CS312: Lab-3

By: Mayank Mittal *(190030026)*
Ayush Gupta *(190030007)*

Random *n-SAT* Problem:

In our program, we are taking the inputs of the number of variables and the number of clauses. Every clause is separately written in file, *'input.txt'* and is generated via the file, *'generate.py'*

This file shall run automatically when running the main program *'17.py'* Make sure that *'generate.py'* and *'17.py'* are in the same directory

The number of literals in every clause have been fixed to three. It can be changed via *'generate.py'* by changing the value of *'NumberOfLiterals'* if required. Since the number of literals weren't mentioned as the input, we decided to fix it to three.

'~' sign is used for the negation. All the output would be printed in the file *'output.txt'*

All the present states while searching for the states in all these three searching algorithms are printed. For a variable *x, whenever x is written in the state, it means that x = 1* and *whenever ~x is written in the state, it means that x = 0* Hence, for a state *~a b c d* it means *a = 0, b = 1, c = 1, d = 1*

Note that ***the clauses and the starting state are generated randomly***. ***Goal state is the state where all the clauses are True***. There are in total 2^n possible states possible.

Move Generation Function:

```
def nextGenfunction(string, density):
    neighbour = []
    allCombinations = list(combinations(range(len(string)), density))
    for combination in allCombinations:
        new_string = string
        for position in combination:
            new_string = string[:position] + \
                str(1 - int(new_string[position])) + string[position+1:]
        value = Heuristic(Clause, new_string)
        neighbour.append((value, new_string))
    return neighbour
```

Our states are represented by Binary Strings, hence our MOVEGEN function takes string as input, as well as the parameter density. Density is usually initialized to one, but can be increased, as is used in Variable Neighborhood Descent. This density is basically the number of variables toggled to go to the state/neighborhood.

Goal Test:

Whenever the value of Heuristic Function is the same as the number of clauses, we have reached the goal state, which means that our SAT problem is solved, and our Heuristic Function is just the number of clauses satisfied.

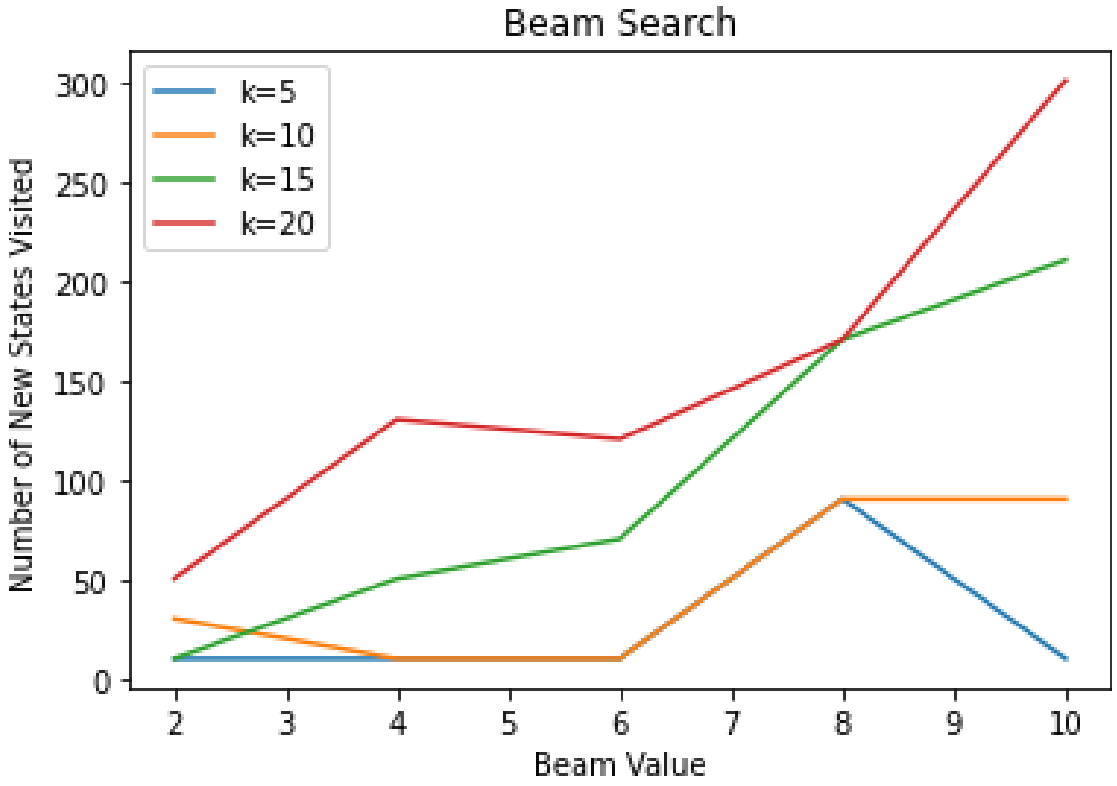
Thus the goal test is implemented using simple conditional statements in the program.

