10/11/2017

Since I’m going to change the container for variables, should I make a separate class? For now, no, as I could not read the variables with a for properly. Actually, I will make a separate class and make the container public, since the for iteration is the same for sets and arrays.

!!! ArrayList and Set both use add and remove for access, so I actually don’t need to implement a separate container class.

I’m just going to use the Collection interface.

I’ll put all the read/write functionality in separate static functions so I can reuse the code if needed.

**QUESTION:** There is a serious question of whether to let constraints which refer to values that cannot be taken for the current assignment be or remove them for now and add them back on backtracking. !!! (For Variable Ordering based on Least/Most Constrained and Value Selection, it matters whether I have useless constraints or not for time consumption)

**QUESTION:** I think I’d rather model the Constraint in code as a class which holds a pair of values (var1, var2) and a pair of values they cannot take (val1, val2). This way I can index the constraints for a variable var1 based on the value val1 corresponding to it. At the moment I am maintaining constraints as (var1, var2) and an array of the pairs of values they cannot take. Other than the memory consumption increase, is there anything else you think I may need to take into account?

For faster constraint access based on value (which we will need for value selection), I have stored the values in a map indexed by the variable value they concern.

I will progressively remove values from domains such that assigning a value for a variable does never violate a constraint for current assignment.

13/11/2017

Due to a more efficient implementation of ConsistentAssignmentValueSelection, values in domains must now be nonnegative. (N/A actually)

Input format does not accept more than 1 constraint targeted at the same variables. Will be solved in input parsing.

23/11/2017

FileHandling does input into argument instead of return because, when reading the problem, we must read both variables and constraints, so we have 2 arraylists. It would be weird to output one as argument and the other one as return. (When reading just the assignment we leave it as argument to be consistent with the previous case)

At the moment, when generating CSP problems, the order of the variables is locked in the file because they are not named (they must be in order of index 0,1,2,3,…). I do have the facility of providing names for them as indexes (so they can have IDs). Should I integrate it? I don’t see any reason why not, although the project is working like this as well.

FileHandling also handles reading the NQueens problem solution into memory, but at that point we need to use int[][] as a return value, because we don’t know the size of the matrix before reading the solution. Passing a int[][] as a matrix and reassigning it inside the function does not reassign it outside the function.

Laptop died on this date due to rain in backpack. Did backup data on github in the morning, so loss was only one days worth. Recovered laptop on 25, with no loss in the end.

11/12/2017

There are multiple ways to generate Maps such as the DrunkWalk or the ForestGrow. At the moment, I have decided to use ForestGrow. The input is the number of zones desired (name it n), as well as an optional size of starting zone (name it w). The algorithm finds the least perfect square greater than n, name it k^2 and creates a map of k\*w lines by k\*w columns, which is split into k^2 zones of w lines \* w columns. Then, the n zones are seeded from n of the k^2 zones and grow outward until there are no empty spaces and their sizes are about k\*k\*w\*w/n.

I’m also modifying the format of the input and the output. From now on, we’ll also add the index of the variable (its “name” basically) when we refer to it (in input: before enumerating number of values and values for a variable; in assignment: before each value we’ll put the index of the variable it refers to).

Assignments also have to specify the number of variables which they are assigning.

12/12/17

There’s a lot of discussion with how the map is generated, ranging from which algorithm we’re using (RandomWalk, ForestGrow), to how the algorithm is implemented (when should a Forest stop growing? Is it allowed to “eat” from other forests?)

21/12/17

The FileHandler class is getting a bit too big as it has the file handling for the CSPs, the MapColoring and the NQueens. At the moment, I have the following packages: Auxiliary, LookBack, MapColoring, nQueens, Solver, Validator, ValueSelection, VariableOrdering. I intend to reorganize them to balance the weight of the code better.

I’ve chosen to just use Collection<Variable> and Collection<Constraint> in favour of doing separate classes for the set of variables and constraints as I want to be able to easily iterate through them. Either way, what I would need to do as a separate class is a subset of what Collection can do, so it seems appropriate to just use Collection here.

Each Variable should have its own domain, i.e. do not use the same ArrayList in more than one domain.

There should not be any duplicate constraints (i.e. if two constraints have the same variable in the same order and have the same pairInts restrictions, then they should be the same object). (There are some optimisations which can be done with this, none in place atm)

PairInts parameters first,second are final as a pairInts has no reason to be modified. This means you can use shallow copies of PairInts without worrying that modifying one would affect the others (since you cannot modify any). (The problem appeared in MapColoring, where the constraints at the borders are always the same, i.e. not the same color)

Generally speaking, I’ve went with representing constraints as the pairs of values which CANNOT be taken, as they were less in number in the case of nQueens and MapColoring.

When calling a …toCSP, the Collections of Variables and Constraints must be initialized before the call. This was done like this so that you can decide whether you want to use an ArrayList, a HashSet, etc. without changing how the method works.

27/12/2017

MapColouring can be tweaked to do graph colouring, though I think it might be faster just to build graph colouring the usual way (i.e. by translating and using Solver).

Ended up having to create a MatrixPosition class for the MapChecker (Since I want to check the same positions in 2 different matrices). Thought about whether to make IntMatrixElement (used at MapGeneration and MapToCSP) be a MatrixPosition or contain a MatrixPosition (as I want to be able to do left,right,up,down on both IntMatrixElement and MatrixPosition). Ended up deciding to make the IntMatrixElement have a MatrixPosition since, the way I see it, elements in a matrix HAVE-A position rather than ARE positions themselves (it seemed more intuitive this way, in case somebody else might have to understand those).

Just to be on the clear side, I have made the mRow, mCol of the MatrixPosition and the mMatrix of the IntMatrixElement ***final***. In how I’ve used them, I never change their values after the initial assignment. This is, again, to help understanding (and protect myself should I accidentally attempt to misuse them).

