26th January 2020

Lab 2 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 check if a given state is tabu

```
is_tabu(matrix, x, y, i, tabu_list, tenure)
{
        // if the given matrix is in tabu list, return true or else return false
}
```

```
best_first_search (matrix, mode)
{
        // initialise priority queue and push initial matrix into it
        // while queue is not empty
                // increment states explored
                // pop the head from the priority queue
                // check for goal state
                // apply heurisitc according to mode value and get next empty cell
                   —> (part of move gen)

                // for each neighbour of current matrix              —> (part of move gen)
                        // if neighbour is valid, push to priority queue based on its heuristic value
}

hill_climbing_search (matrix, mode)
{
        // define queue to contain matrix states
        // initialize the queue with initial state
        // while queue is not empty
                // increment states explored
                // pop the head of the queue
                // check for goal state
                // apply heurisitc according to mode value and get next empty cell
                   —> (part of move gen)

                // define list to store neighbours
                // for each neighbour of current matrix              —> (part of move gen)
                        // if neighbour is valid, add to neighbours list
                // sort the neighbours according to heuristic value
                // if local optimum (no further valid neighbours) is reached, return current state and
                   mention that goal state is not reached
                // add the neighbour with best heuristic value to the queue
}

variable_neighbourhood_descent_search(matrix, mode)
{
        // define queue to contain matrix states
        // initialize the queue with initial state
        // initialize neighbour density to 1
        // while queue is not empty
                // increment states explored
                // pop the head of the queue
                // check for goal state
                // apply heurisitc according to mode value and get next empty cell
                   —> (part of move gen)
        // define list to store neighbours
        // for each neighbour of current matrix              —> (part of move gen)
                        // if neighbour is valid, add to neighbours list
        // sort the neighbours according to heuristic value
        // if local optimum (no further valid neighbours) is reached, restart from initial state and
           increase neighbour density
        // else add those many neighbours to the queue as is the neighbour density
}
```

```
beam_search(matrix, beam_density, mode)
{
        // initialise priority queue and push initial matrix into it
        // while queue is not empty
                // increment states explored
                // pop the head from the priority queue
                // check for goal state
                // apply heurisitc according to mode value and get next empty cell
                   —> (part of move gen)

                // for each neighbour of current matrix              —> (part of move gen)
                        // if neighbour is valid, push to priority queue based on its heuristic value
                           and only those many neighbours are pushed as is the size of beam width

}

tabu_search(matrix, tenure, mode)
{
        // define queue to contain matrix states
        // initialize the queue with initial state
        // define a tabu list of size tenure
        // while queue is not empty
                // increment states explored
                // pop the head of the queue
                // check for goal state
                // apply heurisitc according to mode value and get next empty cell
                   —> (part of move gen)

                // define list to store neighbours
                // for each neighbour of current matrix              —> (part of move gen)
                        // if neighbour is valid and is not in tabu list, add to neighbours list
        // sort the neighbours according to heuristic value
        // if local optimum (no further valid neighbours) is reached, restart from initial state
        // else add neighbours to the queue and to the tabu list. If tabu list is full, pop the oldest
           entry (head) from the list
}

main( )
{
        // process input
        // driver code to call different search algorithms and print results
}
```

Heuristics:

There are two different kinds of heuristics used, one which fetches the next cell according to some value; and the other which sorts the neighbours according to some value.

Mode in the driver code defines the first type of heuristic to be chosen based either on the maximum constrained empty cell or empty cell in the order of matrix traversal; and the second type of heuristic is the default one for sorting neighbours of the current state using the constraint number of the neighbour matrix.

