Minimum Set Cover Problem

Alexandra Wu, Ke Xin Chong, Wei Hong Low, Yuanting Fan
Georgia Institute of Technology
CSE 6140 Spring 2025 Project
{xwu393,kchong43,wlow7,yfan393}@gatech.edu

ABSTRACT

The Minimum Set Cover problem is a classical NP-hard problem with widespread applications in optimization and resource management. We evaluate three algorithmic approaches—Branch-and-Bound, Greedy Set Cover, and two local search variants (Hill Climbing and Simulated Annealing)—on benchmark instances. While the exact Branch-and-Bound method guarantees optimality, it struggles on large inputs under a 900-second cutoff. The Greedy algorithm is fast but less accurate, whereas local search methods offer better trade-offs between speed and solution quality. Our study outlines the practical strengths and weaknesses of each approach under resource constraints.

KEYWORDS: Set Cover, NP-hard, Branch and Bound, Greedy Algorithm, Simulated Annealing, Approximation Algorithm, Heuristic Search

1 INTRODUCTION

The Minimum Set Cover Problem is a fundamental NP-hard problem in combinatorial optimization with numerous applications in areas such as network design, resource allocation, bioinformatics, and data compression. Given a universe of elements and a collection of subsets, the objective is to select the smallest number of subsets whose union covers the entire universe. Due to its computational intractability, a wide variety of exact, approximate, and heuristic algorithms have been proposed to tackle this problem under different constraints.

In this project, we implemented and evaluated four algorithms to solve the MSCP: an exact method using Branch-and-Bound (BnB), a classical greedy approximation algorithm, and two local search techniques—Hill Climbing and Simulated Annealing. Each algorithm was tested across a benchmark suite of both small and large instances, and their performance was evaluated based on solution quality, runtime, and scalability.

Our results demonstrate a clear trade-off between optimality and efficiency. The Branch-and-Bound algorithm guarantees exact solutions on small instances but fails to scale under a fixed cutoff time of 900 seconds. The greedy algorithm consistently produces fast, near-optimal solutions, especially effective on dense instances. The two local search algorithms strike a middle ground: while they lack formal approximation guarantees, they are capable of improving upon the greedy baseline in many cases, especially for large and complex instances.

This study not only benchmarks diverse algorithmic strategies for the MSCP but also highlights practical limitations and strengths under real-world computational constraints.

2 PROBLEM DEFINITION

Given a universe $U = \{x_1, x_2, \ldots, x_n\}$ and a collection of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, where each $S_i \subseteq U$, the **Minimum Set Cover Problem** asks for the smallest possible subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that:

$$\bigcup_{\mathcal{S}_i \in \mathcal{T}} \mathcal{S}_i = U$$

The selected subcollection $\mathcal{S}_i$ is then called a *set cover* of U. If it has the fewest possible subsets, it is called a *minimum set cover.*

Example: Consider the universe $U = \{1, 2, 3, 4, 5\}$ and the following family of subsets:

$$S_1 = \{1, 2\}$$
$$S_2 = \{2, 3, 4\}$$
$$S_3 = \{4, 5\}$$
$$S_4 = \{1, 5\}$$

A minimum set cover is $\{S_2, S_4\}$, as their union covers all elements in U.

The decision version of minimum set cover is NP-complete shown in 1972. The optimization/search version of minimum set cover is NP-hard[1].

3 RELATED WORK

The most commonly used algorithm for the Minimum Set Cover Problem (MSCP) is the Greedy Algorithm. However, this project also explores other approaches: Branch-and-Bound and local search variants. Across datasets of varying sizes, these four algorithms exhibit different characteristics.

