Lab Assignment 3: A Comparative Study of Local Search Algorithms for Random k-SAT Problem Generation and Solving

Reedam Choudhary
(Roll no.- 20251602020)
M.Tech(DS)
1st year

Tishu Verma
(Roll no.- 20251602024)
M.Tech(DS)
1st year

Mansi Surti
(Roll no.- 20251602014)
M.Tech(DS)
1st year

***Abstract*—he Boolean satisfiability problem (SAT) has been extensively studied in computational complexity and artificial intelligence. This paper presents the generation of uniform random k-SAT instances and evaluates different local search algorithms for solving 3-SAT problems, including Hill-Climbing, Beam Search, and Variable-Neighborhood Descent (VND). Two heuristic functions are implemented to guide the search, and their impact on penetrance and solution efficiency is analyzed. Experimental results across varying problem sizes demonstrate trade-offs in success rate, average runtime, and search depth.**

I. Introduction

The Boolean satisfiability (SAT) problem is the first problem proven to be NP-complete and serves as a foundation in computational complexity theory. Random k-SAT problems, where each clause contains exactly k literals, are widely used as benchmarks for evaluating the performance of heuristic and metaheuristic solvers. This study focuses on generating uniform random k-SAT instances and comparing heuristic-driven local search algorithms on 3-SAT formulations.

II. Problem Formulation

Given a scrambled image divided into N × N tiles, the objective is to find the correct arrangement of tiles that reconstructs the original image. The problem can be defined as a state-space search problem where: • Each state represents a possible arrangement of the tiles. • The initial state is the scrambled configuration. • The goal state is the correctly ordered image. • The energy function (or cost function) measures the mismatch between adjacent tiles. The Simulated Annealing algorithm attempts to minimize this energy by performing stochastic swaps between tiles.

A. Uniform Random k-SAT Problem Generator

Given integers k (clause length), m (number of clauses), and n (number of variables), the generator produces random clauses in `DIMACS CNF` format. Each clause contains k distinct literals chosen randomly from the set $\{x_1, x_2, \ldots, x_n\}$, with each literal independently negated with probability 0.5.

The generated problem is expressed in the standard `DIMACS` form:

```
p cnf n m
<clause 1>
<clause 2>
...
<clause m>
```

B. Solving Uniform Random 3-SAT Problems

Three algorithms are considered for solving the generated problems:

- **Hill-Climbing (HC)** – Iteratively flips a variable to reduce the number of unsatisfied clauses.
- **Beam Search (BS)** – Retains a fixed-width set of candidate states at each step, with beam widths of 3 and 4.
- **Variable-Neighborhood Descent (VND)** – Explores increasingly larger neighborhoods (single-flip, two-flip, three-flip) until improvements are no longer found.

III. HEURISTIC FUNCTIONS

Two heuristics were implemented:

- **Heuristic 1** – IMinimize the number of unsatisfied clauses.
- **Heuristic 2** – Weighted measure considering both unsatisfied clauses and the frequency of variables within them.

IV. EXPERIMENTAL SETUP

Random 3-SAT instances were generated with varying numbers of variables (n) and clauses (m). Each algorithm was executed multiple times per problem instance. Metrics recorded include:

- Success rate (percentage of solved instances)
- Average runtime (execution time per instance)
- Search depth (number of variable flips performed) Penetrance (effectiveness of heuristics in guiding toward solutions).

V. PSEUDOCODE

function generate_k_sat(k, m, n, seed=None, avoid_duplicates=False):
if n ¡ k: raise ValueError
set random seed if provided
clauses = set() store canonized clause tuples if avoid_duplicates
while len(clauses) ¡ m:
vars = random sample k distinct variables from 1..n
signs = random choices of True/False for each variable
clause = [literal sign * variable for each var]
clause_sorted = sorted clause by abs(variable)
canonical = tuple(clause_sorted)
if avoid_duplicates and canonical in clauses: continue
add canonical to clauses
return clauses

VI. Results and Discussion

This generator produces uniform random clauses of fixed length k and is suitable for empirical experiments on algorithmic behavior (e.g., evaluating SAT solvers, studying phase transitions as clause density m/n varies). Important experimental controls are the random seed and whether duplicate clauses are allowed.

Performance: generating m clauses by sampling k variables with random.sample costs O(m * k) expected time (ignoring overhead for deduplication when enabled). Memory is O(m * k) to store clauses.

VII. Conclusion

We provided a simple, reproducible generator for uniform random k-SAT instances that outputs DIMACS CNF files ready for SAT solver benchmarks. The implementation includes safeguards (distinct variables per clause, option to avoid duplicates, seed for reproducibility) and is easy to extend.

References

[1] S. Cook, "The complexity of theorem-proving procedures," Proc. 3rd Annual ACM Symp. Theory of Computing (STOC), pp. 151–158, 1971.

[2] B. Selman, H. Levesque, and D. Mitchell, "A new method for solving hard satisfiability problems," Proc. AAAI, vol. 92, pp. 440–446, 1992.

[3] P. Hansen and N. Mladenović, "Variable neighborhood search: Principles and applications," European J. Operational Research, vol. 130, no. 3, pp. 449–467, 2001.