Here are the functions defining the first type of heuristic:

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
}
```

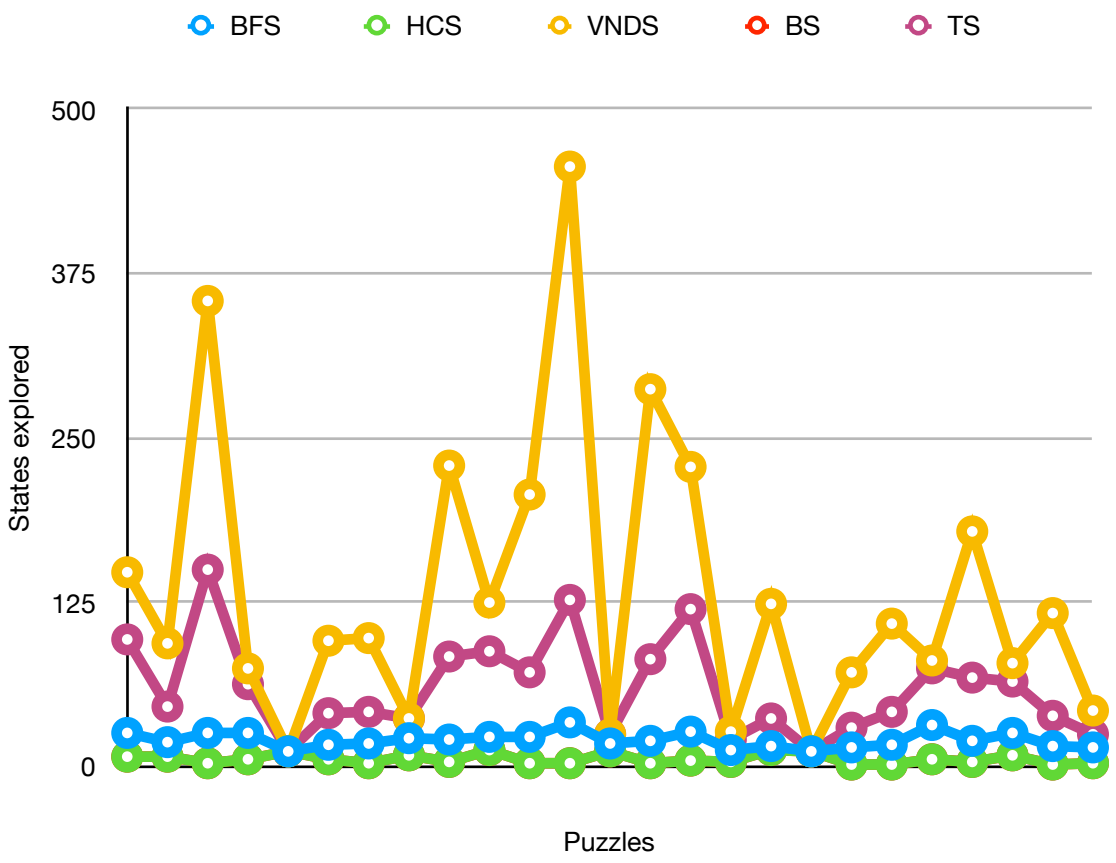
Here is the function describing second type of heuristic, which is the default one for sorting neighbours of the current state using the constraint number of the neighbour matrix:

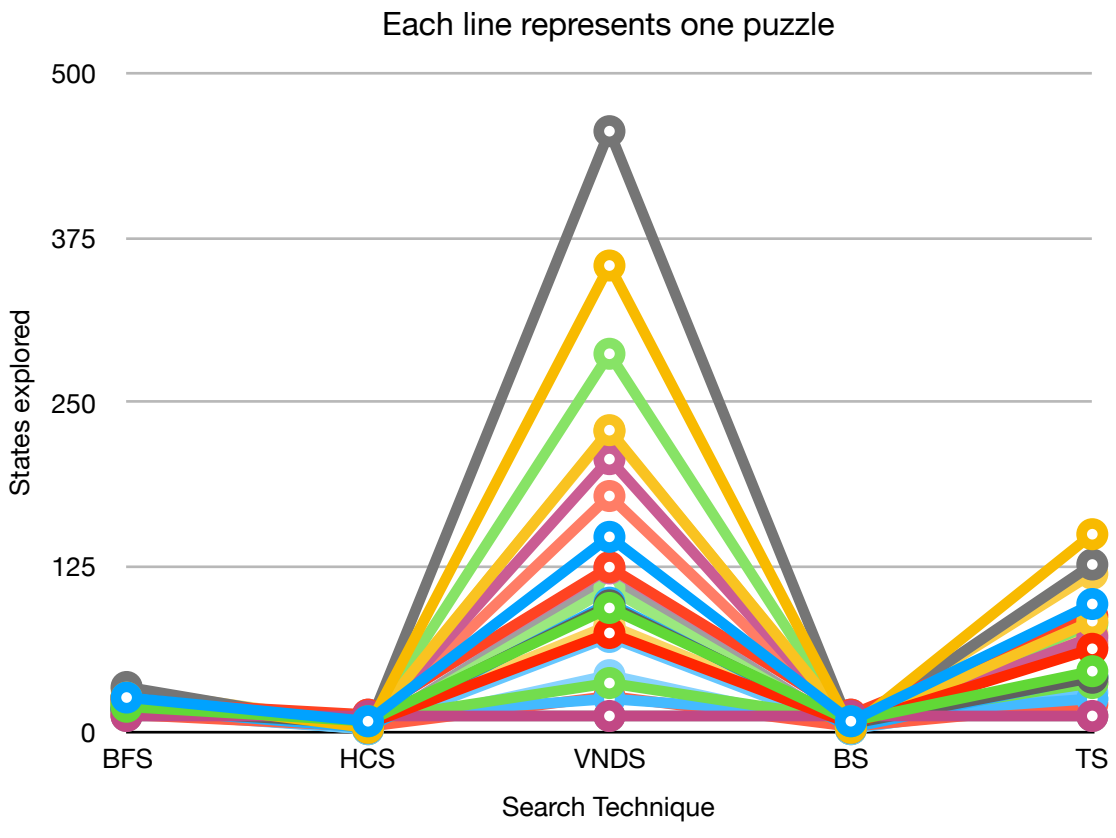
