IT3105 - Artificial Intelligence Programming

Puzzle solver

Building a General Constraint-Based Puzzle Solver

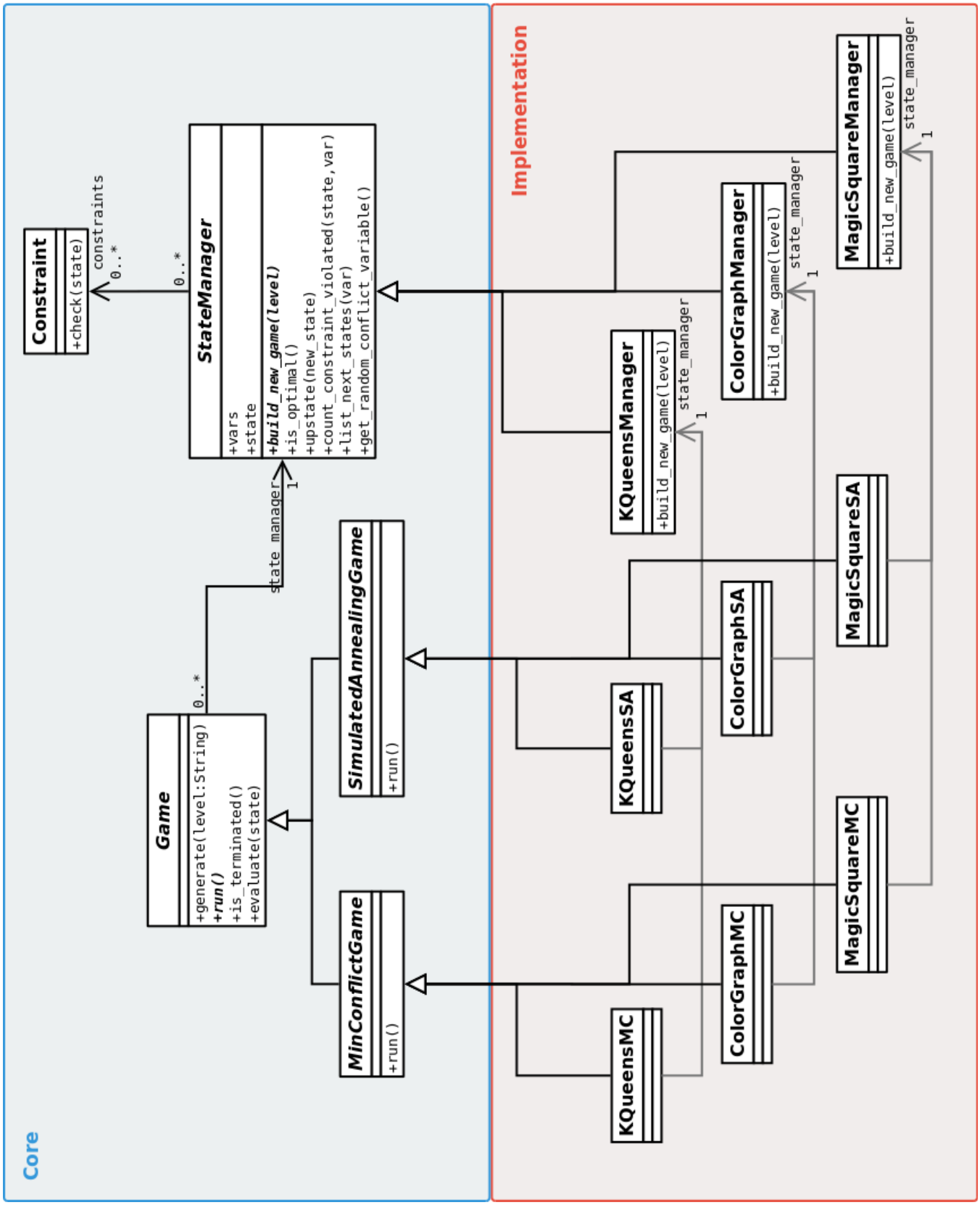
Marien Fressinaud
marienlf@stud.ntnu.no

24th October 2013

Table of Contents

1. Introduction to my Puzzle Solver

General class diagram



This class diagram doesn't include all classes nor all methods. It just describes the global architecture of my program and it's not so far from the one proposed in exercise subject. Main classes are presenting below.

