# Solving the Minimum Hitting Set Problem using Genetic Algorithms

Drumia Petru-Sebastian

## 1. Problem Description

The **Minimum Hitting Set (MHS)** problem is a classical optimization problem. Given:

- A finite universal set $U = \{1, 2, ..., n\}$
- A collection of subsets $S = \{S_1, S_2, ..., S_m\}$ where each $S_i \subseteq U$

The goal is to find the smallest subset $H \subseteq U$ such that $H$ intersects every subset in $S$, i.e., $H \cap S_i \neq \emptyset$ for all $i \in \{1, ..., m\}$. This subset $H$ is known as a *hitting set*.
Example: In the below example we can see that:

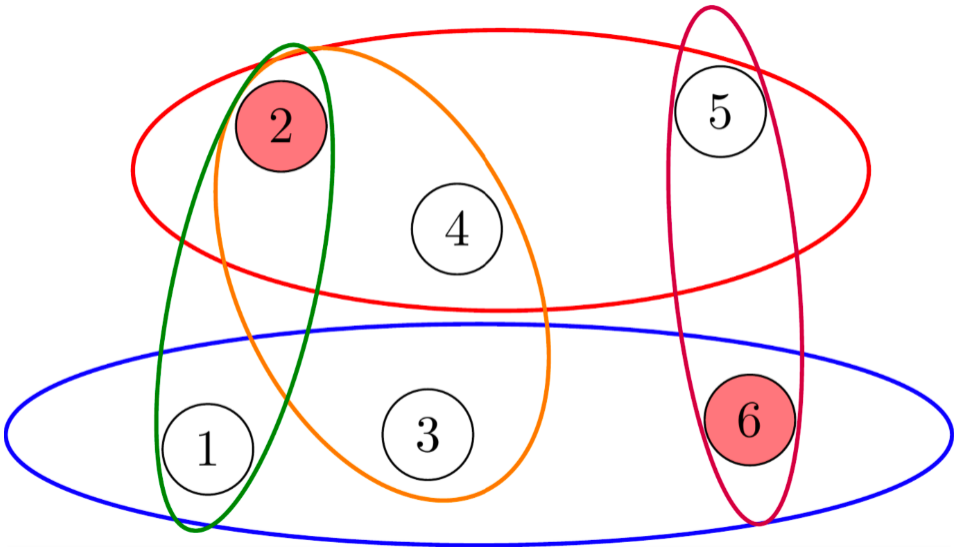


Figure 1: Visual representation of hitting set.

$$U = \{1, 2, 3, 4, 5, 6\}$$

$$S = \{\{1, 2\}, \{2, 3, 4\}, \{2, 4, 5\}, \{5, 6\}, \{1, 3, 6\}\}$$

The minimum hitting set for this example is $H = \{2, 6\}$, as seen by the red nodes.

## 2. Heuristic Track - Genetic Algorithm Overview

We employ a **Genetic Algorithm (GA)** to approximate a solution to the MHS problem. A GA is a population-based metaheuristic inspired by natural selection. It evolves a population of candidate solutions over iterations using selection, crossover, and mutation.

### Representation

Each individual is a binary string of length $n$ (size of $U$). A 1 at position $i$ indicates that element $i$ is included in the hitting set, while a 0 indicates that the element is not included in the hitting set. For example the individual 0011001 has the following nodes included in the hitting set: $\{3, 4, 7\}$ while 0100100 has the nodes $\{2, 5\}$ included.

### Initial Population

We generate an initial population of 150 individuals randomly, therefore each individual is an array of length $U$ with values 1 or 0.

### Fitness Evaluation

We use an objective function in order to calculate the fitness value. The *objective function* of an individual is the number of elements selected (number of 1s). The logic behind that is the fact that we try to find the minimum hitting set, so the smaller the value of the objective function the better.

$$\text{obj}(111) = 3$$
$$\text{obj}(101) = 2$$
$$\text{obj}(001) = 1$$

Certain individuals might not actually be hitting sets, because of the random initialization and because of genetic operators. Before calculating the fitness value, we make sure that each individual is a valid hitting set. We do that by **repairing** it, if it is not a valid hitting set, by greedily adding nodes with the highest frequency in $S$ until all subsets are hit.

