IT3105: Project 2 – Building a General Constraint-Based Puzzle Solver

Due on Wednesday, October 24, 2013

Lecturer: Keith Downing

Pablo Liste Garcia & Dominik Horb

Contents

1 Architecture Description

The general structure we chose to use for our implementation of the general puzzle solver doesn't deviate much from the proposed architecture in the task description. Figure 1 shows the two most important packages we created and their contained types. As you can see the main additions we made are *PuzzleState*, *Conflict* and *GeneralPuzzleSolver*. Our main class that creates the state manager and algorithm objects, initiates the execution and calculates the statistics is *GeneralPuzzleSolver*. It could probably be refactored to separate the different tasks it has a bit more cleanly and distribute the problem specific initialization steps, but this is trivially achievable and wasn't a priority for now. To start an algorithm it creates an object of the desired *LocalStateManager* for the puzzle that should be solved, provides it to an object of the chosen algorithm and starts the algorithm as can be seen in Listing 1.

```
1  this.puzzle = this.choosePuzzle();
2  this.searcher = this.chooseSearchAlgorithm();
3  this.searcher.setStateManager(this.puzzle);
4
5  this.numberOfRuns = this.chooseNumberOfRuns();
6
7  PuzzleState currentSolution;
8
9  for (int i = 0; i < this.numberOfRuns; i++) {
10   currentSolution = this.searcher.run();
11   currentSolution.display();
12 }
```

Listing 1: Combining algorithm and puzzle and starting the search.

The more important bits can be found in the *generalpuzzlesolver.puzzle* package. All three types that can be seen in the lower part of Figure 1 must be implemented by every puzzle that should be added to the application. The algorithm classes will only interact with methods that can be found within theses three types in order to generalize their execution.
Conflict is simply an interface without any methods, but necessary to pass around specific conflicts. For the graph colouring problem an implementation would for example store the indices of two nodes that have the same colour and an edge in the graph.
As the name suggests, an implementation of *PuzzleState* would simply store the data of the current state of the puzzle. It also needs to be able to check if it violates any constraints of the puzzle, provide a list of all the conflicts that exist and display itself.

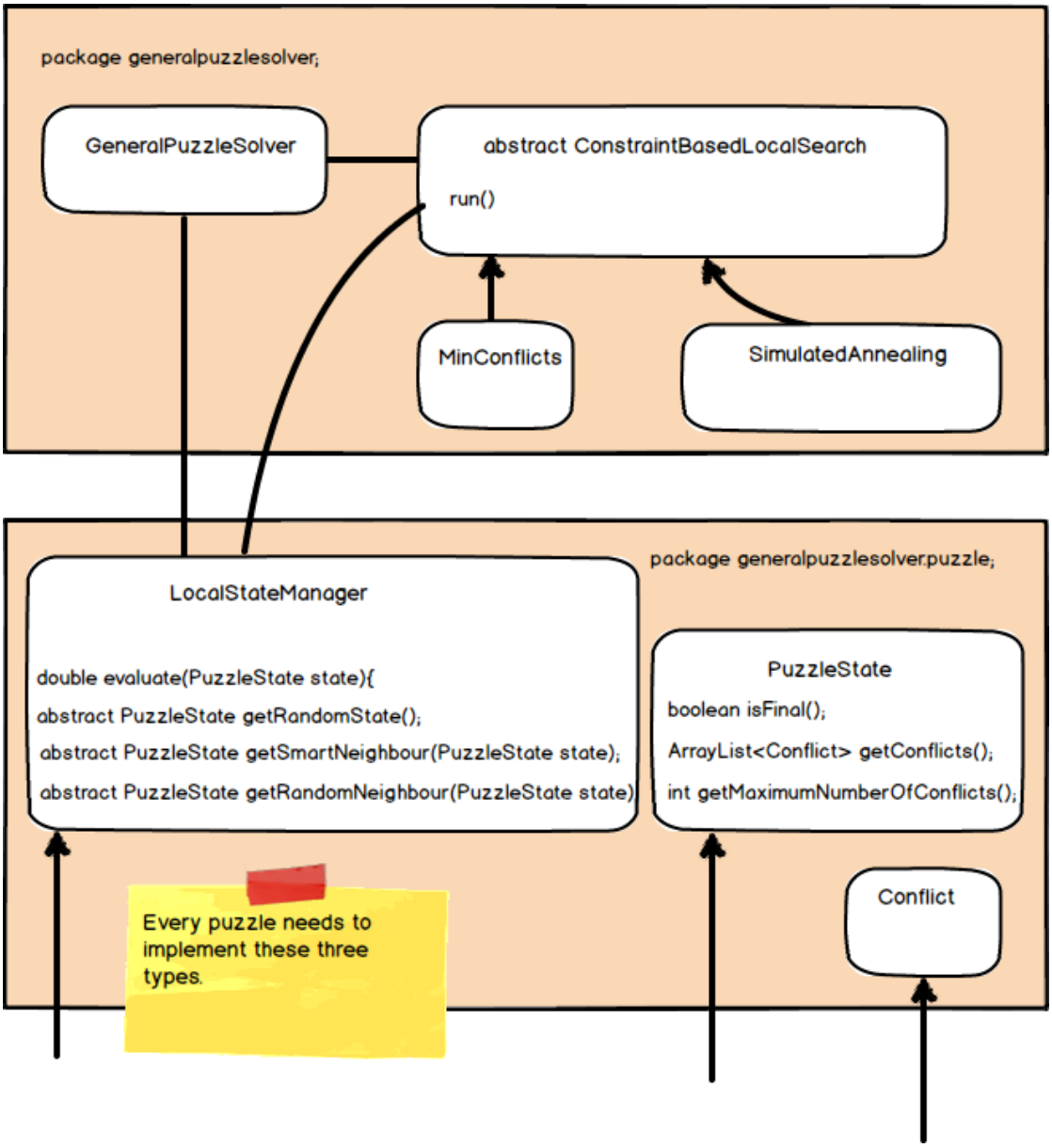

Figure 1: Architecture of our general puzzle solver.

To provide an example: In case of the graph colouring problem, it contains an adjacency matrix and a list with the data for all the vertices (colour and position).
The *PuzzleStateManager* will be a stateless object in most cases, that just takes *PuzzleState* objects into it's methods and changes them to some kind of ruleset provided inside the method.

2 First Puzzle: k-Queens

Combination	Evaluation Average	- SD	- Best	Steps Average	- SD	- Lowest
Easy / SA	0	18	2	6	4	5
Medium / SA	0	18	2	6	4	5
Hard / SA	0	18	2	6	4	5
Easy / MC	0	18	2	6	4	5
Medium / MC	0	18	2	6	4	5
Hard / SA	0	18	2	6	4	5

3 Third Puzzle: Sudoku