Branch-and-Bound (BnB) is a classical exact algorithmic framework frequently applied to NP-hard combinatorial optimization problems such as the *Minimum Set Cover*. Several studies have investigated BnB-based strategies to reduce the exponential search space by employing tight bounding techniques and intelligent node-ordering heuristics. Traditional BnB approaches for Set Cover rely on greedy heuristics to establish initial upper bounds and leverage coverage-based estimations for lower bounds to effectively prune subtrees of the solution space. These methods have been shown to perform well on small- to medium-sized instances, particularly when augmented with problem-specific heuristics or hybridized with approximation strategies. Notably, the effectiveness of BnB in solving Set Cover has also served as a benchmark for evaluating the quality of approximate and heuristic algorithms in related literature[2]. In our implementation, we extend this foundation by incorporating a best-first search strategy with a gain-adjusted priority function, and explicitly managing memory with a queue size cap. These enhancements improve the scalability of the algorithm

on larger instances while maintaining exactness, distinguishing our approach from traditional recursive or depth-first BnB variants.

Chvátal's **greedy heuristic** [3], a widely used baseline for set cover, achieves a logarithmic approximation ratio of $H(n)$ by iteratively selecting the subset that covers the most uncovered elements. Its simplicity and theoretical guarantees have made it a standard in both research and practice. Alternative methods—such as LP relaxations with randomized rounding, local search, and metaheuristics like genetic algorithms—often yield better empirical results but at the cost of increased complexity and parameter tuning. Our implementation extends Chvátal's approach with a size-based pre-sorting heuristic: subsets are sorted by size in descending order before the greedy loop begins. This prioritization can reduce iterations by favoring broader early coverage, improving empirical performance without affecting theoretical bounds. The result is a lightweight, deterministic enhancement suitable as a strong initial solution for more sophisticated refinement techniques.

Hill-climbing strategies is a prominent approach, starting with an initial solution (often greedy) and iteratively exploring a neighborhood of solutions by making small changes, such as adding or removing a single subset, accepting only those that reduce the solution cost while maintaining full coverage, as described in works like Yagiura et al[4]. However, hill climbing often gets trapped in local optima due to its greedy nature and limited exploration. Our improved local search algorithm addresses these limitations by incorporating incremental coverage tracking to efficiently evaluate swaps, prioritizing subsets with low exclusive coverage for removal, and using adaptive swap sizes to balance exploration and exploitation. Additionally, periodic greedy re-optimization and solution perturbations help escape local optima, distinguishing our approach from traditional hill climbing by enhancing both efficiency and solution quality within a time-constrained framework.

Simulated annealing (SA) belongs to the family of local search as well. In 1983 three IBM researchers introduced the concepts of annealing in combinatorial optimization[5]. It is a probabilistic metaheuristic inspired by the physical annealing process in metallurgy, where a material is heated and slowly cooled to reach a low-energy state. During iterations, SA explores the solution space by accepting not only better solutions but also, with some probability, worse ones. The benefit of doing this is to prevent the algorithm from getting trapped in local minima and increases the chance of finding a global optimum. The temperature affects the probability of accepting worse neighboring solution. As the temperature "cools down", the probability of accepting worse solutions also shrinks. **Acceptance criterion (Metropolis condition)** chooses neighboring solution s' over the old solution s with the following probability (for **maximization**):

$$\Pr(s', s) = \begin{cases} 1, & \text{if } f(s') > f(s) \\ \exp\left(\dfrac{f(s') - f(s)}{T}\right), & \text{otherwise} \end{cases}$$

where T represents the temperature, starting with initial T_0 and controlled by the cooling rate α.

The main advantage of SA is its simplicity[6]. It's widely applied to real-life applications, such as the genetic[7] structure, protein structure[8], traveling tournament problems[9],police district design[10],and etc.

4 ALGORITHMS

4.1 Branch and Bound

4.1.1 Description and pseudo-code. The Branch-and-Bound (BnB) algorithm for the *Set Cover* problem explores subspaces of possible subset selections while pruning partial solutions that cannot improve the current best solution. At each step, the algorithm uses an upper bound (from a greedy heuristic) and a lower bound (current set count) to guide and limit the search.

Our implementation prioritizes nodes in a best-first manner using a gain-based heuristic, favoring subsets that cover more uncovered elements. We selected BnB for its ability to guarantee optimality and serve as a benchmark against approximation methods.