Fitness is computed by normalizing and then inverting the objective values:

$$\text{fitness} = 1 - \frac{\text{objective} - \min}{\max - \min}$$

To maintain diversity and discourage elitism, fitness of the top 100 elite individuals is modified:

- If fitness > mean(fitness): fitness = fitness - mean(fitness).
- Else: fitness = 0.

### Selection

*Tournament selection* is used with group size $k = 3$. For each individual, $k$ candidates are selected at random and the one with the best fitness is chosen.

### Crossover and Mutation

- **Crossover**: Single-point crossover is applied between pairs of individuals.
- **Mutation**: Each bit has a small chance, $(\frac{1}{n})$, of being flipped, where $n$ is the number of nodes.

### Elitism and Iteration

At each generation:

1. Top 100 individuals are preserved. These 100 go directly to the next iteration without applying genetic changes to them.
2. The next 50 individuals in the population are created through selection, crossover, and mutation.

The process repeats for 5000 generations.

### Improved algorithm

- **Selection**: The selection method is modified to **rank selection**. Here the individuals are sorted by their fitness and each individual has a chance of being selected (even those with low fitness!), a probability being assigned to everyone, unlike tournament selection.
- **Local Search**: At every 100 iterations, we perform a local search on 30 percent of the population, choosing individuals randomly. By local search, we iterate through the bits of an individual, and we just set them to 0, one by one. If the individual is still a hitting set, we keep him modified. This change takes place after genetic operators.
- **Greedy initialization**: When initializing the population, instead of having the same probability of picking either 0 or 1 for a specific bit, the probability of 0 being picked is going to be much higher. This change is going to affect 30 percent of the population. This way we are already starting with individuals that have fewer nodes selected in their hitting set, increasing the chance of finding the minimum hitting set.

## 3. Heuristic Track (GAs) - Results

A series of experiments were concluded to determine the efficacy of the algorithm, the original one and the modified version of it.
All experiments were performed 30 times. The metrics that were saved to determine the efficacy of the algorithms are:

### Metrics:

- time taken, in seconds;
- the number of nodes in the hitting set, the minimum and average value over 30 runs;
- standard deviation for the number of nodes in the hitting set;
- 95 percent confidence interval for the cardinality of the minimum hitting set.

**Tables:**

| Original GA results | | | | | | |
|---|---|---|---|---|---|---|
| Datasets | Opt HS | Min HS | Avg HS | Std Dev | C.I. | Time |
| bremen 20 | 9 | 9 | 9.93 | 0.21 | (9.81, 10.02) | 27.92 |
| bremen 50 | 17 | 18 | 18.53 | 0.62 | (18.29, 18.76) | 50.11 |
| bremen 100 | 29 | 31 | 32.63 | 0.85 | (32.31, 32.95) | 89.77 |
| bremen 150 | 42 | 43 | 46.63 | 1.47 | (46.08, 47.18) | 142.16 |
| bremen 200 | 57 | 61 | 64.03 | 1.29 | (63.54, 64.51) | 184.82 |
| bremen 250 | 74 | 81 | 85.03 | 1.97 | (84.29, 85.77) | 162.45 |
| bremen 300 | 84 | 94 | 96.3 | 1.53 | (95.72, 96.87) | 190.73 |

| Improved GA results | | | | | | |
|---|---|---|---|---|---|---|
| Datasets | Opt HS | Min HS | Avg HS | Std Dev | C.I. | Time |
| bremen 20 | 9 | 9 | 9.83 | 0.37 | (9.69, 9.97) | 30.67 |
| bremen 50 | 17 | 18 | 18 | 0.0 | (18.0, 18.0) | 45.65 |
| bremen 100 | 29 | 30 | 30.86 | 0.62 | (30.63, 31.10) | 76.99 |
| bremen 150 | 42 | 43 | 44.3 | 0.74 | (44.02, 44.57) | 108.73 |
| bremen 200 | 57 | 58 | 59.76 | 0.77 | (59.47, 60.05) | 199.42 |
| bremen 250 | 74 | 78 | 79.93 | 1.14 | (79.50, 80.36) | 186.86 |
| bremen 300 | 84 | 87 | 90.6 | 1.73 | (89.95, 91.24) | 315.99 |

