IT3105 Module 3

Combining Best-First Search and Constraint-Satisfaction to Solve Nonograms

Representation of variables, domains and constraints

We have decided to follow chapter 2.2 Using an Aggregate Representation described in the module 3 description to represent the variables, domains and constraints in this project. We first started using segments and their index as variables, but found out that this caused problems later on.

The variables are represented as entire rows or columns, where a Variable object has three parameters – **index, type and size**, where index is row or column number, type is simply a string stating whether this is a *row* or a *column*, and size is the length of the row or column.

The domain for a variable is a list of all the possible permutations of segment placements this variable has, given a number of segment sizes and the row or column length. A permutation is represented as a list of Booleans, where each index is either True or False based on whether this permutation allows a segment at the appropriate index. We first calculate the initial possible ranges of a segment in a variable using the following formula:

$$r_{1s} = 0,$$

$$r_{js} = \sum_{i=1}^{j-1} (LB_i + 1), \quad \forall j = 2, \ldots, k$$

$$r_{je} = (n-1) - \sum_{i=j+1}^{k} (LB_i + 1), \quad \forall j = 1, \ldots, k-1$$

$$r_{ke} = n - 1$$

n = size of row or column, k = number of segments, LBi = length of segment i
Function output: [[0, 1, 2, 3], [4, 5, 6]]

Next, we calculate all the permutations for this row or column using the following function:

```
def calculate_permutations(self, segment_domains, segments):
    permutations = list(itertools.product(*segment_domains))
    for list_element in copy.deepcopy(permutations):
        for i in range(len(list_element)-1):
            if isinstance(list_element, tuple):
                if not list_element[i] + segments[i]  <
list_element[i+1]:
                    if list_element in permutations:
                        permutations.remove(list_element)
                        break
    return permutations
```

We now have everything we need to create Boolean lists of all possible permutations for a variable.

The constraints are generated by iterating through every variable corresponding to each column for every variable corresponding to each row, and then assigning it the expression *x == y.* A constraint is represented as its own object, Constraint, which holds a list of variables that are involved in this constraint, and a string expression on the form "x==y". Constraints therefore exist as a relationship between rows and columns. We can filter out domains for a row by checking if it has a True value for an index that does not have a True value in a specified column.

Heuristics

When generating successors for a given state, we choose the variable with the smallest domain that is bigger than 1. When calculating the heuristic for A*'s traditional h function, we decided to use the rather simple heuristic of adding the length of all domains, minus one.

Subclasses and methods needed to specialize the general-purpose A*-GAC system to handle nonograms

To represent the current state, we have created a new state object called NoNoState which inherits CSPState (which, in turn, inherits the superclass State). The *calculate_neighbours* function is overridden to handle *banned keys*, and the only other difference is that we check if the smallest domain key is *None* or not.

We also have a class called mod3GAC which inherits from GAC, and we had to slightly modify the *revise* function in order for the system to handle nonograms.

The difficulty and challenge is this last module was really how to represent the variables, and especially how we could calculate all the possible permutations of each row or column. We spent quite some time reading papers on different approaches as to how we could calculate all the legal permutations of a row, but we finally figured it out as described earlier in this document.