1st March 2020

Lab 5 Report

K. Sai Anuroop (170030035) and Mehul Bose (170030010)

Problem domain:

We chose to work on solving 4 x 4 Sudoku puzzles.

A 4 x 4 Sudoku puzzle consists of 16 cells divided into 4 quadrants of 4 cells each.
A valid solution of the puzzle is one where the numbers 1 to 4 appear exactly once in each row, column and quadrant.

	2	**1**	
3			
	1		

Puzzle

1	3	4	2
4	**2**	**1**	3
3	4	2	1
2	**1**	3	4

Valid solution

1	2	4	2
4	**2**	**1**	3
3	4	4	1
2	**1**	3	4

Invalid solution

Solution space consists of 4 x 4 matrices with initial entries of the problem retained and other added entries.

1	4		
	2	**1**	
3			
	1		

1	3	4	2
4	**2**	**1**	3
3	4	2	1
	1		

1	3	4	2
4	**2**	**1**	3
3	4	2	1
2	**1**		

Some matrices in the solution space for the example problem chosen above

We represent empty cells in the solution space by 0

In our example, here are the start and the goal nodes.

0	0	0	0
0	**2**	**1**	0
3	0	0	0
0	**1**	0	0

Start node

1	3	4	2
4	**2**	**1**	3
3	4	2	1
2	**1**	3	4

Goal node

Pseudo-code for different implementations of sudoku solver:

function to process input sudoku puzzle from text file into matrix
process_input(input_file)

```
{
    // define a matrix
    // open file and read contents of the file into matrix
    // return matrix
}
```

function to check if goal state is reached —> **(goal test)**
Note that it suffices to check every cell of the matrix is non-zero to conclude we reached the goal state, as we do validation of every state before processing and expanding it further. We don't have invalid states when the matrix is filled-in completely and thus it is the goal state.
is_matrix_solved (matrix)

```
{
    // if every cell of the matrix is non-zero
        // return true
}
```

function to get the next most constrained empty cell in the matrix (heuristic mode = 1)
get_most_constrained_cell (matrix)

```
{
    // define max_constraints
    // define cell with max_constraints
    // for every empty cell in the matrix
        // define constraints
        // if there exists a non-zero entry in this cell's row
            // increment constraints
            // break
        // if there exists a non-zero entry in this cell's column
            // increment constraints
            // break
        // if constraints > max_constraints
            // max_constraints = constraints
            // cell with max_constraints = current cell
    // return cell with max_constraints
}
```

function to get the next empty cell in the matrix in the order of traversal (heuristic mode = 0)

get_next_empty_cell (matrix)

```
{
    for i in range(0,4):
        for j in range(0,4):
            if matrix[i][j] == 0:
                return i, j
}
```

function to calculate the constraint number of the given configuration of matrix (state)

Note that this is the default heuristic for sorting neighbours of the current state using the constraint number of the neighbour matrix

get_constraint_number_of_matrix (new_matrix, val, x, y)

```
{
    // define constraints
    // for every empty cell in the matrix
        // if there exists a non-zero entry in this cell's row
            // increment constraints
            // break
        // if there exists a non-zero entry in this cell's column
            // increment constraints
            // break
        // if constraints > max_constraints
    // return constraints
}
```

function to get the quadrant of a given cell in the matrix

get_quadrant (x, y)

```
{
    if ((x == 1 or x == 2) and (y == 1 or y == 2)):
        return 0
    elif ((x == 1 or x == 2) and (y == 3 or y == 4)):
        return 1
    elif (x == 3 or x == 4) and (y == 1 or y == 2)):
        return 2
    else:
        return 3
}
```

function to check if a given configuration of matrix (state) is valid

is_valid_configuration (matrix, x, y, val)

```
{
    // check if the value entered val in current cell (x,y) is unique in its column, if not return not
       valid
    // check if the value entered val in current cell (x,y) is unique in its row, if not return not
       valid
    // check the quadrant of the value entered val in current cell (x,y) and check if all the values
       in this quadrant are unique, if not return not valid
    // return is valid
}
```

