University of Leeds – School of Computing

COMP2611 **COURSEWORK 1** (10% of module)

Coursework issued:	Thursday 13th February 2020
Submission deadline:	Thursday 5th March 2020

To be submitted via Minerva (the VLE)

General Overview

In this coursework you will produce a representation of a problem that can be used with the implementation of search algorithms that you have been experimenting with in the lab sessions.

The Problem

We have seen the 8 queens problem, which asks for an arrangement of 8 queens on an 8×8 chess board so that no two of them can take each other. That is, no two of the queens can be in the same column, or in the same row, or in the same 45 degree diagonal line. We have also mentioned the problem of placing n queens on an $n \times n$ board.

In this coursework the problem is given an $m \times n$ board, where m and n are any integers greater than zero, to find the smallest number of queens that cover all the squares on the board. That is, to find the minimum number of queens which can be placed in such a way that every square on the board either has a queen on it, or is in the same column, or in the same row, or in the same 45 degree diagonal line (in any direction) as at least one queen. Unlike the 8 queens problem, there is no requirement about queens threatening each other.

Your solution must use the implementation of search provided in the files `queue_search.py` and `tree.py` as seen in the lab exercises.

File submission

Submission is be via Minerva (the VLE). Your code will be tested using the version of Python2 on the Linux computers in the DEC10 Lab. If you develop

it under a different version, check that this does not cause problems. In one compressed directory, submit the following:

- <student_ID>.pdf
  What your report must contain is explained in the next section

- `tester_outcome.txt`
  The file called `tester_outcome.txt` will be automatically generated when you run `qc_tester.py`

- `queen_cover.py`
  The file called `queen_cover.py` containing the problem representation. You can base the overall structure on the examples of the knights tour, the eight puzzle, and the lab exercises which you have already studied. It is essential that the representation works with the code in `queue_search.py` and `tree.py` without making any changes at all to these files. Do not submit these two files. Your solution will be tested using the versions of these files provided on the VLE.

- `qc_tester.py`
  The file called `qc_tester.py` which should be either the file provided (if all the tests work with your code in reasonable time) or the provided file with some of the tests commented out (if your code does not work with these). If the search algorithm is reporting more than 60 seconds that would count as unreasonable for that test; you should expect very much less for many of them. Your coursework will not be marked on how fast the tests run provided it is not unreasonable.

Marking scheme (total available: 100)

To get full marks there should be a few appropriate comments in the code but not excessively long. Un-commented code will not be marked.

- Shown in the tester_outcome.txt file:
  - Code runs and works on all examples in the test file, producing a correct solution in each case. (20)
  - Problem info prints out number of rows and columns on the board (10)

- Shown in your <student_ID>.pdf report:
  - **State representation:** in one or two sentences, describe your state representation (10)
  - **Possible actions:** where do you look for possible actions on the board? Attach a small screenshot of your $qc\_possible\_actions(state)$ function to the report (25)
  - **Successor state function:** Attach a small screenshot of your $qc\_successor\_state(action, state)$ function to the report (25)
  - **Test for goal state:** in one or two sentences, explain how do you check if the goal state is reached. Attach a small screenshot of your $qc\_test\_goal\_state(state)$ function to the report (10)

Test File

Note that this runs uniform cost search by providing a constantly zero heuristic to A* search and the cost function used is the length of the path in the search tree.

```
import sys
from tree          import *
from queue_search  import *
from queen_cover  import *

def zero_heuristic(state):
    return 0

search(make_qc_problem(1,1), ('A_star', zero_heuristic), 5000, [])
search(make_qc_problem(3,3), ('A_star', zero_heuristic), 5000, [])
search(make_qc_problem(4,4), ('A_star', zero_heuristic), 5000, [])
search(make_qc_problem(5,5), ('A_star', zero_heuristic), 5000, [])

search(make_qc_problem(5,6), ('A_star', zero_heuristic), 5000, [])
search(make_qc_problem(6,5), ('A_star', zero_heuristic), 5000, [])
search(make_qc_problem(10,3), ('A_star', zero_heuristic), 5000, [])

search(make_qc_problem(3,4), ('A_star', zero_heuristic), 5000, [])
search(make_qc_problem(4,7), ('A_star', zero_heuristic), 5000, [])
```

```
search(make_qc_problem(2,50), ('A_star', zero_heuristic), 5000, [])
```