function to calculate the constraint number of the given configuration of matrix (state)

```
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
}
```

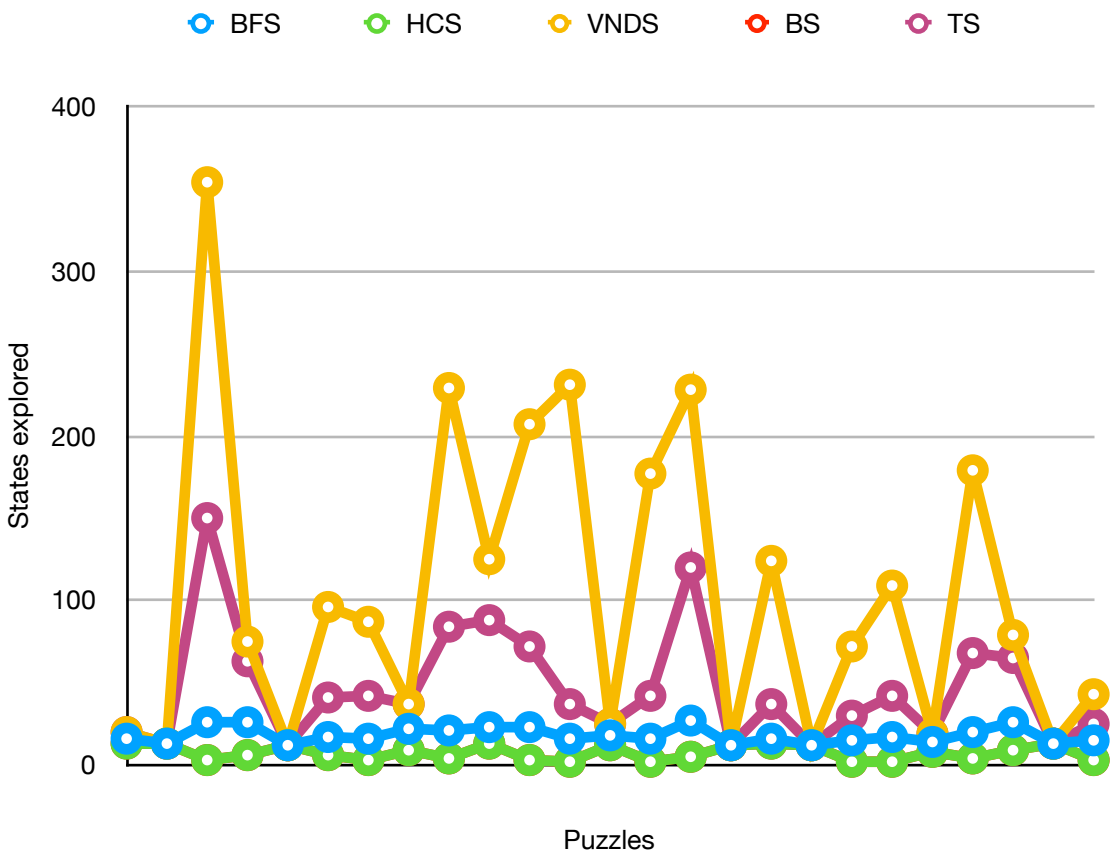
In the following pages, we describe our observations and analyses of these search techniques using two heuristic modes which were tested for 25 different 4 x 4 sudoku puzzles.