Also, in MapToCSP, I found out that clone() is undefined for Collection<PairInts> but defined for ArrayList<PairInts> (I’m pretty sure you can sub out PairInts for any user-defined class).

30/12/2017

For Most/Least constrained variables, the tie-breaker would usually be the most constraints with remaining variables. However, I am not keeping the constraints up-to-date:

* By not keeping constraints up-to-date, I am accessing the constraints of the variables in order (so O(nrcon) for going over the nrcon constraints of a variable). Thus, complexity is O(nrvar\*nrcon), where nrvar is the number of variables I went over.
* By keeping constraints up-to-date, I would be accessing the constraints of the variables sometimes in order, sometimes by jumping around to one constraint in O(log nrcon), which would lead to a complexity of more than O(nrvar\*nrcon). Thus, it is more efficient to not keep the constraints up-to-date, at least as of now.

Note: This may change as I implement more algorithms. I will then make a way to turn on or off the updated keeping track of constraints. The updating will probably happen in the buildFrame of the Solver.

**TODO:** Put a checker inside solve. If we get to a variable with no values, it is safe to LookBack, since that variable will never work out.

LeastConstrainedVariableOrdering is the same as MostConstrainedVariableOrdering, but with the result flipped. Thus, I think it is safe to say that LeastConstrainedVariableOrdering extends MostConstrainedVariableOrdering (it is more natural in code that way as well, since the classes aren’t static).

12/01/2018: ORDERING IMPLEMENTATION HAVE CHANGED! So most of what is here is outdated.

31/12/2017

Since I only run the LookBack algorithm when I hit a dead-end, I will implement a lazy version of Gaschinig’s, which does the processing it should do while searching only at leaf dead-ends. The VariablesRestrictions class will lend itself to this.

Note: I will look only for the dead-end variable’s culprit. This is not necessarily better than keeping track of everything. By looking only at the variable’s culprits, it results in O(nr\_var \* log nr\_var) complexity in the worst case per each Gaschnigs call. Keeping track of everything would result in O(nr\_var \* total\_stack\_frames\_encountered). So, if we have less than total\_stack\_frames\_encountered / (log nr\_var) calls of Gaschnigs, the variation I’m proposing is better. Note that total\_stack\_frames\_encountered = total nodes in search tree.

Also, to have access to the Backtrack in Gaschnig’s for when we’re dealing with an internal dead-end, I’ve made GaschnigsBackjumping extend Backtrack

**TODO:** I’ve made an auxiliary class for the timer. I also need to make an auxiliary debug class so I can keep track of outputs and stuff.

09/01/2018

Discovered I was doing Variable Ordering at each step instead of only once at the beginning, which is wrong (if a Variable cannot get a value at a step, it won’t be able to get a value at any other further step from there).

11/01/2018

Apparently, I was removing values from domains of assigned variables. This is not necessary. Upon removing this inefficiency, I lost the differences I had between using different Backtrack techniques on a set of 60 map coloring CSPs. I have saved those separately, to make the difference visible, and I have pushed the project on github.

Fixed, it was a bug. You can see the difference with/without the optimisation if you run it until instance 40 on nQueens.

Also, **TODO:** Reimplement the Dynamic Variable Ordering. It does not need to return an ArrayList of Variables, just the next Variable to assign a value to. When you corrected it last time, it was changed to Static Variable Ordering, but the Rina Dechter book calls them Dynamic Variable Ordering.

12/01/2018

I think Partial Look-Ahead is supposed to be incompatible with Dynamic Variable Ordering, since it assumes some kind of order to the variables to assign.

I will allow the solver to switch between Static Variable Ordering and Dynamic Variable Ordering with a flag. In case Dynamic Variable Ordering is used with Partial Look-Ahead, the order in which the remaining variables are processed will be the order they were initially passed in.

I have decided to pass the Solver to each sub-algorithm (of Var Ordering, Val Selection and Backtrack), to ensure that the sub-algorithms have all information they would ever need should any new sub-algorithm be implemented in the future.

This is also because the Partial/Full Look-Ahead and the Arc-Consistency required access to all the variables remaining to be assigned, while Forward Checking didn’t. This means that the algorithms may require access to some info while others wouldn’t, so if an algorithm needs access to something, it should go get it itself rather than relying on the Solver to supply it.

For LookBack methods, I’m also passing the mSteps of the mSolver even though I could not so that I make it clear that LookBack methods modify mSteps.

I opted to pass the Solver as an argument instead of making the Solver static to allow for possible concurrency in the future. If the solver is static, one cannot make multiple instances of solving problems at the same time from what I know in Java.

Since there was a potential time improvement by keeping track of unassigned variables, the Solver now also has an unassigned variables ArrayList mVarsLeft and an assigned variables ArrayList mVarsAssigned that all sub-algorithms can access. In case of static variable ordering, the mVarsLeft is ordered in the reverse order the variables will be assigned and the mVarsAssigned is in the order the variables are assigned, thus the Partial Look-Ahead can work the way it is described in the Rina Dechter book. Note that I’ve chosen reverse order for mVarsLeft so that insertion and removal can work in O(1).

In order to permit fast access to constraints based on the 2 variables they refer to, I have made the mConstraints member of the Variable class be a Map<Variable, Collection<Constraint> > instead of just Collection<Constraint>. I have thought about collapsing the whole Collection<Constraint> into a single constraint, but I feel that fundamentally, we can have two different constraints referring to the same 2 variables (such as: they have different values AND they cannot have both even values).

Also, you can have an inverted constraint: i.e. var1 and var2 cannot have (1,2) AND var2 and var1 cannot have (1,2)