Important classes and functions

StateManager

The state manager is the core of my program and should permit to manipulate all kind of game states. A game state is defined as following:

- A list of variables defined by an id and a domain (a list of possible values)
- A list of constraints
- A state which gives, for each variable, a value from its domain

`StateManager` provides also a set of helpful methods (I don't explain all of them here):

- `count_constraint_violated` counts the number of violated constraints for a given variable in a specific state. If no variable is given, it counts for all variables.
- `list_next_states` returns a list of states for a given variable. Basically, it generates a new state for each value from the variable's domain.
- `get_random_conflict_variable` returns a random variable which is involved into, at least, one conflict.
- `upstate` changes the current state by a given one.

Constraints

We can represent a constraint by a function which returns `True` if the constraint is not violated, `False` else.

My basic `Constraint` class is instanciated by giving a function with the following signature:

```
Boolean function(state, list_vars)
```

`list_vars` contains variables involved into the constraint and are given during instanciation.

There are also two more constraint classes (don't represent on the previous class diagram): `MustBeDifferentConstraint` and `SumEqualsConstraint`. They can be considered as generic and can be useful for several games.

Games

`Game` class is inherited by two other classes: `MinConflictGame` and `SimulatedAnnealingGame`. Here, the main method is `run()` and it executes the corresponding algorithm by using methods of the state manager. Algorithms are completly independant of games like I explain it in a following part, *Modularity*.

Evaluation function

I wrote a generic evaluation function in order to have not to redefine it in the child classes.

The idea is simple: I substract the number of violated constraints (NVC) of the given state to a constant (E_{max} = 1000). So, if there is no violated constraints, the evaluation will be maximal.

Because I wanted to return an evaluation between zero and E_{max}, and NVC can be greater than E_{max}, I had to do a rule of three: $\frac{NVC}{NC} = \frac{NVC'}{E_{max}}$ where

- NC = total number of constraints
- NVC' = scaled number of violated constraints

So finally we can calculate evaluation: $E = E_{max} - NVC'$

Finally, I force the minimal value to 1 because during the simulated annealing algorithm we have a division by this evaluation.

This function is pretty efficient in term of evaluation and I have good results with it. Unfortunately, it costs a lot of ressources (in specially time) since it must recalculate a lot of constraints. Moreover, the estimation of total number of constraints is not so perfect and I will discuss about it in the *Problems* part of *The Magic Square game* section.

Implementation part

For the implementation part, `build_new_game` of `StateManager` is the most important function since its the only one which must be redefined to create a new game. It's where we define variables and constraints for the game.

Modularity

During this exercise, I tried to keep my code the more modular as I could and I think I did something pretty good.

The previous class diagram shows which classes are important when we want create a new game. For example, when I have added the magic square game at the end, I had just to implement two new `Games` (min conflict and simulated annealing versions) and a new `StateManager`. Note that implementations of `Game` assign just the new state manager, no more because main algorithm is implemented by the parent.

So the most difficult part was to define variables, their domain and constraints (in `build_new_game` function): no more than 30 lines!

Finally, to add a new entry to the menu, I use a `Factory` class which contains a list of games:

```
__LIST_GAMES = {
    "mc": {
        "K-Queens": KQueensMC,
        "Graph Coloring": GraphColorMC,
        "Magic Square": MagicSquareMC
    },
    "sa": {
        "K-Queens": KQueensSA,
        "Graph Coloring": GraphColorSA,
        "Magic Square": MagicSquareSA
    }
}
```

The game must be added to “mc” and “sa” lists to be proposed during the selection but this code can still be improved!

To resume this part, to add a game to my Puzzle Solver, it's only ~100 lines of code (30 for the core) and two new items in a list. I could not do much better...

2. The Magic Square game

Overview

We can briefly describe magic square as following:

> *"A magic square is an arrangement of numbers (usually integers) in a square grid, where the numbers in each row, and in each column, and the numbers in the forward and backward main diagonals, all add up to the same number."*

Wikipedia - https://en.wikipedia.org/wiki/Magic_square

4	3	8
9	5	1
2	7	6

= 15

Illustration 1: A magic square of 3x3

A **normal** magic square of size N (with N > 2) contains numbers from 1 to N^2 (previous illustration is an exemple), so all the numbers must be differents.