Analysis of performance of various algorithms with heuristic mode 0:
(get the next empty cell in the matrix in the order of traversal)

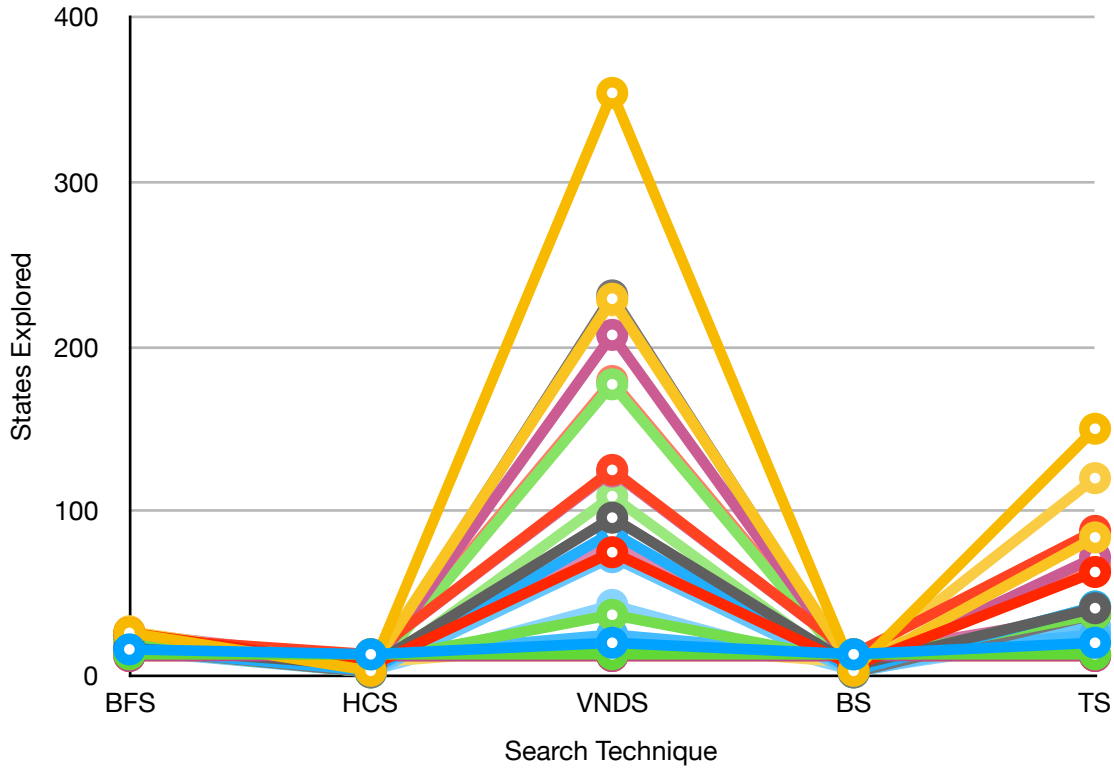




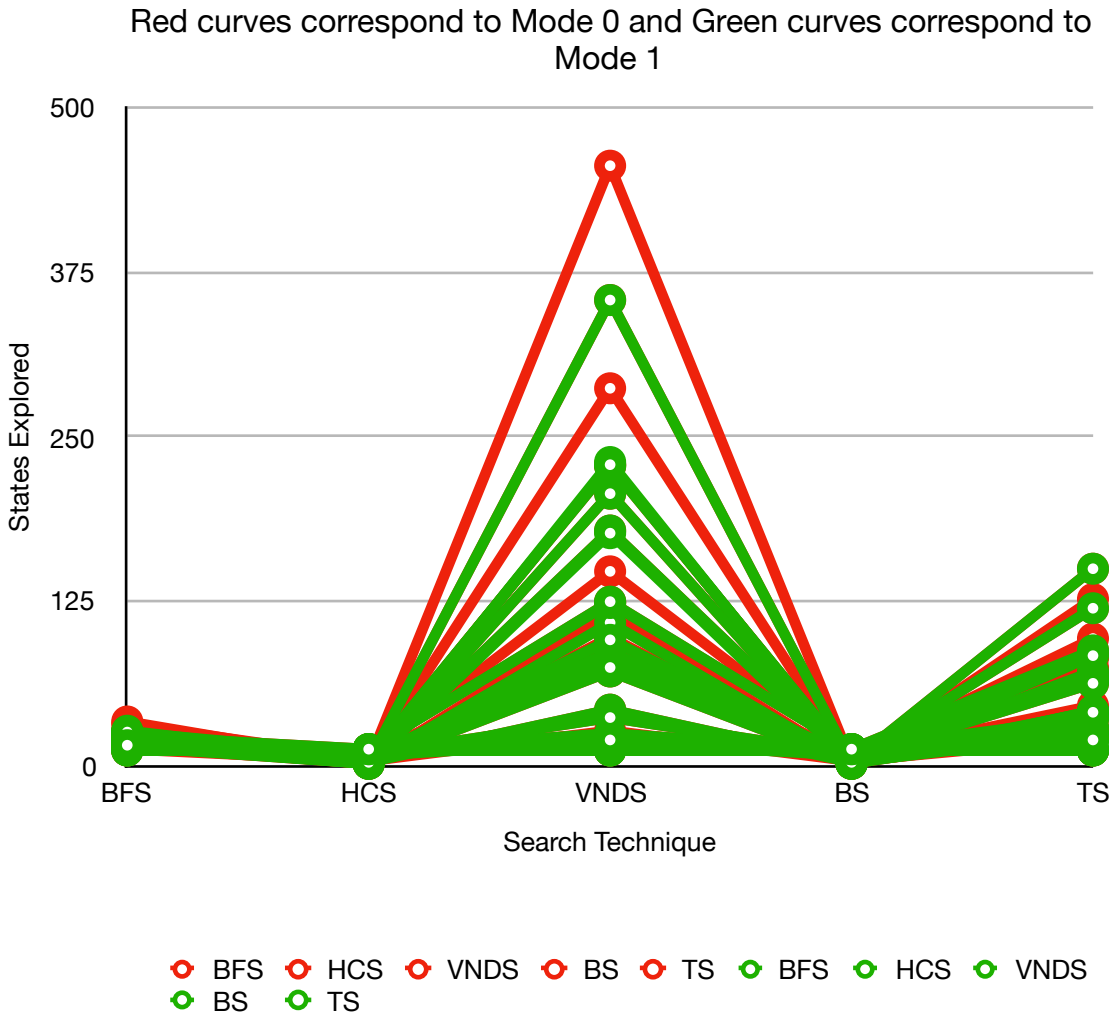
Analysis of performance of various algorithms with heuristic mode 1:
(get the next most constrained empty cell in the matrix)

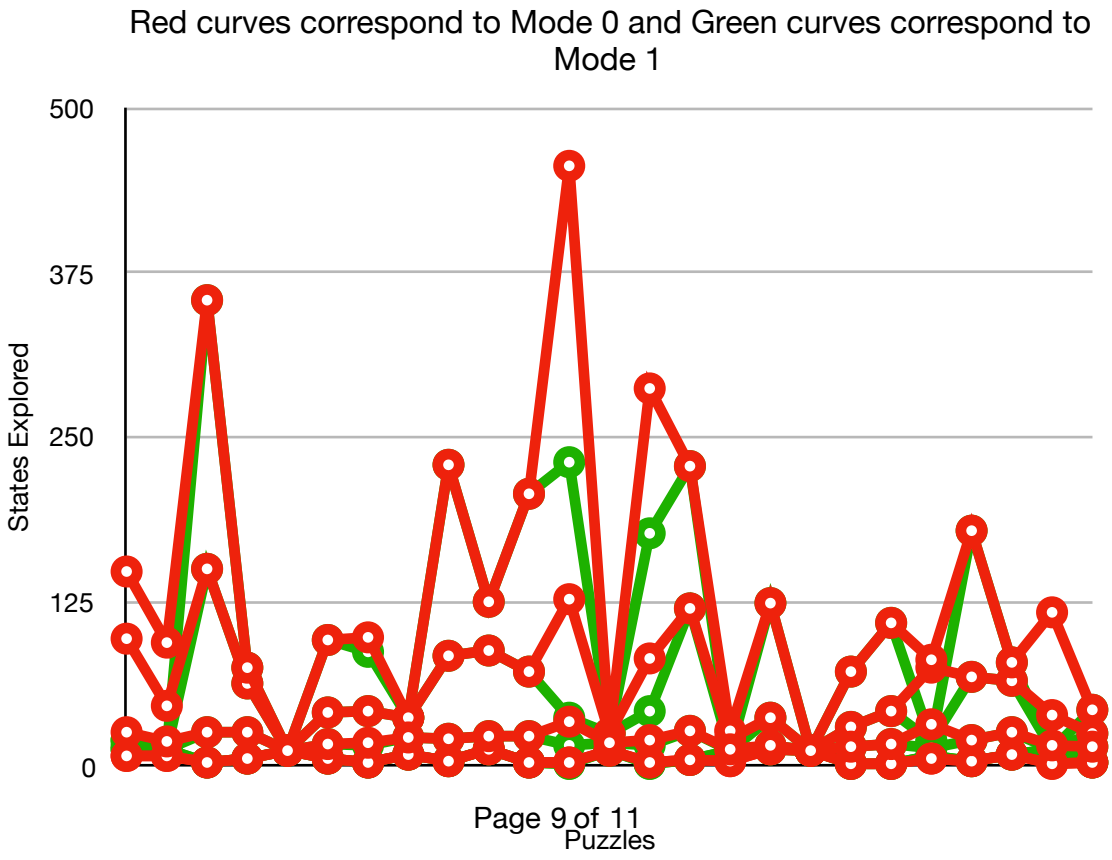


Each line represents one puzzle



Comparative analysis of performance of various algorithms for two modes of heuristic:





Heuristic mode 0 vs Heuristic mode 1:

Heuristic mode 1 is observed to perform slightly better than heuristic mode 0 for the very reason that it is a greedy strategy (get the next most constrained empty cell in the matrix). Since we fill-in the most constrained cells first, in the successive empty cells, we are reducing the number of valid values that ca be filled-in and thus our algorithms' branching factor decreases as a result of which less states are explored. Whereas, in heuristic mode 0, as we are going in the order of traversal of