Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

AY: 2023-24

Name: Divyesh Khunt **Sapid:60009210116** **Batch:D12**

Experiment 4

(Solution Space)

Aim: Find the solution of a SAT (Satisfiability) problem using Variable Neighborhood Descent.

Theory:

The SAT problem

Given a Boolean formula made up of a set of propositional variables V= {a, b, c, d, e, … } each of which can be *true* or *false*, or *1* or *0*, to find an assignment for the variables such that the given formula evaluates to *true* or *1*.

For example, F = ((aV~e) ∧ (eV~c)) ⊃ (~cV~d) can be made *true* by the assignment {a=*true*, c=*true*, d=*false*, e=*false*} amongst others.

Very often *SAT* problems are studied in the *Conjunctive Normal Form (CNF)*. For example, the following formula has five variables (a,b,c,d,e) and six clauses.

(bV~c) ∧ (cV~d) ∧ (~b) ∧ (~aV~e) ∧ (eV~c) ∧ (~cV~d)

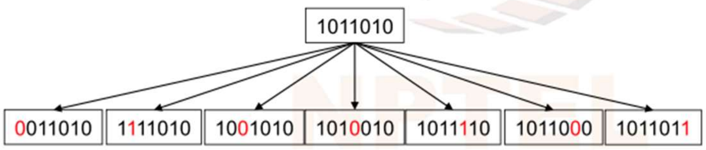
Solution Space Search and Perturbative methods

The Solution Space is the space of candidate solutions.

A local search method generates the neighbours of a candidate by applying some perturbation to the given candidate
MoveGen function = neighbourhood function

A SAT problem with N variables has 2^N candidates
- where each candidate is a N bit string

When N= 7, a *neigbourhood function* may change one bit.



Variable Neighbourhood Descent

```
VariableNeighbourhoodDescent()
    node ← start
    for i ← 1 to n
        do moveGen ← MoveGen(i)
           node ← HillClimbing(node, moveGen)
    return node
```

The algorithm assumes that the function *moveGen* can be passed as a parameter. It assumes that there are *N moveGen* functions sorted according to the density of the neighbourhoods produced.

Lab Assignment to do:

Solve the following SAT problems using VND

1. F = (A V ~B) ^ (B V ~C) ^ (~B) ^ (~C V E) ^ (A V C) ^ (~C V ~D)
2. F = (A V B) ^ (A ^ ~C) ^ (B ^ D) ^ (A V ~E)

1]

```
[1] import random

    def satf1(A, B, C, D, E):
        clause1 = (A or not B)
        clause2 = (B or not C)
        clause3 = (not B)
        clause4 = (not C or E)
        clause5 = (A or C)
        clause6 = (not C or not D)

        return clause1 and clause2 and clause3 and clause4 and clause5 and clause6
```

```
[3] def objective_function(A, B, C, D, E):
        return sum([not satf1(A, B, C, D, E)])

    def random_initial_state():
        return {var: random.choice([True, False]) for var in ['A', 'B', 'C', 'D', 'E']}
```

```
def local_search(state):
    while True:
        neighbors = neighborhood_structure(state)
        best_neighbor = min(neighbors, key=lambda s: objective_function(**s))
        if objective_function(**best_neighbor) >= objective_function(**state):
            break
        state = best_neighbor
    return state
```

```
4] def neighborhood_structure(state):
       neighbors = []
       for var, value in state.items():
           neighbor = state.copy()
           neighbor[var] = not value
           neighbors.append(neighbor)
       return neighbors
```

```python
num_iterations = 80
best_state = None
best_cost = float('inf')
for _ in range(num_iterations):
    current_state = random_initial_state()
    current_state = local_search(current_state)
    current_cost = objective_function(**current_state)
    if current_cost < best_cost:
        best_state = current_state
        best_cost = current_cost

print("Final Assignment after", num_iterations, "Iterations:")
print(best_state)
print("Number of Unsatisfied Clauses:", objective_function(**best_state))
```

```
Final Assignment after 80 Iterations:
{'A': True, 'B': False, 'C': False, 'D': True, 'E': True}
Number of Unsatisfied Clauses: 0
```

2]

```
[6] def satf2(A, B, C, D, E):
        clause1 = (A or B)
        clause2 = (A or not C)
        clause3 = (B or D)
        clause4 = (A or not E)

        return clause1 and clause2 and clause3 and clause4
```

```
num_iterations = 80
best_state = None
best_cost = float('inf')
for _ in range(num_iterations):
    current_state = random_initial_state()
    current_state = local_search(current_state)
    current_cost = objective_function(**current_state)
    if current_cost < best_cost:
        best_state = current_state
        best_cost = current_cost

print("Final Assignment after", num_iterations, "Iterations:")
print(best_state)
print("Number of Unsatisfied Clauses:", objective_function(**best_state))
```

```
Final Assignment after 80 Iterations:
{'A': False, 'B': True, 'C': False, 'D': True, 'E': False}
Number of Unsatisfied Clauses: 0
```