Algorithm 1 Branch-and-Bound

Input: Universe U, subset collection $\mathcal{S}$
$F \leftarrow \{(\emptyset, \emptyset, \mathcal{S})\}$ ▷ Frontier: selected sets, covered elements, undecided sets
$B \leftarrow (+\infty, \emptyset)$ ▷ Best solution: (cost, subset indices)
while F is not empty **do**
  Choose (X, C, R) from F with lowest priority
  Pick a set i from R to expand
  Let:
  $(X_1, C_1, R_1) \leftarrow (X \cup \{i\}, C \cup S_i, R \setminus \{i\})$ ▷ Include i
  $(X_2, C_2, R_2) \leftarrow (X, C, R \setminus \{i\})$ ▷ Exclude i
  for each configuration (X_i, C_i, R_i) **do**
    if $C_i = U$ **then**
      if $|X_i| < \text{cost}(B)$ **then**
        $B \leftarrow (|X_i|, X_i)$
      end if
    else if $R_i \neq \emptyset$ **then**
      if $|X_i| < \text{cost}(B)$ **then**
        $F \leftarrow F \cup \{(X_i, C_i, R_i)\}$
      end if
    end if
  end for
end while
return B

4.1.2 Algorithm. Given a universe $U = \{1, ..., n\}$ and collection of subsets $\mathcal{S} = \{S_1, ..., S_m\}$, BnB starts with an empty selection and initializes an upper bound via a greedy set cover. A priority queue maintains partial solutions ordered by their estimated lower bound.

Each node is expanded into two branches:

- Include a subset: add it to the current solution and update covered elements.
- Exclude the subset: proceed without it.

Nodes are pruned if their lower bound exceeds the current best solution. Promising branches are reinserted into the queue for further exploration.

4.1.3 Time and Space Complexity. In the worst case, BnB explores 2^m states. However, pruning drastically reduces the number of expansions in practice. Each node expansion takes $O(m)$ time, and queue operations add overhead depending on its size.

To manage memory, we cap the queue at 800,000 entries. Each entry stores a partial solution, covered elements, and a priority value, resulting in space complexity proportional to the queue size.

4.1.4 Strengths and Weaknesses. BnB guarantees optimal solutions and is ideal for benchmarking. For our case, however, due to preset cut off time, the optimal solution was not found for large cases. Its pruning and heuristics make it effective on moderate-size instances. However, it does not scale well for large or dense problems due to exponential growth and high memory use. As such, it trades scalability for exactness and is best used when correctness is critical.

4.2 Greedy Approximation Algorithm

4.2.1 Description and pseudo-code. The algorithm is a standard greedy approach for the Set Cover problem. It iteratively selects subsets that cover the largest number of yet-uncovered elements. Initially, subsets are sorted in descending order based on their size. This sorting serves as a heuristic to prioritize sets with potentially higher coverage early on.

Algorithm 2 Greedy Approximation

Input: Universe U, subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$
Output: Approximate solution X (subset indices), solution size $|X|$
$X \leftarrow \emptyset$ ▷ Selected subset indices
$C \leftarrow \emptyset$ ▷ Covered elements
Compute sizes: $L \leftarrow$ sorted list of $(i+1, |S_i|)$ in descending order of size
while $C \neq U$ **do**
  $best\_idx \leftarrow -1$
  $best\_coverage \leftarrow 0$
  for all $(i, \_) \in L$ **do**
    if $i \notin X$ **then**
      $new\_cov \leftarrow |S_{i-1} \setminus C|$
      if $new\_cov > best\_coverage$ **then**
        $best\_coverage \leftarrow new\_cov$
        $best\_idx \leftarrow i$
      end if
    end if
  end for
  if $best\_idx = -1$ or $best\_coverage = 0$ **then**
    break
  end if
  $X \leftarrow X \cup \{best\_idx\}$
  $C \leftarrow C \cup S_{best\_idx-1}$
end while
return $(X, |X|)$

4.2.2 Approximation Guarantees. This greedy algorithm follows the classical greedy heuristic for set cover and is known to provide a logarithmic approximation ratio.

$$\text{Approximation ratio} \leq H(d) \leq \ln n + 1$$

where $H(d)$ is the d-th harmonic number and d is the size of the largest subset, and $n = |U|$ is the size of the universe. This approximation bound is tight unless $P = NP$ [3].

