

Week-3: Generating and Solving Uniform Random 3-SAT

Chetankumar kamani(20251603005), Sakariya Devraj (20251603006), Divyesh Dodiya (20251603007),
M.Tech Sem-1 IIIT Vadodara

I. PROBLEM STATEMENT

Write programs to generate uniform random k -SAT instances and, for $k=3$, solve a set of random 3-SAT problems for different (m, n) combinations. Compare **Hill-Climbing**, **Beam-Search** (beam widths 3 and 4), and **Variable-Neighborhood Descent** (three neighborhoods). Use two heuristic functions and compare them with respect to *penetration* (fraction of runs solved).

A. Uniform Random k -SAT (Fixed Clause Length Model)

Given k, m, n , generate m clauses. Each clause picks k distinct variables from $\{x_1, \dots, x_n\}$ uniformly at random; each literal is negated with probability $1/2$. For $k=3$ we obtain uniform random 3-SAT.

II. PYTHON CODE

A. A. Simple k -SAT Generator

```
import random

def generate_kSAT(k, m, n):
    clauses = []
    for _ in range(m):
        # pick k distinct variables
        vars_chosen = random.sample(range(1, n+1), k)
        clause = []
        for v in vars_chosen:
            if random.choice([True, False]):
                clause.append(f"x{v}")
            else:
                clause.append(f"~x{v}")
        clauses.append(clause)
    return clauses

# Example usage
k = 3 # length of each clause
m = 5 # number of clauses
n = 4 # number of variables

formula = generate_kSAT(k, m, n)

print("Random k-SAT formula:")
for i, clause in enumerate(formula, 1):
    print(f"C{i}: ({' '.join(clause)})")

%% Example function signature:
%% generate_kSAT(k, m, n) -> list[list[str]]
```

B. B. 3-SAT (HC, Beam=3/4, VND) with Two Heuristics

```
import random

# 3-SAT
def generate_3sat(m, n):
    """Generate random 3-SAT instance with m
    clauses, n variables"""
    clauses = []
    for _ in range(m):
        vars3 = random.sample(range(1, n+1), 3)
        clause = []
        for v in vars3:
            if random.choice([True, False]):
                clause.append(v)
            else:
                clause.append(~v)
        clauses.append(clause)
    return clauses

def evaluate(clauses, assignment):
    """Number of satisfied clauses"""
    count = 0
    for c in clauses:
        ok = False
        for lit in c:
            v = abs(lit)
            val = assignment[v]
            if (lit > 0 and val) or (lit < 0 and not val):
                ok = True
        if ok: count += 1
    return count

# two heuristics
def h1(clauses, A): return evaluate(clauses, A)
def h2(clauses, A): return 2*evaluate(clauses, A) - len(clauses)

# hill climbing
def hill_climbing(clauses, n, heuristic, steps=1000):
    A = [None] * n
    for _ in range(steps):
        if evaluate(clauses, A) == len(clauses):
            return True

        best_score, best_var = heuristic(clauses, A), None
        for v in range(1, n+1):
            A[v] = not A[v]
            score = heuristic(clauses, A)
            A[v] = not A[v]
            if score > best_score:
                best_score, best_var = score, v
        if best_var is None:
            v = random.randint(1, n)
            A[v] = not A[v]
        else:
            A[best_var] = not A[best_var]
    return False
```

```

def beam_search(clauses, n, heuristic, beam_width=3, steps=200):
    beam = []
    for _ in range(beam_width):
        A = [None] + [random.choice([False, True]) for _ in range(n)]
        beam.append(A)
    for _ in range(steps):
        new_beam = []
        for A in beam:
            if evaluate(clauses, A) == len(clauses): return True
            for v in range(1, n+1):
                B = A.copy()
                B[v] = not B[v]
                new_beam.append(B)
        new_beam.sort(key=lambda x: heuristic(clauses, x), reverse=True)
        beam = new_beam[:beam_width]
    return False

def vnd(clauses, n, heuristic, steps=1000):
    A = [None] + [random.choice([False, True]) for _ in range(n)]
    for _ in range(steps):
        if evaluate(clauses, A) == len(clauses): return True
        improved = False
        for v in range(1, n+1):
            A[v] = not A[v]
            if heuristic(clauses, A) > heuristic(clauses, A):
                improved = True
            A[v] = not A[v]
        if not improved:
            v = random.randint(1, n)
            A[v] = not A[v]
    return False

n = 6 # variables
m = 15 # clauses
clauses = generate_3sat(m, n)
print("Generated 3-SAT clauses:", clauses)

for hname, hfun in [("h1", h1), ("h2", h2)]:
    print(f"\nUsing heuristic {hname}:")
    print("Hill-Climbing:", hill_climbing(clauses, n, hfun))
    print("Beam Search (width=3):", beam_search(clauses, n, hfun, beam_width=3))
    print("Beam Search (width=4):", beam_search(clauses, n, hfun, beam_width=4))
    print("VND:", vnd(clauses, n, hfun))

%% - instance generator generate_3sat(m, n)
%% - evaluate(), heuristics h1 and h2
%% - hill_climbing(), beam_search(beam_width=3/4), vnd()
%% - a small main that prints solved/unsolved for both heuristics

```

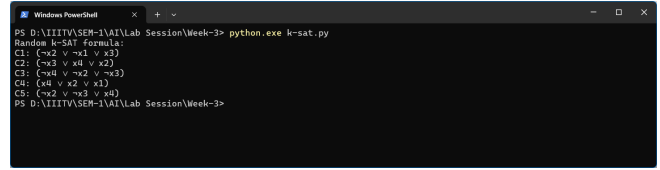
III. HOW TO RUN

- Generate k -SAT or run 3-SAT solvers from terminal:

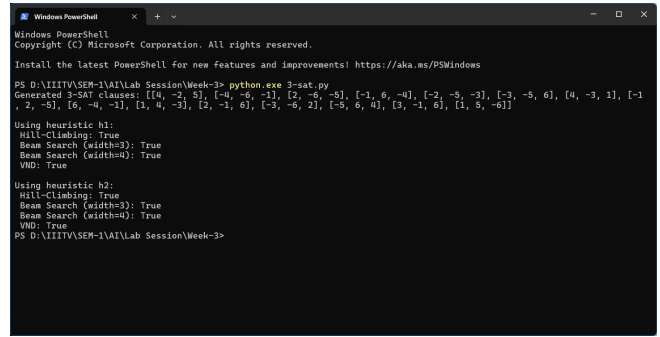
```
python k-sat.py
python 3-sat.py
```

IV. SCREENSHOTS OF RESULTS

A. k -sat.py



B. 3-sat.py



V. CODE AVAILABILITY

The complete source code is available at: GitHub Repository (CS659 – AI Laboratory).

GitHub Repository:

<https://github.com/ChetanKamani/CS659-AI-Lab>