function to propagate movement of changes to a given state to its subtree

```
propagate_movement (node, open_list, closed)
{
    // for each neighbour s of current matrix m
        // if neighbour is valid
            // if g(m) + k(m,s) < g(s)
                // g(s) = g(s) + k(m,s)
                // set parent of the neighbour s to current matrix m
                // if s is already in closed list
                    // propagate_movement (s)

}

astar_search (matrix, mode, hmode)
{
    // initialise priority queue for open list
    // create root node for storing initial matrix
    // initialise queue for closed list
    // while open list is not empty
        // increment states explored
        // pop the head from the open list
        // append this matrix to the closed list
        // check for goal state
        // apply meta heurisitc according to mode value and get next empty cell
          —> (part of move gen)

        // for each neighbour n of current matrix m            —> (part of move gen)
            // if neighbour is valid
                // if neighbour is a new node
                    // g(n) = g(m) + k(m,n)
                    // compute h(n) according to the heuristic hnode
                    // f(n) = g(n) + h(n)
                    // set parent of the neighbour n to current matrix m
                    // push to open list based on its f value

                // if neighbour is already in open list
                    // if g(m) + k(m,n) < g(n)
                        // g(n) = g(m) + k(m,n)
                        // compute h(n) according to the heuristic hnode
                        // f(n) = g(n) + h(n)
                        // set parent of the neighbour n to current matrix m

                // if neighbour is already in closed list
                    // if g(m) + k(m,n) < g(n)
                        // g(n) = g(m) + k(m,n)
                        // compute h(n) according to the heuristic hnode
                        // f(n) = g(n) + h(n)
                        // set parent of the neighbour n to current matrix m
                        // propagate_movement (m)
}

main( )
{
    // process input
    // driver code to call A* search algorithm and print results
}
```

Meta Heuristic:

Mode in the driver code defines the meta heuristic to be chosen based on the maximum constrained empty cell.
Here is the function defining the meta heuristic:

<u>function to get the next most constrained empty cell in the matrix (heuristic mode = 1)</u>

```
get_most_constrained_cell (matrix)
{
    // define max_constraints
    // define cell with max_constraints
    // for every empty cell in the matrix
        // define constraints
        // if there exists a non-zero entry in this cell's row
            // increment constraints
            // break
        // if there exists a non-zero entry in this cell's column
            // increment constraints
            // break
        // if constraints > max_constraints
            // max_constraints = constraints
            // cell with max_constraints = current cell
    // return cell with max_constraints
}
```

<u>function to calculate the constraint number of the given configuration of matrix (state)</u>

```
get_constraint_number_of_matrix (new_matrix, val, x, y)
{
    // define constraints
    // for every empty cell in the matrix
        // if there exists a non-zero entry in this cell's row
            // increment constraints
            // break
        // if there exists a non-zero entry in this cell's column
            // increment constraints
            // break
        // if constraints > max_constraints
    // return constraints
}
```

A* Heuristic Functions h(n):

We assume that every edge in the graph costs 1 unit, i.e., $g(n) = g(m) + 1$ (or equivalently, $k(m,n) =1$) where n is a child of m. This is a valid assumption as in Sudoku solving, every transition from a given state to next valid state should have equal weightage.
<u>Heuristic hmode = 0 (Over-estimating Heuristic):</u>
This heuristic returns the constraint number of a given matrix. Clearly, this is an over-estimating heuristic as the number of constraints in a given matrix is greater than or equal to the number of empty cells (one empty cell can have 0 or more constraints).