## Graphs:

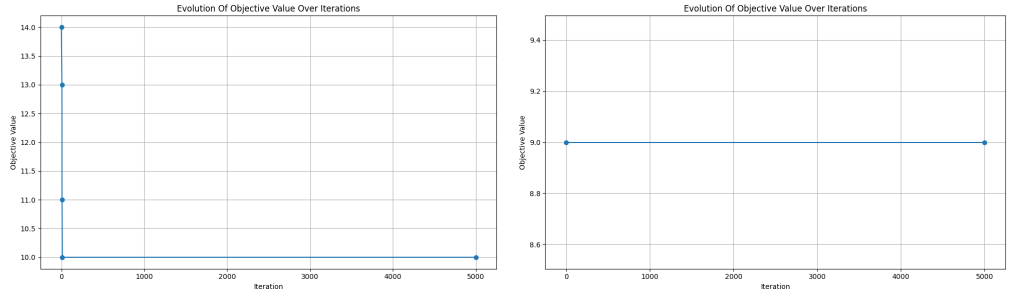


Figure 2: Evolution of minimum hitting set for *original GA*(left) and *improved GA*(right) on *bremen20* instance, across iterations.

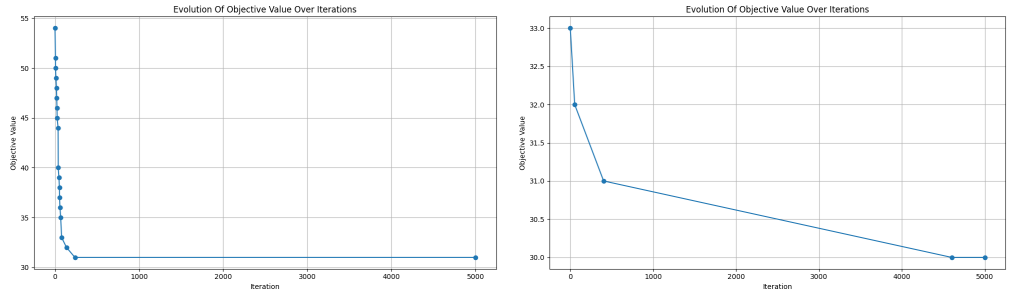


Figure 3: Evolution of minimum hitting set for *original GA*(left) and *improved GA*(right) on *bremen100* instance, across iterations.

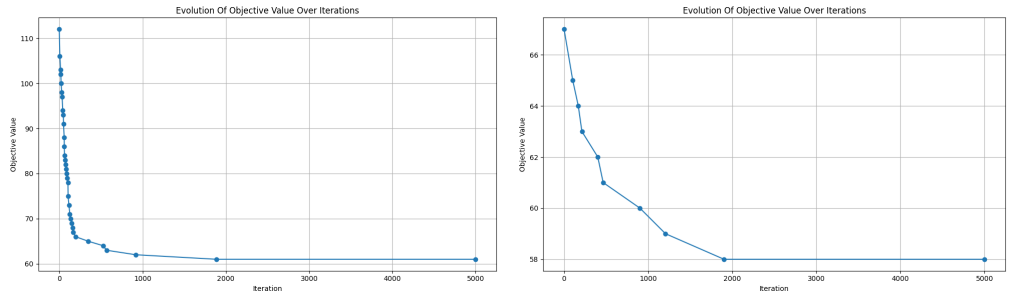


Figure 4: Evolution of minimum hitting set for *original GA*(left) and *improved GA*(right) on *bremen200* instance, across iterations.

