CSP-solver, Assignment 2 IT3105

Anders Sildnes, Andrej Leitner *students*

***Abstract*—This text answers assignment 2 this is some more text that I can probably fill in somehow. Hello my name is alfred.**

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. In this document, we explain our implementation of CSP modeled as a search problem. Each node in our graph corresponds to a temporary assigment to a subset of objects $o \subset O$. If o is complete, that is, every object has an assigned value, the problem is considered sjolved.

I. Initializing the CNET

The Constraint Network is modeled as a class object called *CNET*. It initializes a set of variable instances (*VI*), each with a domain a complete domain. Each domain is currently assumed to be a list of integers. This can e.g. correspond to indexes in a list of colors (as used in this assignment), such that the domain denotes what colors of all availible colors are availible.

The CNET is also responsible for reading (canonical) constraints. First, it takes in a line of input. These are currently in the form of integers, where each integer maps to one variable in the domain. If then creates a function. Currently it is assumed that non-equality is desired; to extend the source code you would need to read an operator in the constraint and map it to the type of function.

The function is created using sympy. To create a function, you first create symbols. The symbols represent objects that can take on any value. You then use symbols to create functions. In this assignment, all functions take form $x! = y$. The current procedure is shown in Listing I.

```
1 sym_to_variable  = dict()
2 variables = line.split()
symvars = symbols(...variables)
4 lambdafunc =lambdify(symvars, Ne(*symvars))
for symv,v in zip(symvars, variables):
6     sym_to_variable[symv] = int(v)
c = Constraint(lambdafunc, (symsvars,
    variables), D, 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 dicitionary will later assist in evaluation the constraint. This is mapped in the for loop

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. You would have to read in the type of function from *line* (from Listing I), and map it in the call to *lambdify*. The rest should more or less work itself.

II. Iterating from CNET to a valid solution

As in the previous assignment, our A* implementation works with *state* and *problems*. From the CNET, you generate an initial state, which is a complete copy of the CNET's domains and variables. This means that the initial state is independent of the CNET.

The initial state makes 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 as in Listing II.

Before entering this function, it temporarily reducing another variable, $v! = o$, in the constraint, to a singleton set. In the case of binary constrains, this means that it will compare the singleton set of o and its opposing object. If the constraint is not satisfied, can_revise is False, and the value from the domain of v is taken away.

```
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"

In Listing II, each *tup* is one possible assignment of all domain variables. As soon as one possible assignment is valid, it is returned that the current assignment to all objects is valid. In the case

III. Generality of A*

The provided source code provides both assignment 1 and assignment 2 as inputs to the same 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.

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*.

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 benefitial.

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. Still

Each VI and CI has their own pointer to its original CNET.

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 heuristical 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. This heuristic gu

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

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