatisfiable method.

The program uses the StateHeuristic of pddl4j (FAST_FORWARD) to estimate the number of steps needed to solve the problem. If the problem is satisfiable, we then decode the result if not we return null.

The SATEncoding class has a constructor that takes a PDDL problem. The encode method takes an integer nbSteps as an argument, which represents the number of planning steps to encode. The method returns a list of lists of integers, where each list represents a clause.

Encoding Phases:

- The first part of the encode method encodes the initial state of the problem. It iterates over the fluents (propositional variables) of the problem and adds a clause that is satisfied by the initial state. If a fluent is true in the initial state, it is added to the clause with a positive sign, otherwise, it is added with a negative sign.

- The second part of the encode method encodes the actions of the problem. We iterate over the actions and encode their preconditions, positive effects, and negative effects. For each precondition of an action, a clause is added that requires the action to be applied if the precondition is true. For each positive effect of an action, a clause is added that requires the effect to be true if the action is applied. For each negative effect of an action, a clause is added that requires the effect to be false if the action is applied.

- The third part of the encode method encodes the transitions. We create a mapping between each fluent and the actions that can modify it. For each fluent that can be modified by multiple actions, a clause is added that requires at most one of the actions to be applied.
- The fourth part of the encoding process involves creating disjunctions for the actions. For all pairs of distinct actions, we create a new clause that requires at least one of these actions to be applied. This clause is then added to the list of clauses that encode the problem.
- The last part is the encoding of the goal of the problem. For each positive fluent in the goal, a new clause is created and added to the list of clauses.