We can calculate the number M (the "magic sum") for a normal magic square by a simple function:

$$M = \frac{N(N^2+1)}{2}$$

So we can say every square has a solution for N > 2.

Implementation

Representation

Magic square game is quite easy to represent:

- There are N^2 variables that are defined by the interval domain [1 ; N^2]
- For each row, column and diagonal, there is a constraint implying the sum must equal the value M. For that we have the `SumEqualsConstraint` class mentioned

above which takes just a list of variables (corresponding to one row for example)

- There is also a constraint implying each variable must be different from the others. `MustBeDifferentConstraint`, used also into the Graph Coloring game, is helpful here.

I have taken respectively N = 3, 8 and 24 for easy, medium and hard levels.

Problems

The previously representation is correct and can be resolved by both min conflict and simulated annealing algorithms... But it's far from perfect! Take an example with N = 3, where variables are defined by the domain [1 ; 9] and M = 15 for the continuation.

First, basic count of violated constraints was not efficient enough: granularity is bad because we can't evaluate at the same time row, column and diagonal sums. So I decided to modify the count with a coefficient (only used into `SumEqualsConstraint` class). This coefficient equals the difference between M and the current sum. For example if a row contains numbers 1, 5 and 6, sum equals 12 and the coefficient will be 15 − 12 = 3. So, here, I consider number of violated constraints is 3. Like I evaluate row, column and diagonal, I add these results and I can choose a value which is the best for most of the different sums.

Second (and third) problem comes unfortunately from my first solution. Indeed, `MustBeDifferentConstraint` is not "coefficiented", so the signification of this constraint is a bit diminished by the coefficient of `SumEqualsConstraint` and we can't satisfied this constraint as easily as before. But since this constraint is required only for a **normal** magic square, I just decided to resolve the global magic square problem and I deleted this constraint. It's a bit drastic, but I have let the corresponding lines in source code as comments to see how it's easy to add a constraint.

Finally, my last problem comes from the evaluation function and estimation of total number of constraints. Since I have introduced a coefficient, number of violated constraints can exceed the total... which is a bit strange. As I can't calculate a perfect total, I have just multiplied current total by 1000. It's an arbitrary constant but high enough to have a "working" total.

Evaluation function

As I said before, my evaluation function is a generic one and I have nothing specific to add for this game. See *Evaluation function* part of *Introduction to my Puzzle Solver* section for more information.

3. Results

By running my algorithms 20 times on each games' variant, I have obtained a list of interesting metrics (evaluation, number of steps and running time). Following parts give an overview of these results, except for the running time but I give a word about it in my observations. Each table is followed by the standard output of one iteration.

K-Queens results

		Evalutation			Steps		
		Average	Standard deviation	Best	Average	Standard deviation	Quickest
Min Conflict	Easy	1000	0	1000	41.65	42.69	6
	Medium	1000	0	1000	71.50	46.48	31
	Hard	1000	0	1000	727.85	41.17	674
Simulated Annealing	Easy	1000	0	1000	98.35	42.58	44
	Medium	999.98	0	999.99	*Out of steps!*		
	Hard	*Too long!*					

```
---------------------------------
   |   |   |   |   |   | Q |   |
   |   | Q |   |   |   |   |   |
 Q |   |   |   |   |   |   |   |
   |   |   |   | Q |   |   |   |
   |   |   |   |   |   | Q |   |
   |   |   | Q |   |   |   |   |
   |   |   |   |   |   | Q |   |
   |   |   |   |   |   |   | Q |
---------------------------------
---------------------------------
   |   |   |   | Q |   |   |   |
   |   | Q |   |   |   |   |   |
 Q |   |   |   |   |   |   |   |
   |   |   |   |   | Q |   |   |
   |   |   |   |   |   |   | Q |
   | Q |   |   |   |   |   |   |
   |   |   | Q |   |   |   |   |
   |   |   |   |   |   | Q |   |
---------------------------------
This is the end of K-Queens! (perfect game) eval = 1000.000000 ; steps = 72 ; time = 30ms
```