matrix, if the highly constrained cells happen to be arranged in that order, the algorithms perform better or else the branch factor in successive steps remains almost the same without significant decrease.

Analysis and observations of Best-First Search:

Best-First Search is observed to give the best performance in terms of number of states explored before reaching goal state. Also, it always reaches goal state no matter what the puzzle is and is thus complete in nature.

Best-First Search vs Hill Climbing:

- Hill-Climbing does not reach goal state in 20 out of 25 puzzles in heuristic mode 0. So, its probability of finding solution in heuristic mode 0 (w.r.t. examples considered) is 20%.
- Hill-Climbing does not reach goal state in 16 out of 25 puzzles in heuristic mode 1. So, its probability of finding solution in heuristic mode 1 (w.r.t. examples considered) is 36%.
- Also, Hill-Climbing (if) reaches goal state in less number of states when compared to Best-First Search as is expected.
- So, Hill-Climbing is faster but not complete.

This also implies Hill-Climbing algorithm works better in mode 1 (get the next most constrained empty cell in the matrix) which is a greedy strategy.
Best-First Search is complete and almost has optimal performance in terms of number of states explored before reaching goal state.

Analysis of Beam Search:

It is observed that
- Beam Search results in the goal state in the same number of states explored as does Best-First Search, if the beam width is equal to 2 (and above).
- Beam Search (if it does) results in the goal state in the same number of states explored as does Hill-Climbing Search, if the beam width is equal to 1.

Above observations are justified as Beam Search is designed to get the best of both Best-First and Hill-Climbing Search techniques. When the beam width is equal to 1, we are choosing only the neighbour with best heuristic value and is thus in principle to Hill-Climbing. Also, when beam width is equal to 2 (and above), we are allowing the valid neighbours to be added to the queue and hence it is behaving similar to Best-First Search.

