Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

AY: 2023-24

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, $F = ((a \lor \sim e) \land (e \lor \sim c)) \supset (\sim c \lor \sim d)$ 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.

$$(b \lor \sim c) \land (c \lor \sim d) \land (\sim b) \land (\sim a \lor \sim e) \land (e \lor \sim c) \land (\sim c \lor \sim d)$$

Solution Space Search and Perturbative methods

The Solution Space is the space of candidate solutions.

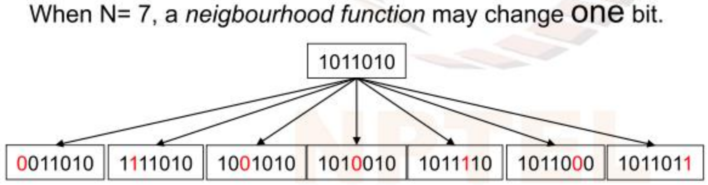
A local search method generates the neighbours of a candidate by applying some perturbation to the given candidate

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)

CODE FOR QUESTION 1 :

```
import random

def sat_like_problem(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

def hill_climbing_for_sat_like(initial_solution, max_iterations):
    current_solution = initial_solution
    for iteration in range(max_iterations):
        neighbors = movegen(current_solution)
        best_neighbor = max(neighbors, key=lambda x: sat_like_problem(*x))
        if sat_like_problem(*best_neighbor):
```

```
            current_solution = best_neighbor
            return current_solution, iteration  # Return the solution and
the iteration
    return current_solution, max_iterations  # If no satisfying solution
is found

def movegen(solution):
    neighbors = []
    for i, var in enumerate(solution):
        flipped_solution = solution[:i] + (not var,) + solution[i+1:]
        neighbors.append(flipped_solution)
    return neighbors

if __name__ == "__main__":
    max_iterations = 1000
    while True:
        initial_solution = [random.choice([True, False]) for _ in
range(5)]
        best_solution, iteration =
hill_climbing_for_sat_like(tuple(initial_solution), max_iterations)
        if sat_like_problem(*best_solution):
            print("Best Solution:", best_solution)
            print("Satisfiability:", sat_like_problem(*best_solution))
            break
```

SOLUTION

```
Best Solution: (True, False, False, False, True)
Satisfiability: True
```

CODE FOR QUESTION 2 :

```
import random

def sat_like_problem(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

def hill_climbing_for_sat_like(initial_solution, max_iterations):
    current_solution = initial_solution
    for iteration in range(max_iterations):
        neighbors = movegen(current_solution)
        best_neighbor = max(neighbors, key=lambda x: sat_like_problem(*x))
        if sat_like_problem(*best_neighbor):
            current_solution = best_neighbor
            return current_solution, iteration  # Return the solution and
the iteration
    return current_solution, max_iterations  # If no satisfying solution
is found

def movegen(solution):
    neighbors = []
    for i, var in enumerate(solution):
        flipped_solution = solution[:i] + (not var,) + solution[i+1:]
        neighbors.append(flipped_solution)
    return neighbors

if __name__ == "__main__":
    max_iterations = 1000
    while True:
        initial_solution = [random.choice([True, False]) for _ in
range(5)]
        best_solution, iteration =
hill_climbing_for_sat_like(tuple(initial_solution), max_iterations)
        if sat_like_problem(*best_solution):
            print("Best Solution:", best_solution)
            print("Satisfiability:", sat_like_problem(*best_solution))
            break
```

SOLUTION
Best Solution: (False, True, False, True, False)
Satisfiability: True