CS312 - Artificial Intelligence Lab Assignment 3

Arun R Bhat (190010007), Mohammad Sameer (190010024)

1 Introduction

Uniform Random-4-SAT is a collection of SAT problem distributions created by generating 3-CNF formulae at random. Clauses that contain numerous copies of the same literal or are tautological are not acceptable for the construction of the problem instance (i.e., they contain a variable and its negation as a literal). As a result, each n and k value generates a Random-4-SAT distribution. The union of these distributions over all n and k is Uniform Random-4-SAT.

2 Domain

2.1 State Space

All possible permutations of truth values of all literals make up the **state space**. A state is represented using a dictionary in python, where the capital letters represent a literal and small letters represent its negation. And these literal make up the keys of the dictionary and values are their truth values.
Example:

```
{'A': 0, 'a': 1, 'B': 1, 'b': 0, 'C': 1, 'c': 0, 'D': 1, 'd': 0}
```

To represent the Tabu Tenure a similar dictionary has been used, but excluding the small letters, and the values represent the key's current tabu tenure
Example:

```
{'A': 0, 'B': 1, 'C': 2, 'D': 0}
```

2.2 Start Node & Goal Node

By assigning random truth values to all the literals, the **start node** is determined. The **goal node** is the one where the given formula of clauses is satisfied. Consider the following clause, where the capital letters represent a literal and small letters represent its negation.:

```
[['B', 'c', 'd'],
 ['A', 'b', 'c'],
 ['a', 'B', 'D'],
 ['a', 'c', 'D'],
 ['A', 'B', 'C']]
```

Example of Start Node:

```
{'A': 0, 'a': 1, 'B': 1, 'b': 0, 'C': 1, 'c': 0, 'D': 1, 'd': 0}
```

Example of Goal Node:

```
{'A': 1, 'a': 0, 'B': 1, 'b': 0, 'C': 1, 'c': 0, 'D': 1, 'd': 0}
```

3 Pseudo Codes

3.1 moveGen(state, k)

This function takes a state and ‘k’ as input, where ‘k’ represents the number of bits to flip in each neighbour.

```
def moveGen(state, k):
    #Argument 'k' indicates the number of bits to flip
    #We iterates from 0 to 2^{numOfLiterals} - 1 in binary string format
    #In the above string, 1s postion represent bits to be flipped
    neighbours = []      #List to store the neighbours
    global numOfVaribles    #Total number of literals in clause
    for string from 0 to {2^{numOfVariables} - 1}:
        string = toBinary(string, width = numOfVaribles)
        if(string.numOfOnes == k):
            neighbour = state.copy()
            ind = 0
            for x in string:
                if x == "1":
                    assign = neighbour.variables[ind]
                    neighbour.variables[ind] = 1 - assign
                neighbour.findHeuristic()
                neighbours.append(neighbour)
    return neighbours
```

3.2 goalTest(state)

This function takes a state as input and returns True if the state satisfies all the clauses and returns False otherwise.

```
def goalTest(state):
    if {state} satisfies all Clause:
        return True
    else:
        return False
```

4 Heuristic Function

The heuristic function returns an integer value equal to number of clause satisfied by the state.

```
def heuristic(clauses, state):
    value = 0
    for clause in clauses:
        if {state} satisfies clause:
            value += 1
    return value
```

5 Beam Search Analysis for different beam lengths

We have run the beam search algorithm for 3 different clause, with n (number of variables) equal to 4 and k (number of clause) equal to 5, while varying the beam width from 1 to 4. The results are summarized in the table given below.

Beam Width	**Number of States Explored**		
	Clause 1	**Clause 2**	**Clause 3**
1	2	1	2
2	3	2	3
3	4	3	4
4	7	4	5

Table 1: Number of States Explored vs. Beam Width

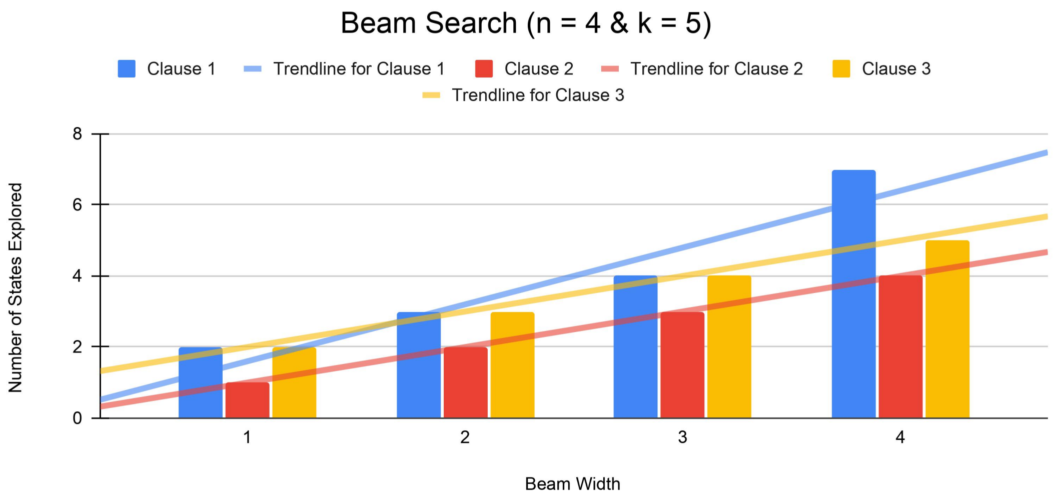

Figure 1: Graph comparing number of states explored vs. Beam Width

The above figure shows the same statistics as the table in a graphical way. From this graph, we can observe that as beam width increases, the number of states explores goes up. This is indeed correct since, the greater the beam width, the fewer states are pruned. And as beam width increases we move from exploitative to explorative search, thus increasing the number of states explored.