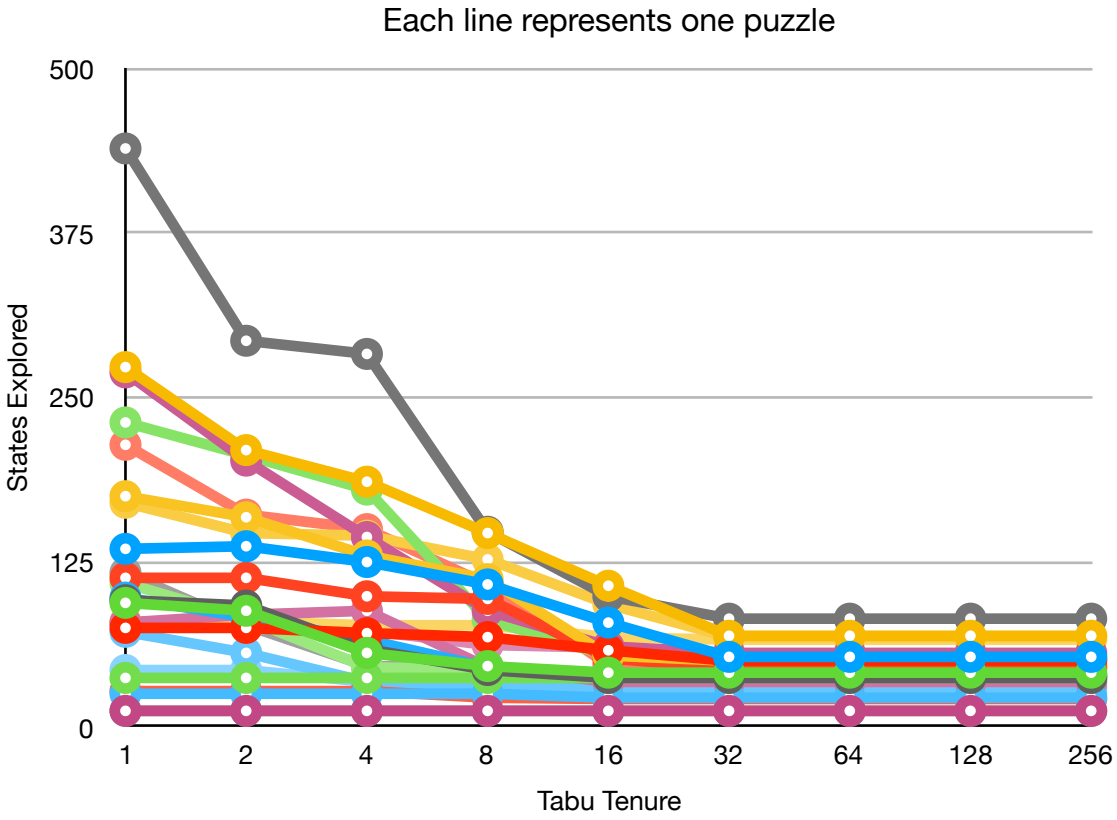
Note that maximum meaningful beam width would be 4, as there are a maximum of 4 different values that can fill any given empty cell and these values also reduce in number (less than 4) depending on the number of constraints on the empty cell in consideration.

Comparative analysis of Variable Neighbourhood Descent Search, Beam Search and Tabu Search:

- It is observed that performance of Tabu Search in terms of number of states explored falls in between Variable Neighbourhood Descent Search and Beam Search. This is so because Variable Neighbourhood Descent Search simply keeps on increasing the neighbourhood density every time it reaches a local optimum (or runs out of valid moves) and in the process may visit the already visited states. In Tabu Search, since the algorithm is such that it does not visit the

states in tabu list, the chances of it visiting the already visited states is comparatively a lot lesser than Variable Neighbourhood Descent Search even at times it restarts after getting stuck at local optima.

- For some puzzles, its performance is observed to be as good as that of Beam Search and Best-First Search.
- We see that as Beam Search is midway between Best-First Search and Hill-Climbing Search in its methodology of search, and as Variable Neighbourhood Descent Search and Tabu Search address the drawbacks of Hill-Climbing Search, its performance is seen to be better than Variable Neighbourhood Descent Search and Tabu Search.



Analysis of Tabu Search:

This graph plots States Explored vs Tabu Tenure for different sudoku puzzles in heuristic mode 0.

- It is observed that after around Tabu Tenure of 32, all puzzles are solved in about same number of States Explored.
- Increasing Tabu Tenure beyond this value does not have any significant observable effect on number of States Explored before reaching goal state.

This can be explained by the fact that an increase in Tabu Tenure means visited states stay longer in the Tabu List and are thus not re-visited often, leading to goal state being reached in less number of steps. It is also evident that this saturation is because after a particular value of Tabu Tenure, almost all the visited states can be accommodated in the Tabu list and frequent evictions may not take place, coupled with the reasonable assumption that the algorithm will not visit the initial states when it is close to the goal state.

Similar trends can be observed when the algorithm is run in heuristic mode 1.

END OF REPORT