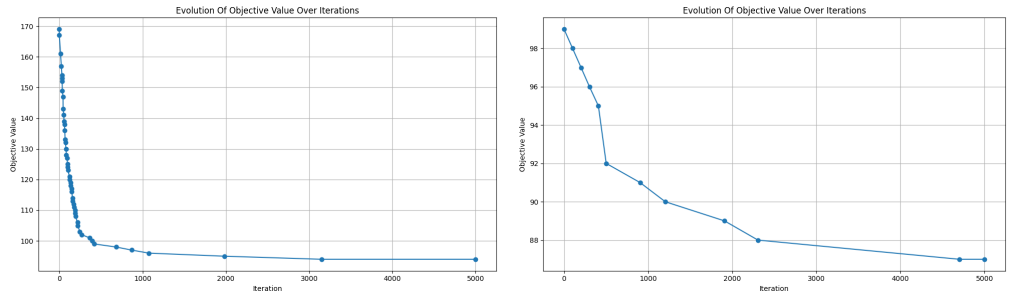


Figure 5: Evolution of minimum hitting set for *original GA*(left) and *improved GA*(right) on *bremen300* instance, across iterations.

We can observe that all taken times are mostly under 300, only the dataset *bremen300*, for the improved algorithm, taking a bit longer. The algorithm gets close to the best solution, even finds it for the *bremen20* dataset. If we compare the two tables with both algorithms, we can see that the improved GA truly finds better solutions, but it usually needs more time for the bigger problem instances. Judging by the visual representation of the evolution of the best solution over the 5000 iterations, we can see that the algorithm usually needs half of those iterations to converge.

# 4. Exact Track - Branch And Bound Algorithm Overview

**Branch and Bound (B&B)** is a general algorithm for finding optimal solutions in combinatorial and discrete optimization problems. It systematically explores the space of candidate solutions by branching into subproblems and pruning those that cannot produce a better solution than the best found so far (using a bound).

### The Branching Function

1. **Base Case Check:**
   - If a complete and valid solution is found, compare it to the current best and update if better.
   - If no more decisions can be made, or the lower bound is worse than the current best, return (prune).
2. **Select a Branching Variable:** Choose a variable (or element) to branch on. The vertices are processed in descending order of degree (so vertices which belong to more sets are processed first).
3. **Create Subproblems:**
   - One where the variable is *included*.
   - One where the variable is *excluded*.
4. **Solve recursively**

The algorithm maintains:

- A partial solution

- The remaining decision variables
- A **lower bound** and an **upper bound** (greedy) to determine whether to prune the branch.

The lower bounds used in the reference paper[3] are:

- **Max-degree bound:** Assume all vertices have max degree to bound the number of picked vertices ($|F|/d_{max}$), where $d_i$ is the degree of a vertex and $F$ is the set of sets/edges).
- **Sum-degree bound:** Use the number of vertices that, sorted by highest degree and summed, go over the amount of sets ($\sum_{sorted} d_i \geq |F|$) .
- **Efficiency bound:** Compute a solution score as $\sum_F \sum_{v \in F} 1/d_v$ for $v$ in all picked vertices in the partial solution.
- **Packing bound:** Use the largest collection of disjoints sets as the minimum amount of vertices to pick.
- **Sum over packing bound:** Use both the sum-degree and packing bounds for a better bound.

### CPLEX Branch and Bound

**IBM CPLEX** uses an advanced version of Branch and Bound, exploring the search space using **Mixed-Integer Programming (MIP)**.

#### How It Works

1. **Solve LP Relaxation:** Relax the integer constraints and solve the resulting Linear Program (LP).
2. **Check Integer Feasibility:**
   - If the solution is integer-feasible, it's a valid MIP solution.
   - Otherwise, identify a variable with a fractional value to branch on.
3. **Branch:** Create two new LP subproblems:
   - One where the chosen variable is forced to be less than or equal to its floor.
   - One where it is greater than or equal to its ceiling.
4. **Repeat:** Solve each subproblem and prune based on bounds and feasibility.

### Branch and Cut

**Branch-and-Cut** is a version of Branch and Bound, used in solving MIPs more efficiently.

#### Key Features

- **Cuts:** Additional constraints (called cuts) are generated and added dynamically during the search. These cuts are valid inequalities that eliminate infeasible (typically fractional) solutions without removing any feasible integer ones.
- **Tighter Bounds:** Cuts strengthen the LP relaxation, often reducing the number of branches required.