6 Tabu Search Analysis for different tabu tenures

We have run the tabu search algorithm for 3 different clause, with n (number of variables) equal to 4 and k (number of clause) equal to 5, while varying the tabu tenure value from 0 to 3. The results are summarized in the table given below.

Tabu Tenure	**Number of States Explores**		
	Clause 1	**Clause 2**	**Clause 3**
0	2	1	2
1	2	1	2
2	2	1	2
3	2	1	2

Table 2: Number of States Explored vs. Tabu Tenure

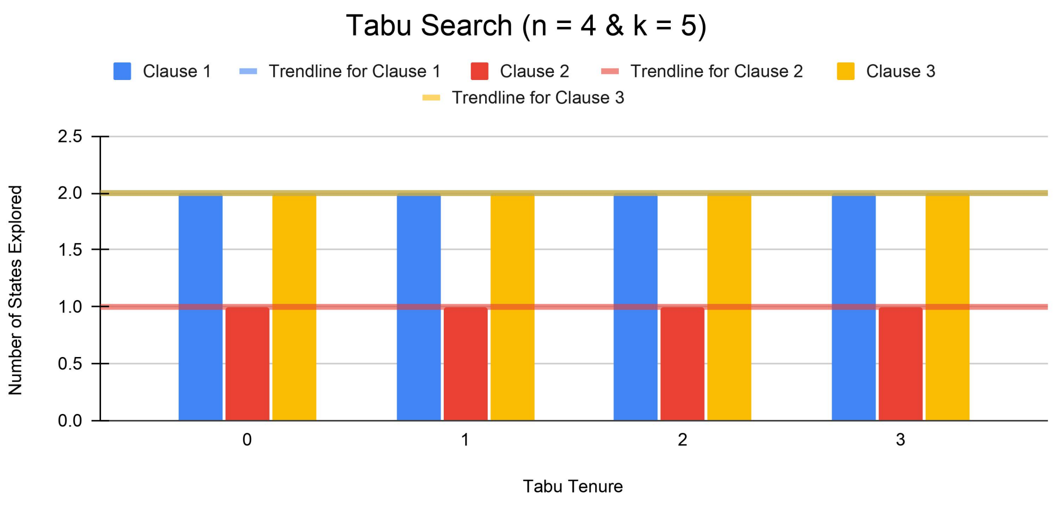

Figure 2: Graph comparing number of states explored vs. Tabu Tenure

The above figure shows the same statistics as the table in a graphical way. From this graph, we can observe that the tabu tenure doesn't effect the number of states explored much. This is due to the fact that there are not many local minima/maxima in this particular domain.

7 Comparison of all 3 Algorithms

To compare all 3 algorithms we have run all 3 algorithms on 3 different clause, with n (number of variables) equal to 4 and k (number of clause) equal to 5, beam width equal to 4 and tabu tenure equal to 3. The results are summarized in the table given below.

Algorithm	**Number of States Explores**		
	Clause 1	**Clause 2**	**Clause 3**
VND	3	2	3
Beam Search	7	4	5
Tabu Search	2	1	2

Table 3: Number of States Explored vs. Algorithms

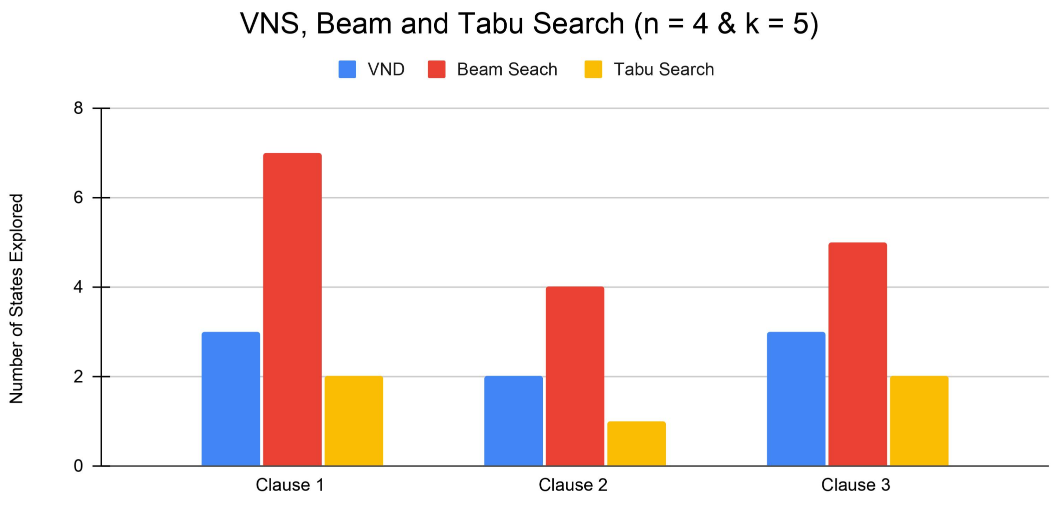

Figure 3: Graph comparing number of states explored vs. Different Algorithms

The above figure shows the same statistics as the table in a graphical way. As seen from this graph, the Variable Neighbour Descent and Tabu Search explore relatively less number of states as compared to Beam Search. The Variable Neighbour Descent and Tabu Search explore almost same number of states on average.