IT3105 Module 2

A-GAC, A General Constraint-Satisfaction Problem Solver*

Generality of A* algorithm: any aspect geared specifically towards CSP and GAC should be in subclasses.

The generality of the astar algorithm has been discussed in the previous module, so it needs not be discussed more. However the subclassing that will allow us to combine the two problems, will be.

The methods that have been overridden from **astar.py** are "generate_initial_searchstate", that calls on a method in gac that reads in the board from specifications and then creates the objects instances needed to perform the algorithm. Then we have **generate_successor**s, which call for the subclassed searchstate object to generate its legal neighbours. **Arc_cost** returns one unless the domain is empty. **appendtoopen**, **popfromopen** and **generate_openlist** have been overridden to use a heap instead of having the possibility for bfs and dfs.

The methods overriden i **state.py** are "**check_if_contradictory**" that returns true if the domain if any variable in the domain is 0. "**check_if_goalstate"** returns true if all the domains have a length of one. "**calculate_neighbours"** goes through the domains of the state and picks the smallest domain that still needs reducing and creates a **searchstate** with a **deepcopy** of its own domains. It then runs the **rerun** method from gac on them. If they still are legal states they are added to its list of **neighbours** and returned.

Generality of A*-GAC algorithm: any aspect geared specifically towards vertex coloring should be in subclasses.

The functions **revise**, **domain_filtering** and **rerun** have all been made with for-loops so they. The constraint object contains all specific constraint logic and will be discussed further down this document. They have a method **get_other** that takes in a variable and returns a list of all the other involved variables. This way revise can iterate over all possible pairs of variables in a constraint. The same goes for the functions needed to add the specific variable and constraint tuples to the revise queue.
Because we need to use it with the gui and update the gui for each astar step we had to implement it in a specific gui class, but the general algorithm would look like this.

```
def astar_gac(self):
    csp = GAC()
    astar =Astar (csp.domains)
    csp.initialize_queue()
    csp.domainfilter()
    while len(astar.openlist)> 0:
        astar.do_one_step()
```

Verification of clean separation between Constraint Network and the VIs and CIs of the search states.

To implement this we made a class named ConstraintNet that contains all constraints in a dictionary with the hash of a variable as its key. This way all we have to copy when we generate new states are the variable domains. The deepcopy operation is one that has to be performed a lot of times, so any unnecessary data would slow up the algorithm further so we decided not to explicitly pass the constraint instances as a parameter when new states when new states were generated. Instead we chose to do it implicitly by letting the state use a hash of its domains as a key in the dictionary .

The variable domains are represented also as a dictionary with the hash of a variable as a key. This, does change however so we have to deepcopy the dictionary each time we generate a new state. This allows for the separation of the VIs and CIs.

Dynamic code chunks:

To create dynamic code we used the makefunc method suggested in description of the problem. We then pass the expression to the constraint when we initiate it.

```
def __init__(self, vertices,expr):
    self.vertices = vertices
    self.expr = expr
    self.function = self.makefunc(["x","y"], self.expr)
```

Now the **revise** function can call this function with a pair of values to see if they satisfy the constraint.

Domain_filtering
This method pops element from the queue and revises them until the queue is empty. If revise returns True (meaning that a domain was reduced) it calls for a support method that adds all constraint/variable pairs that are not the ones that were just revised.

Revise:

The revise function has a boolean that it sets to True if it reduces the domain of a variable, and then returns it.

```
def revise(self, searchstate, statevariable, focal_constraint):
    revised = False
    for value in searchstate.domains[statevariable]:
        satisfies_constraint = False
        for other_variable in focal_constraint.vertices:
            if other_variable != statevariable:
                for some_value in searchstate.domains[other_variable]:
                    if focal_constraint.function(value, some_value):
                        satisfies_constraint = True
                        break
                if not satisfies_constraint:
                    searchstate.domains[statevariable].remove(value)
                    revised = True
    return revised
```

For each possible value in the domains of the focal variable it checks to see if there is a variable in the other domain that can satisfy their shared constraint. If no variable in the other domain can satisfy it, it is removed from the domain of the variable.

Each time the revise function returns true, all constraints and variables affected by the domain reduction (except the pair we just revised) are added to the queue again.