Graph-Coloring results

		Evalutation			Steps		
		Average	Standard deviation	Best	Average	Standard deviation	Quickest
Min Conflict	Easy	1000	0	1000	31.45	14.29	15
	Medium	1000	0	1000	109.25	70.54	52
	Hard	1000	0	1000	675.05	84.69	559
Simulated Annealing	Easy	1000	0	1000	101	58.66	50
	Medium	1000	0	1000	648.40	526.60	87
	Hard	999.98	0.01	999.99	*Out of steps!*		

To get an output for this game, I use the Graphviz library: I generate a .dot file by browsing the graph, by generating nodes (with specific color) and edges. Then, I can export .dot file as PNG image with the following command:

```
dot graph.dot -Tpng -o graph.png
```

Two files (starting and final graphs) are provided as attached files because they don't fit in this report. Note I don't use coordinates provided by original files.

Magic Square results

		Evalutation			Steps		
		Average	Standard deviation	Best	Average	Standard deviation	Quickest
Min Conflict	Easy	1000	0	1000	125.25	74.20	45
	Medium	1000	0	1000	2033.05	1415.58	411
	Hard	999.83	0.21	1000	8465.35	2702.04	2556
Simulated Annealing	Easy	1000	0	1000	100.20	58.66	31
	Medium	997.41	0.74	998.50	*Out of steps!*		
	Hard	977.03	3.06	981.62	*Out of steps!*		

Note hard MC version was often out of steps too but it seems to me it's interesting to see the gap between average and the quickest iteration.

```
SUM = 15
------------------
   8 |   7 |   8 |
   4 |   8 |   9 |
   9 |   1 |   6 |
------------------
SUM = 15
------------------
   6 |   7 |   2 |
   1 |   5 |   9 |
   8 |   3 |   4 |
------------------
This is the end of Magic Square! (perfect game) eval = 1000.000000 ; steps = 65 ; time = 8ms
```

Observations

A general observation we can make is about the simulated annealing algorithm. There are a lot of game stuck at an evaluation around 999.99 but never reach the optimal state. The reason seems quite simple. Indeed, there is a lot of empirical constants which are hard to define: max temperature, number of neighbors to generate, the function to decrease temperature and the evaluation function. I tried different values to try to keep a good ratio between performances and results, but I'm not totally satisfied because I don't explore enough the neighborhood.

Then, if we compare simulated annealing (SA) and min conflict (MC) algorithms, we can be tempted to say MC is the best. But if we look closer, we can see that's not totally true for magic square game and the easy level is resolved quicker with the SA algorithm. I explain this result by the fact MC can't evaluate globally the state of a game but just the case of one specific variable. Since it doesn't really matter for the two first games, it's critical in the case of magic square. So my conclusion is the algorithm must be chosen carefully. It's a shame my SA algorithm is not better to have more complete results.

Another interesting result comes from the standard deviation. It's always nearly zero in the case of evaluation and it's pretty simple to understand since both of algorithms try to approch the optimal state.

In the case of standard deviation of steps, it's a quite more interesting:

- It's constant for each variant of K-Queens (~42). So we can suppose any K-Queens game is equivalent to each other (no matter the first positionning of queens) and it will take almost the same amount of ressources to be resolved.
- It's also quite reasonnable for Graph Coloring, but not constant. We can assume the starting state is affecting speed of algorithm.
- It's really huge for magic square! So the first values in the square will totally impact the final result. Moreover, we can suppose MC algorithm doesn't fit perfectly to this problem.

My final observation is about running times. Generally SA algorithm is longer than MC one. It's normal since SA generates several neighbors and evaluates each of them. But in the case of magic square, I noticed that was the contrary for medium and hard versions (~4 minutes MC against ~60 seconds SA). This result is a bit strange and I have some difficulties to find a good reason to that.

Also, MC algorithm can take a really long time before find a solution: I needed to wait 30 minutes for each iteration of hard K-Queens! But I think to know how to improve it: for the moment, I recalculate the number of violated constraints at each iteration whereas I move just one queen. By keeping a sort of cache, I can memorize violated constraints and just recalculate the impacted ones. Unfortunately, I'm out of time to do this improvement!