Artificial Intelligence
Lab 3 Report

Vibhuti Ramn - 180010040
Vipul Nikam - 180010041

How to run:

python3 Problems.py

To create problems.txt.

python3 180010040_180010041_Lab2.py

To create output.txt.

Or

./run.sh

Brief description about the domain:

Here heuristic search algorithms are implemented. Uniform Random-4-SAT is a family of distributions of SAT problems obtained by randomly generating 3-CNF formulas. Say with 4 variables & 5 clauses, all the clauses are constructed from 3 literals that are drawn randomly from the 2n possible literals where n indicates the number of variables, so that each possible literal is selected with the same 1/2n probability. Clauses for the construction of the problem instance will not be accepted if they contain multiple copies of the same literal or in case they are tautological, that is, they contain a variable & its negation as literal. Each choice of n & k thus induces a distribution of Random 4-SAT instances. Uniform Random-4-SAT is the union of these distributions over all n & k.

1. State space

Each state is defined using a class which has literal values. Each variable is either 0 or 1. Representation of the state is modified a bit and

one more list is added to store the Tabu Tenure values to the state as shown below in Tabu Search .

In *Variable neighborhood descent*, the state used are ([a, b, c, d])
In *Beam Search*, the state used are ([a, b, c, d])
In *Tabu Search*, the state used are ([a, b, c, d], [M1, M2, M3, M4])

Assuming that all literal values are 0 and that it is allowed to change, it is represented by ([0,0,0,0]) or maybe ([1,1,1,1]) for the variable neighborhood descent algorithms and beam search. In Tabu Search, such a state can be ([0,0,0,0], [0,0,0,0]).

2. Start node and goal node

The start & target Nodes will be calculated by the same logic as the aforementioned algorithms. We will start with all literal values 0 which is allowed to change any literal in the next move.

Start nodes are as follows.

In *Variable neighborhood descent*, the start node used are ([0, 0, 0, 0])
In *Beam Search*, the start node used are ([0, 0, 0, 0])
In *Tabu Search*, the start node used are State = ([0, 0, 0, 0], [0, 0, 0, 0])

The target node or goal node is a node with a literal list [a, b, c, d] such that the Boolean values of a, b, c, d, when placed in a given 4-SAT expression, return 1, that is, satisfies the expression 4-SAT CNF.

3. MOVEGEN and GOALTEST algorithm

pseudo-codes for code are as follows:

movGen: The function takes a state as input and returns a set of states that are accessible from the input state in one step. Code for it as below.

```
procedure moveGen(state)
    nextStates ← ()                    > nextStates as empty set
    for neighbour n of state in order(Heuristic Values) do
        bit = negation(bit)            > a := ã
        neighbours ← new.node()
    return neighbours
```

GoalTest: This function returns True if the input state is target and False otherwise. Code for it as below.

```
procedure goalTest(state)
    if [a, bc, d] satisfies CNF {
        return true
    }
    return false                > state is not goal
```

Variable Neighborhood Descent: The variable neighborhood descent method is obtained if a neighborhood change is performed in a deterministic way. pseudocode is in the algorithm below. It is presented in below algorithm, where the neighborhoods are denoted as

N_k where k = {1, k_{max}}

```
procedure variableNeighbor(x, k max )
    k ← 1
    do{
        x 0 ← arg min y∈N k (x) f (y)          > best neighbor in
        N k(x)
        x, k ← NeighborhoodChange(x, x 0 , k) > change
    neighborhood
    }while k = k_max
    return x
```

Beam Search: pseudocode is in the algorithm below. containsGoal() is a function that determines whether the target state has been reached. score() function uses the heuristic function to score states. prune() function selects the best states to keep.

```
procedure beamSearch()
    initialStates ← currentStates
    do{
        next(currentStates) =: candidatesStates
        score(candidatesStates)
        currentStates := prune(candidatesStates)
    } while != containsGoal(currentStates)
    return state
```

Tabu Search: We used the Tabu Search Algorithm as shown in the below algorithm.

```
procedure tabuSearch() {
        sBest ← s0
        bestCandidate ← s0
        tabuList ← [ ]                          > initializing empty list
        tabuList.push(s0)
        while ! stoppingCondition() {
                sNeighborhood ← getNeighbors(bestCandidate)
                bestCandidate ← sNeighborhood[0]
                for sCandidate in sNeighborhood {
                        if not tabuList.contains(sCandidate) {
                                if fitness(sCandidate) greater fitness(bestCandidate)
                                        bestCandidate ← sCandidate
                        }
                }
                if fitness(bestCandidate) is greater fitness(sBest)
                        sBest ← bestCandidate
                tabuList.push(bestCandidate)
                if tabuList.size is greater maxTabuSize
                        tabuList.removeFirst()
        }
        return sBest
}
```

Heuristic functions considered:

The heuristic function returns an integer equal to the total number of satisfied clauses in the formula by the following logic:

```
if clauseSatisfied()
        value++;
else
        continue;
return value;
```

Beam search analysis for different beam lengths:

In the following cases, when comparing the number of states explored between algorithms, the values are observed as in the following table. The width of the beam was varied from 1 to 4 for 3 different clauses and the corresponding initial states.

