Module 2
IT3105

Simon Borøy-Johnsen
MTDT

October 5, 2015

1 A*

The general A* and node classes used in module one is also used here. A problem specific node class, as described in module one, has to be implemented.

2 A*-GAC

The A* is as general as in module one. The four functions in the node class has to be implemented.

The CSP class is completely general. To run the CSP, one only needs to generate the variable domain dictionary and the list of constraint instances.

The A*-GAC takes in a variable domain dictionary and a list of constraint instances in order to initialize a CSP class, and a general A* class, along with a problem specific node class, in order to make guesses if no solution is found. No problem specific sub classing is needed.

3 Constraint network

The state of the nodes in the A* are whole CSP objects. To separate the constraint network from the vertex instances, a deep copy of the variable dictionary is made every time a new node is being expanded. Because the constraints are only read, not written (at least in this course), there is no need to make a copy of them.

Neighbours in the A* algorithm are generated this way;

```
csp = CSP(deepcopy(variables), constraints)

if csp.rerun(next_variable, value) is not None:
    neighbours.append(Node(self, csp))
```

A copy of the variables is made, before the rerun function is executed. This (hopefully) reduces the domains of some variables. The CSP, with reduced variables, is then added as the state of the new neighbour node.

4 Explanation

a) To create the code chunks, I basically copied the method supplied in the lecture notes. The *generate_function* function takes in a list of variable names, and an logical expression containing all the variable names. A function that evaluates the expression using *eval* and *lambda* is returned.

```
@staticmethod
def generate_function(variables, expression,
    environment=globals()):

    return eval("(lambda␣%s:␣%s)" % (",".join(variables),
        expression), environment)
```

b) In the vertex colouring problem, the only thing that matters is that neighbouring vertices don't have the same colour. To interpret this into canonical form is simply making sure the values of two adjacent vertices are unequal.

```
@staticmethod
for edge in self.edges:
    variables = [node for node in edge]
    expression = '%s!=%s' % (edge[0], edge[1])
```

c) This contradicts point b). I haven't avoided using "both code chunks and a hand-built interpreter".