# 5. Exact Track - Results

### CPlex

We ran the CPlex algorithm configured to force the use of a branch and cut algorithm instead of a dynamic one. We ran a grid search for the following parameters of the cplex model:

- **LP Method:** [primal, dual] Which form of the LP to solve.
- **Gomory Cuts:** [auto, minimal, moderate, agressive] Apply rounding to the coefficients of variables with non-integer solutions.
- **Lifproj Cuts:** [auto, minimal, moderate, agressive] Find new constraints which are valid for both branches, while being invalid for the root node.
- **Mixed Integer Rounding Cuts:** [auto, minimal, moderate, agressive] Apply rounding to the coefficients and right hand side of constraints for variables with non-integer solutions.
- **Zero Half Cuts:** [auto, minimal, moderate, agressive] Combine multiple constraints with integer coefficients by summing them and rounding down the right hand side.

The resulting configurations performed similarly, except for those which used aggressive Liftproj cuts.

| Cplex branch and cut results | | | | | |
|---|---|---|---|---|---|
| Datasets | Opt HS | Min time | Max time | Avg time | Std time |
| bremen 20 | 9 | 0.000500 | 0.005267 | 0.002399 | 0.000856 |
| bremen 50 | 17 | 0.009367 | 0.017733 | 0.014156 | 0.001894 |
| bremen 100 | 29 | 0.164633 | 0.459333 | 0.249033 | 0.103218 |
| bremen 150 | 42 | 0.031267 | 0.070333 | 0.051027 | 0.011138 |
| bremen 200 | 57 | 0.246300 | 0.660900 | 0.374004 | 0.151481 |
| bremen 250 | 74 | 0.622367 | 1.681767 | 1.180976 | 0.370687 |
| bremen 300 | 84 | 0.456233 | 1.337467 | 0.790320 | 0.337318 |