Beam Width	Clause Initial state	States Explored
1	{'a': 0, 'b': 0, 'c': 1, 'd': 1, 'A': 1, 'B': 1, 'C': 0, 'D': 0} [[('d', 'A', 'B'), ('b', 'd', 'A'), ('c', 'B', 'C'), ('c', 'd', 'A'), ('b', 'A', 'B')]]	1
2		1
3		1
4		1
1	{'a': 0, 'b': 0, 'c': 1, 'd': 0, 'A': 1, 'B': 1, 'C': 0, 'D': 1} [[('d', 'C', 'D'), ('a', 'd', 'B'), ('d', 'B', 'C'), ('b', 'A', 'D'), ('A', 'C', 'D')]]	5
2		5
3		5
4		5
1	{'a': 1, 'b': 0, 'c': 0, 'd': 0, 'A': 0, 'B': 1, 'C': 1, 'D': 1} [[('d', 'A', 'D'), ('c', 'A', 'C'), ('a', 'd', 'B'), ('b', 'd', 'A'), ('a', 'b', 'd')]]	9
2		9
3		9
4		13

Tabu search for different values of tabu tenure

The number of explored states is compared between algorithms, the values are observed as in the following table. The taboo tenure was varied from 1 to 4 for 3 different clauses and corresponding initial states.

Tabu Tenure	Clause Initial state	States Explored
1	{'a': 0, 'b': 0, 'c': 1, 'd': 1, 'A': 1, 'B': 1, 'C': 0, 'D': 0} [[('d', 'A', 'B'), ('b', 'd', 'A'), ('c', 'B', 'C'), ('c', 'd', 'A'), ('b', 'A', 'B')]]	1
2		1
3		1
4		1
1	{'a': 0, 'b': 0, 'c': 1, 'd': 1, 'A': 1, 'B': 1, 'C': 0, 'D': 0} [[('b', 'A', 'D'), ('c', 'd', 'C'), ('b', 'c', 'B'), ('b', 'd', 'B'), ('a', 'C', 'D')]]	5
2		7
3		3
4		5
1	{'a': 0, 'b': 0, 'c': 0, 'd': 1, 'A': 1, 'B': 1, 'C': 1, 'D': 0} [[('a', 'd', 'A'), ('a', 'B', 'C'), ('b', 'd', 'D'), ('b', 'A', 'D'), ('b', 'c', 'D')]]	8
2		7
3		5
4		5

Comparison of Variable neighborhood descent, Beam Search, Tabu Search: Nodes explored by each:

With reference to the data mentioned above, we can say for k-SAT problems that the problems need to be compared on different parameters like Tabu Tenure & Beam Width depending on the algorithms. Also, the number of states scanned to reach the target node is compared through different algorithms, namely beam search, taboo search and Variable Neighborhood Descent, and the clear optimal taboo holding is 1 and the optimal beamwidth is 1 for 4-SAT problems. the number of explored states is compared between algorithms, the values are observed as in the following table. We assumed *Beam Width = 2* & *Tabu Tenure = 2.*

Algorithm	Clause Initial state	States Explored
Beam Search	{'a': 0, 'b': 0, 'c': 1, 'd': 1, 'A': 1, 'B': 1, 'C': 0, 'D': 0} [[('d', 'A', 'B'), ('b', 'd', 'A'), ('c', 'B', 'C'), ('c', 'd', 'A'), ('b', 'A', 'B')]]	1
Tabu Search		1
Variable neighborhood descent		1
Beam Search	{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'A': 0, 'B': 0, 'C': 0, 'D': 0} [[('B', 'C', 'D'), ('b', 'A', 'D'), ('a', 'c', 'C'), ('b', 'B', 'D'), ('a', 'd', 'D')]]	5
Tabu Search		5
Variable neighborhood descent		5
Beam Search	{'a': 1, 'b': 0, 'c': 0, 'd': 0, 'A': 0, 'B': 1, 'C': 1, 'D': 1} [[('d', 'A', 'D'), ('c', 'A', 'C'), ('a', 'd', 'B'), ('b', 'd', 'A'), ('a', 'b', 'd')]]	9
Tabu Search		9
Variable neighborhood descent		9

XXX