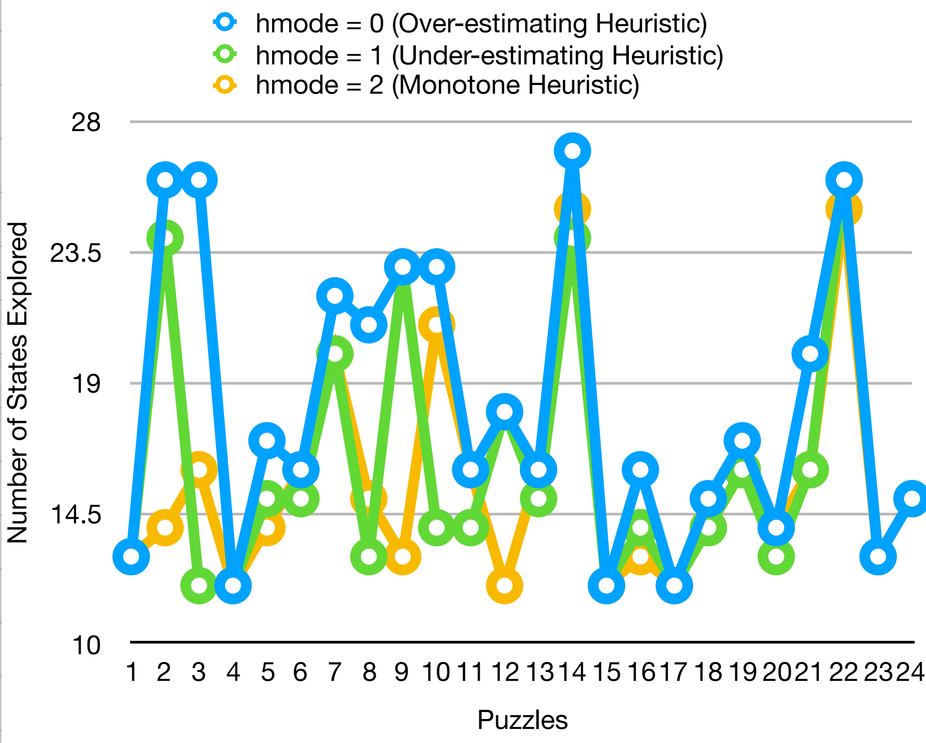
<u>Heuristic hmode = 1 (Under-estimating Heuristic):</u>
This heuristic returns returns a random number in the range of 0 and 1. Since every edge in the graph is assumed to cost 1 unit, this is obviously underestimating heuristic.

<u>Heuristic hmode = 2 (Monotone Heuristic):</u>
This heuristic returns the number of empty cells in the matrix as the estimate for distance to goal state. Since in every state, we fill-up one empty cell and go to the next state, we see that the next state has this heuristic value to be one less than the current state, i.e., $h(n) = h(m) - 1$, which satisfies the monotone property that $h(m) - h(n) <= k(m,n)$ where n is the child of m.

Analysis of performance of the A* search algorithm with heuristic modes 0, 1 and 2:

Number of States Explored

Puzzle	hmode = 0	hmode = 1	hmode = 2
1	13	13	13
2	26	24	14
3	26	12	16
4	12	12	12
5	17	15	14
6	16	15	16
7	22	20	20
8	21	13	15
9	23	23	13
10	23	14	21
11	16	14	16
12	18	18	12
13	16	15	16
14	27	24	25
15	12	12	12
16	16	14	13
17	12	12	12
18	15	14	14
19	17	16	16
20	14	13	14
21	20	16	16
22	26	26	25
23	13	13	13
24	15	15	15



Comparative analysis of performance of the A* search algorithm with heuristic modes 0, 1 and 2:

It is evident from the plot above that the over-estimating heuristic takes more number of states explored to reach the goal state. This is justified as the heuristic is over-estimating and ignores the shorter path.
For the under-estimating heuristic, we observe that the number of states explored are less than that of the over-estimating heuristic.
For the monotone heuristic, we observe that for most of the puzzles, it is reaches the goal state in least number of states explored. This is justified as the heuristic has the monotone property and it should yield optimal 'path'.

Conclusion:

Slight discrepancies that we observe in the plot above are due to the two reasons that number of states explored is not equivalent to the path length to reach goal state and that the under-estimating heuristic is random in nature. In fact, it was observed that g(Goal_State) for all the underestimating and monotone heuristics were the optimal g values (here, equal to 12, as every input matrix had 12 empty cells to be filled and according to our notion that every graph edge has equal weight of 1, the shortest path should be of 12 units as expected). So, plotting g(Goal_State) vs puzzles does not shed light on the variations, hence we chose to plot the number of states explored. Another point to be noted here is that while solving the Sudoku puzzle, we never empty an already filled cell and re-fill it with new values, so there does not exist a back-edge in the graph so A* algorithm does not visit a node in the closed list ever. Thus, as the concept of path here is slightly vague and due to the inherent nature of Sudoku problem, using A* is not a very good idea for solving Sudoku puzzles.

END OF REPORT