CSP-solver, Assignment 2 IT3105

Anders Sildnes, Andrej Leitner *students*

***Abstract*—This text answers assignment 2: combining CSP with best first search. The assignment builds from the A* implemented in the previous assignment. In this document, we explain our implementation of CSP modelled as a search problem.**

MODELING problems as CSP has multiple benefits. The goal is to assign each object a value from a finite domain, without violating a set of constraints. Each node in our graph corresponds to a temporary assignment to a subset of objects $o \subset O$. If o is complete, that is, every object has an assigned value, the problem is considered solved.

I. Initialising the VENT

The Constraint Network is modelled as a class object called *NET*. It initialises a set of variable instances (*VI*), each with a complete domain. The domains are modelled as 1D standard lists with integers. This can e.g. correspond to indexes in a list of colors (as used in this assignment), such that the domain denotes indexes in a (global) list of colors.

The CNET is also responsible for reading (canonical) constraints. First, it takes in a line of input. The lines of input are currently just two numbers, although the code should support any number > 2. Each number (integer) maps to one variable in the domain.

First a general function is created. Currently, it is always assumed to be a non-equality constraint (to modify this, the input line would need to be checked for an operator and find a corresponding mapping, which should be a trivial task). The non-equality constraint does not consider domains or assignments, it will have general form $x! = y$, for any x and y to be specified later. Here, it is important to map each symbol to the index of a VI in a state. To do this, only the index of the VI if stored, and we can lookup the domain from state during invocation. Listing I shows a shorter version of creating a constraint:

```
sym_to_variable  = dict()
variables = line.split()
symvars = symbols(...variables)
lambdafunc =lambdify(symvars, Ne(*symvars))
for symv,v in zip(symvars, variables):
    sym_to_variable[symv] = int(v)
c = Constraint(lambdafunc, (symsvars,
    variables), sym_to_variable, CNET)
```

Listing 1: "Creation of constraint"

sym_to_variable maps each variable to its position in the constraint. E.g. for $x < y$, the position of the variables matters, so the dictionary maps a symbol to a variable.

c , the *CI* is created once. In the CNET, it is then passed on to a list of constraints. This list is indexed by variable; such that each variable has a list of constraints it occurs in. This means that a constraint with two variables will occur twice in the list of constraints. This is to make lookups easier.

The way the constraint reader works now, it should be easy to extend to more advanced constraints. Apart from the mapping from type of constraint to code, the rest of the code is written using list comprehensions and Cartesian product mappings which should be able to test constraints for any number of variables.

II. Iterating to a valid solution

As in the previous assignment, our A-GAC* implementation works with *state* and *problems*. The initial state is a copy from CNET, i.e. a complete copy of the CNET's domains and

variables. This means that the initial state is independent to the memory address of the CNET.

From the initial state there is an assumption: it takes a random object o, and reduces the domain to the singleton set. Then, from the CNET, it retrieves all the constraints associated with o. Each constraint is then checked using arc-consistency algorithms. The AC-method queses constraint-checks for all variables affected by the assumption. If at any point a constraint forces a domain to become empty, "False" is returned and the assumption is left alone. The revision is done as per Listing 2:

```
self.addState(state)
can_satisfy = False
arg_list = ... # ordered list of possible
    domain value in the constraint
for tup in product(*arg_list):
    if self.function(*tup):
        can_satisfy = True
        break
return can_satisfy
```

Listing 2: "Checking constraint"

Each tup is a possible assignment of domain variables. At any point, if the constraint can be satisfied, we return "True"

III. Generality of A*

The provided source code provides both assignment 1 and assignment 2 as inputs to the same file and code with A*. The only difference is that the CSP and navigation have different subclasses of *Problem* and *State*. Here the *state* adds new states by using the method described in the previous paragraph, and assesses value of states by using the heuristic mentioned in the end of this text.

In other words, to implement the CSP you do need to know anything about A*, you just need to define a state, a success-full termination and how to progress from a state.

IV. Generality of A*-GAC

The code we have provided gives *csp* as its own file. It is implemented so that it has its own instances of an A*-state, each with their own *VIs* and *CIs*. The "Problem" class is not implemented; this is left as problem specific (reminder: the Problem class defines successor states and neighbour generation).

To solve a problem in A*-GAC, you implement your own subclass of *Problem*. From the CNET you get the initial state. We have set the generation of neighbours to be specific for each problem. This is because some problems have different properties. For example, we felt that propagating from two different random points in the graph was not beneficial.

V. Generality of CNET

The root nodes startes by making a deep copy from the original CNET. Since then, new states are generated by creating a copy from its successor state. Hence, the CNET is separated from each VI and CI.

VI. Heuristic

For the A* algorithm, an index value to each node in the search tree is given. This is determined by $f(x) = g(x) + h(x)$, where $h(x)$ is a objective value ...

Our current heuristic is to count the size of all domains to each variable. The size is taken minus one, such that assigned objects (that has a domain of size 1) does not count in the value.

Anders Sildnes, 4.års student, master informatikk, NTNU.

Andrej Leitner, 4.års student, master informatikk, NTNU.