Heuristic Function:

As said above, here the Heuristic Function is the number of clauses satisfied. It is the only requirement for our problem, hence the only Heuristic function we define here.

```
def Heuristic(Clause, stringOfVar):
    state_value = 0
    for PerClause in Clause:
        for literal in PerClause:
            if int(literal) < 0 and stringOfVar[abs(int(literal)) - 1] == '0':
                state_value += 1
                break
            if int(literal) > 0 and stringOfVar[abs(int(literal)) - 1] == '1':
                state_value += 1
                break
    return -1*state_value
```

Beam Search Analysis:



The number of states explored in beam search generally increases as the beam width increases.

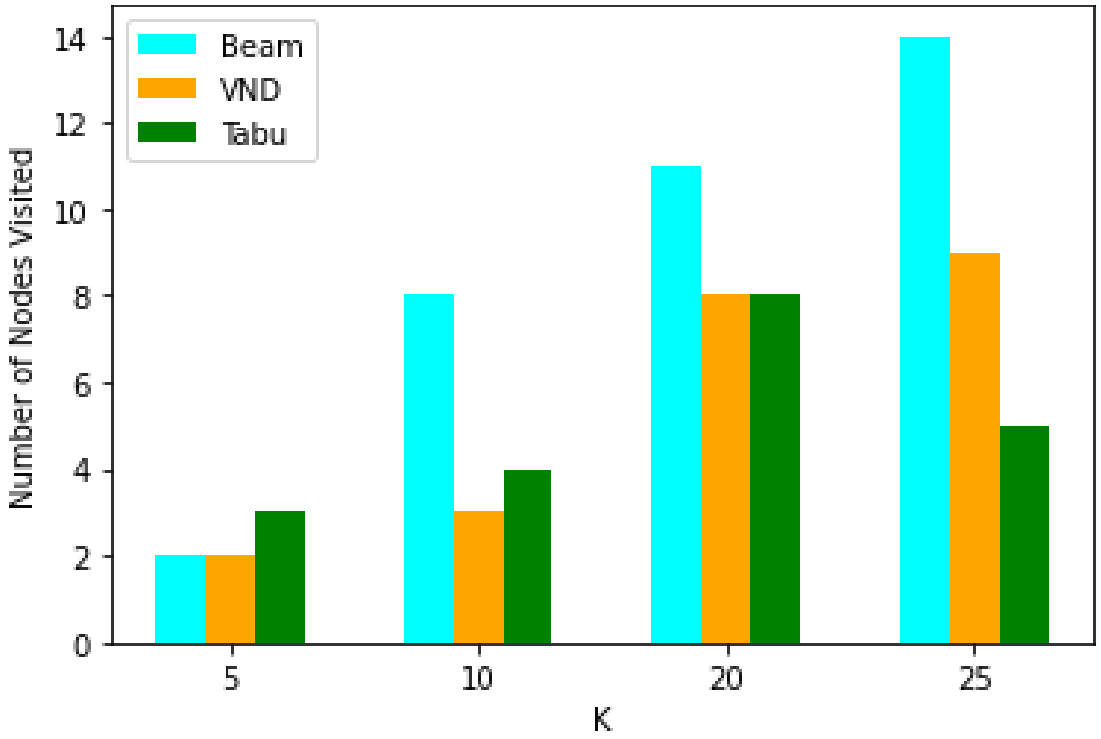
Tabu Search Analysis:

When running Tabu Search for different Tabu List sizes, we observed that the number of states explored didn't vary much at all. Hence, we can say that the states explored are independent. We ran our program for values:

$$n = 10, k = 20, tt = 2, 4, 6, 8, 10$$

Hence, we have not plotted the graph for the same.

Number of States explored for each Search:



Beam Width = 6 , Tabu Tenure = 6

As we can see, the number of nodes explored in the Beam Search are always more than the total number of nodes explored in Variable Neighborhood Descent and Tabu Search. This is because for larger values of k, the number of nodes increases considerably.