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

Jhanvi Parekh

60009210033

D11

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.

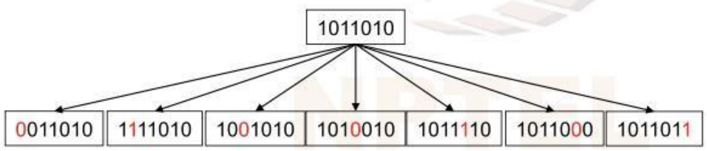
A local search method generates the neighbours of a candidate by applying some perturbation to the given candidate

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)

LINK:
https://colab.research.google.com/drive/1LABTWENznkzpt760H0NcMIefUkk5iFkV?usp=sharing

```python
import copy
import heapq
class Block:
  def __init__(self, name):
    self.name = name
  def __str__(self):
    return self.name

class State:
  def __init__(self, stacks):
    self.stacks = stacks
  def __str__(self):
    return '\n'.join([' '.join([str(block) for block in stack]) for stack in self.stacks])

def heuristic_misplaced_count(state, goal_state):
  count = 0
  min_stack_length = min(len(state.stacks), len(goal_state.stacks))
  for i in range(min_stack_length):
    min_block_length = min(len(state.stacks[i]), len(goal_state.stacks[i]))
    for j in range(min_block_length):
      if state.stacks[i][j] != goal_state.stacks[i][j]:
        count += 1
  return count

def heuristic_sum_of_distances(state, goal_state):
  total_distance = 0
  for i in range(len(state.stacks)):
    for block in state.stacks[i]:
      for j in range(len(goal_state.stacks)):
        if block in goal_state.stacks[j]:
          distance = abs(i - j)
          total_distance += distance
  return total_distance

def movegen(state):
  n = len(state.stacks)
  neighbors = []
  for i in range(n):
    for j in range(n):
      if i != j and state.stacks[i]:
        new_state = copy.deepcopy(state)
        block = new_state.stacks[i].pop()
        new_state.stacks[j].append(block)
        neighbors.append(new_state)
  return neighbors


def hill_climbing(initial_state, goal_state, heuristic_fn):
  current_state = initial_state
  path = [current_state]
  while True:
    neighbors = movegen(current_state)
    if not neighbors:
      break
    neighbor_costs = [heuristic_fn(neighbor, goal_state) for neighbor in neighbors]
    best_neighbor_index = neighbor_costs.index(min(neighbor_costs))
    best_neighbor = neighbors[best_neighbor_index]
    if heuristic_fn(best_neighbor, goal_state) >= heuristic_fn(current_state, goal_state):
      break
    current_state = best_neighbor
    path.append(current_state)
  return path


initial_state = State([['A', 'B', 'C', 'D'], ['E', 'F'], []])
goal_state = State([['A', 'E', 'B', 'C', 'D'], ['F'], []])
print("Initial State:")
print(initial_state)
print("\nGoal State:")
print(goal_state)
print("\nHill Climbing with Heuristic 1:")
path_hill_climbing_1 = hill_climbing(initial_state, goal_state, heuristic_misplaced_count)
if path_hill_climbing_1[-1]==initial_state:
  print("No solution found")
else:
  print("Path")
```

```
  for state in path_hill_climbing_1:
    print(state)

print("\nHill Climbing with Heuristic 2:")
path_hill_climbing_2 = hill_climbing(initial_state, goal_state, heuristic_sum_of_distances)
if path_hill_climbing_2[-1]==initial_state:
  print("No solution found")
else:
  print("Path")
  for state in path_hill_climbing_2:
    print(state)
```

```
Initial State:
A B C D
E F


Goal State:
A E B C D
F


Hill Climbing with Heuristic 1:
Path
A B C D
E F

A B C
E F D

A B
E F D C

A
E F D C B


Hill Climbing with Heuristic 2:
No solution found
```

```
import random

sat_problem_1 = [['A', '!B'], ['B', '!C'], ['!B'], ['!C', 'E'], ['A', 'C'], ['!C', '!D']]
sat_problem_2 = [['A', 'B'], ['A', '!C'], ['B', 'D'], ['A', '!E']]

def is_solution_satisfactory(solution, sat_problem):
    for clause in sat_problem:
        clause_satisfied = False
        for literal in clause:
            if literal[0] == '!':
                negated_literal = literal[1:]
                if negated_literal not in solution:
                    clause_satisfied = True
                    break
            else:
                if literal in solution:
                    clause_satisfied = True
                    break
        if not clause_satisfied:
            return False
    return True

def neighborhood_structure(solution):
    neighbors = []
    literal_to_flip = random.choice(solution)
    neighbor = solution[:]
    if '!' + literal_to_flip in neighbor:
        neighbor.remove('!' + literal_to_flip)
    else:
        neighbor.remove(literal_to_flip)
    neighbors.append(neighbor)
    return neighbors


def variable_neighborhood_descent(sat_problem):
    variables = set(literal for clause in sat_problem for literal in clause)
    current_solution = generate_initial_solution(variables)
```

```
        neighbors = neighborhood_structure(current_solution)

        for neighbor in neighbors:
            if is_solution_satisfactory(neighbor, sat_problem):
                current_solution = neighbor
                break
        else:
            return False

    return True

def generate_initial_solution(variables):
    return [random.choice([var, '!' + var]) for var in variables]

solution_1 = variable_neighborhood_descent(sat_problem_1)

print("Satisfying assignment for SAT problem 1 found:", solution_1)

solution_2 = variable_neighborhood_descent(sat_problem_2)

print("Satisfying assignment for SAT problem 2 found:", solution_2)
```

```
Satisfying assignment for SAT problem 1 found: False
Satisfying assignment for SAT problem 2 found: False
```

```
import random

def sat_formula(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 objective_function(A, B, C, D, E):
    return sum([not sat_formula(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


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

current_state = random_initial_state()
best_state = local_search(current_state)

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

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

problem 1

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

```
# @title 1. F = (A V ~B) ^ (B V ~  C) ^ (~B) ^ (~C V E) ^ (A V C) ^ (~C V ~D)
import random

def sat_formula(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 objective_function(A, B, C, D, E):
    return sum([not sat_formula(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

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

# Specify the number of iterations
num_iterations = 100

# Run local search for a fixed number of iterations
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 the final assignment after the specified number of iterations
print("Final Assignment after", num_iterations, "Iterations:")
print(best_state)
print("Number of Unsatisfied Clauses:", objective_function(**best_state))
```

```
Final Assignment after 100 Iterations:
{'A': True, 'B': False, 'C': False, 'D': True, 'E': False}
Number of Unsatisfied Clauses: 0
```

problem 2

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

```
# @title 2. F = ( A V B) ^ ( A ^ ~C) ^ ( B ^ D ) ^ ( A V ~E)
import random

def sat_formula(A, B, C, D, E):
    clause1 = (A or B)
```

```
    clause2 = (A and not C)
    clause3 = (B and D)
    clause4 = (A or (not E))

    return clause1 and clause2 and clause3 and clause4

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

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

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

# Specify the number of iterations
num_iterations = 100

# Run local search for a fixed number of iterations
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 the final assignment after the specified number of iterations
print("Final Assignment after", num_iterations, "Iterations:")
print(best_state)
print("Number of Unsatisfied Clauses:", objective_function(**best_state))
```

```
Final Assignment after 100 Iterations:
{'A': True, 'B': True, 'C': False, 'D': True, 'E': True}
Number of Unsatisfied Clauses: 0
```