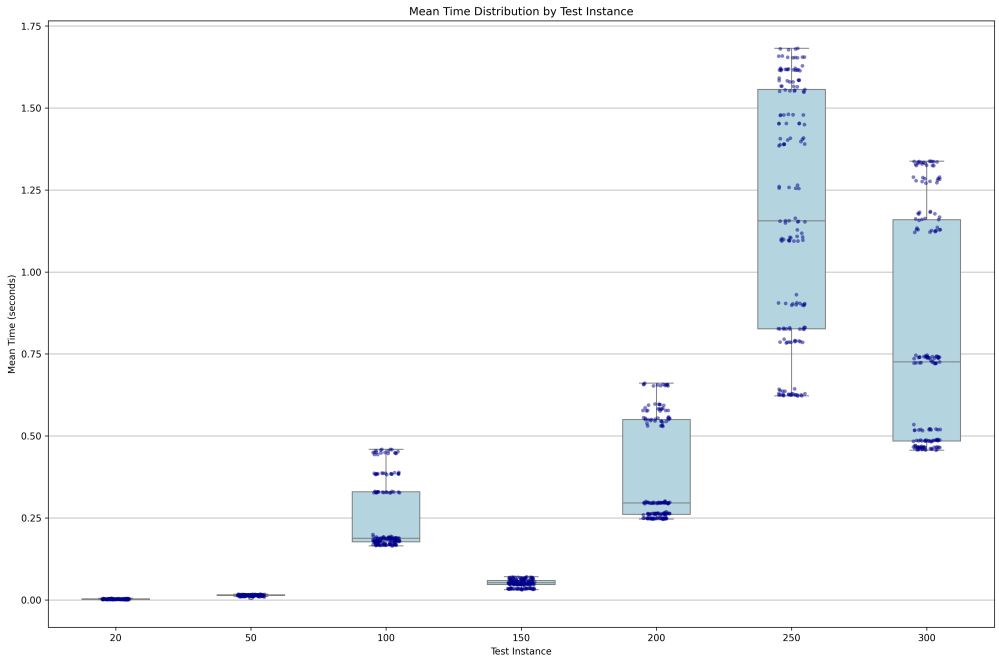


Figure 6: Box plots of all configurations by test instance

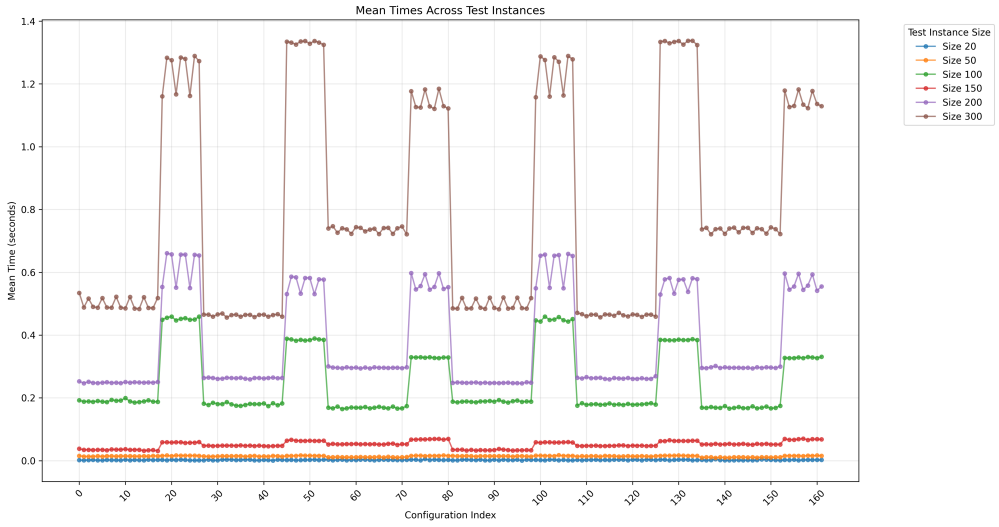


Figure 7: Average time by configuration, for all problems. Spikes indicate configurations with agressive lliftproj cuts.

**Rust Branch and Bound Algorithm**

In addition, we used an implementation of branch and bound used in [3], written in the rust programming language. This method aims to solve the combinatorial problem of which vertices to pick, without using integer relaxation. Here, we varied the number of bounds computed per node and compared them against the final configuration used in the paper. Configuration C1 uses all provided bounds, while C2 disables the local search (for improving the packing bound) and the sum degree bound, while only using greedy for initialization instead of for each node.

| Branch and bound | | | | | | |
|---|---|---|---|---|---|---|
| Datasets | Opt HS | Best HS | Avg time (C1) | Std time (C1) | Avg time (C2) | Std time (C2) |
| bremen 20 | 9 | 9 | 0.006428 | 0.000526 | 0.004075 | 0.000832 |
| bremen 50 | 17 | 17 | 0.132550 | 0.007273 | 0.165850 | 0.005128 |
| bremen 100 | 29 | 29 | 57.848613 | 0.267232 | 55.040053 | 0.171625 |
| bremen 150 | 42 | 42 | 19.971120 | 0.102979 | 43.482252 | 0.182666 |
| bremen 200 | 57 | [57] | 107.505261 | 0.181327 | 418.421873 | 0.563841 |
| bremen 250 | 74 | [77] | 269.271170 | 0.159959 | 145.017631 | 0.138797 |
| bremen 300 | 84 | [86] | 19.170108 | 0.040512 | 11.592157 | 0.022253 |

Table 1: For bigger problems the search tree was not explored completely. In that case, the end search time denotes the time the best solution was found (end time was capped at 10 minutes).

# 6. Bibliography

# References


[1] Cendic, Bojana Lazovic, *A genetic algorithm for the minimum hitting set*, Scientific Publications of the State University of Novi Pazar 6.2 (2014): 107.

[2] Li, Lin, and Jiang Yunfei, *Computing minimal hitting sets with genetic algorithm*, Algorithmica 32.1 (2002): 95-106.

[3] Thomas Bläsius, Tobias Friedrich, David Stangl, Christopher Weyand, *An Efficient Branch-and-Bound Solver for Hitting Set*, `https://arxiv.org/abs/2110.11697`.

[4] *PACE 2025 - Hitting Set Problem*, `https://pacechallenge.org/2025/hs/`.

[5] *Hitting Set Problem Instances*, `https://github.com/MarioGrobler/hs_verifier/tree/main/Hitting%20Set%20Verifier/src/test/resources/testset`.