COMP2611 COURSEWORK 1 REPORT

Student ID: 201255350

Introduction

The objective of this report is to show the implementation and results of a chess domination puzzle using search algorithms. A domination problem is a type of chess puzzle in which the minimum number of pieces for a specific piece must be found so that they cover (attack or occupy) the whole chessboard.

In this case the program must find the minimum number of queens necessary to cover all of the squares of an m * n chessboard, as well as displaying the way in which these queens should be laid out in the board.

An A* search strategy will be used to test the functions. As per the coursework specification a zero-heuristic will be used along this strategy, however a simple heuristic has been implemented in the code that allows the algorithm to run significantly faster (in queen_cover.py → empty_squares_heuristic(state)).

State Representation

The chosen state representation is formed by four variables. The first one stores the number of squares controlled by the current state, the second one stores a list with the controlled squares' coordinates, the third stores the position of the current queen, and the fourth one stores the current board layout.

Possible Actions

The possible actions for a given state are calculated by counting all squares with value "0" in the board (state[3]), and returning a list containing their coordinates. Every time a queen is placed in a square, the value on that square is turned to "1", so it won't be counted.

```
# Goes through the array and counts all squares with a 0 on them
def qc_possible_actions(state):
    moves = []
    for x in range(BOARD_X):
        for y in range(BOARD_Y):
            if state[3][y][x] == 0:
                moves = moves + [(x, y)]
    return moves
```

Successor state function

The successor state function replaces the current queen position (state[2]), with the coordinates of the new action, counts the number of controlled squares (state[0]), produces a list with the controlled squares coordinates (state[1]), and turns the value of the current square to 1.

```
def qc_successor_state(action, state):
    board = deepcopy(state[3])
    x_position = action[0]
    y_position = action[1]
    control_list = list(dict.fromkeys(state[1] + controlled_squares_list(x_position, y_position)))
    control_count = len(control_list)
    board[y_position][x_position] = 1
    return control_count, control_list, (x_position, y_position), board
```

```
# Outputs the list of controlled and occupied squares by a queen
def controlled_squares_list(x, y):
    row = queen_row(y)
    column = queen_column(x)
    diagonal = queen_diagonal(x, y)
    squares_list = list(dict.fromkeys(row + column + diagonal))  # To remove duplicate squares
    return squares_list


def queen_row(y):
    row = []
    for x in range(BOARD_X):
        row = row + [(x, y)]
    return row


def queen_column(x):
    column = []
    for y in range(BOARD_Y):
        column = column + [(x, y)]
    return column
```

```
# Calculate all squares controlled in diagonal by a queen in (x, y)
def queen_diagonal(x, y):
    diagonal_left_right = []  # Diagonal that goes from up left to down right
    diagonal_right_left = []  # Diagonal that goes from up right to down left

    # Initial position for left-to-right diagonal (as up and left as possible)
    min_left_right = min(x, y)
    x_left_right = x - min_left_right
    y_left_right = y - min_left_right

    # Initial position for right-to-left diagonal (as up and right as possible)
    x_difference = (BOARD_X - 1) - x  # Difference between x position of the queen and most right x border of the board
    min_right_left = min(x_difference, y)
    x_right_left = x + min_right_left
    y_right_left = y - min_right_left

    # Count squares in the left-to-right diagonal
    while x_left_right < BOARD_X and y_left_right < BOARD_Y:
        diagonal_left_right += [(x_left_right, y_left_right)]
        x_left_right += 1
        y_left_right += 1

    # Count squares in the right-to-left diagonal
    while x_right_left >= 0 and y_right_left < BOARD_Y:
        diagonal_right_left += [(x_right_left, y_right_left)]
        x_right_left += -1
        y_right_left += 1

    diagonal = list(dict.fromkeys(diagonal_left_right + diagonal_right_left))  # Remove duplicate squares
    return diagonal
```

Test for goal state

The goal state is reached by comparing the number of controlled squares to the number of total squares in the board. If they are equal, the goal is reached.

```
def qc_test_goal_state(state):
    if state[0] == BOARD_X * BOARD_Y:
        print("\nGOAL STATE:")
        print_board_state(state)
        print("\nThe minimum number of queens to cover a ", BOARD_X, "*", BOARD_Y,
              " chessboard is equal to the \"Path Length\" below:")
        return True
    return False
```

Heuristic

A simple heuristic has been implemented to compare the amount of time required to run the tests with and without a heuristic. It calculates the number of empty squares left for a given state. The program needs around 1-2 minutes to run all tests without the heuristic, and less than 1 second to run the same tests with a heuristic.

Test	No Heuristic	Heuristic
1	0.0006132 seconds	0.0004959 seconds
2	0.0025147 seconds	0.0007185 seconds
3	0.0305226 seconds	0.0020316 seconds
4	7.4658206 seconds	0.0075672 seconds
5	40.8777228 seconds	0.0079321 seconds
6	33.5377019 seconds	0.0078356 seconds
7	16.8255407 seconds	0.0074548 seconds
8	0.0232458 seconds	0.0015514 seconds
9	23.0413705 seconds	0.0092784 seconds
10	6.5759377 seconds	0.0625318 seconds