4.2.3 Time and Space Complexity Analysis: The time complexity of the algorithm involves sorting the subsets by size, which takes $O(m \log m)$, and in each of the at most n iterations (where n is the number of elements in the universe), it checks all subsets for coverage, contributing $O(m \cdot n)$ per iteration. This leads to a total time complexity of $O(m^2 \cdot n)$.

The space complexity of the greedy approximation algorithm for the Set Cover problem is dominated by the storage of the input and auxiliary data structures. The universe U with n elements requires $O(n)$ space, while the collection of m subsets $\mathcal{S} = \{S_1, \ldots, S_m\}$, where each subset may contain up to n elements, demands $O(m \cdot n)$ space when stored explicitly (e.g., as lists or sets). The algorithm maintains the set X of selected subset indices (up to m, using $O(m)$ space), the set C of covered elements (up to n, using $O(n)$ space), and the sorted list L of m pairs (using $O(m)$ space). Temporary variables and auxiliary space for set operations (e.g., computing $S_{i-1} \setminus C$) require at most $O(n)$ space. Thus, the total space complexity is $O(m \cdot n)$, driven by the input storage.

$$\text{Space Complexity: } O(m \cdot n)$$

4.2.4 Strengths and Weaknesses. This approach is simple and easy to implement, provides a guaranteed approximation ratio, and scales well for moderately large instances. However, it has several weaknesses: it ignores potential synergies between subsets, early greedy decisions can lead to suboptimal final solutions, and it does not incorporate any local search or optimization refinement.

4.2.5 Heuristics and Tuning. A simple heuristic was applied: sorting the subsets by their size in descending order before beginning the greedy loop. This does not change the worst-case behavior but can improve performance in practice by reducing the number of iterations needed to find sets with high coverage.

No local search, tuning, or metaheuristics (e.g., simulated annealing, genetic algorithms) were used in this implementation.

4.3 Local Search (Hill Climbing)

4.3.1 Description and pseudo-code. We implemented a local search algorithm to approximate the solution for the Set Cover problem. The algorithm begins with a greedy initialization to quickly generate a valid cover. During the main search loop, it iteratively removes subsets that contribute least to the solution based on exclusive coverage. If the cover becomes invalid, a repair phase is triggered where subsets covering the newly uncovered elements are re-added. A new solution is accepted if it is valid and smaller in size than the current best. Additionally, if no improvement is observed for several iterations, the algorithm invokes a greedy re-optimization as a heuristic to escape stagnation.

4.3.2 Approximation Guarantees. The initial greedy solution guarantees an $O(\log n)$ approximation, where n is the number of elements. The local search may improve this empirically, but it does not guarantee better theoretical bounds.

4.3.3 Complexity Analysis. The algorithm initializes with reading the instance ($O(m \cdot n)$), computing a greedy solution ($O(m \cdot n \cdot$

Algorithm 3 Local Search (hill climbing)

Require: Instance, cutoff time, seed
Initialize random seed and read instance
$S \leftarrow$ GreedySolution(), $C \leftarrow$ CoverageCounts(S)
$S^* \leftarrow S$, $swap \leftarrow 1$, $no\_improve \leftarrow 0$
while time < cutoff **do**
  $E \leftarrow$ ExclusiveCoverage(S)
  Remove $swap$ subsets with smallest E
  if All elements covered **then**
    if $|S| < |S^*|$ **then**
      $S^* \leftarrow S$, $swap \leftarrow 1$, $no\_improve \leftarrow 0$
      continue
    end if
  end if
  $U \leftarrow$ UncoveredElements()
  Add top subsets covering U to S
  if Valid(S) and $|S| < |S^*|$ **then**
    $S^* \leftarrow S$, $swap \leftarrow 1$, $no\_improve \leftarrow 0$
  else
    Revert S, $swap \leftarrow swap + 1$, $no\_improve++$
  end if
  if $no\_improve > limit$ **then**
    $S \leftarrow$ GreedyImprove(S), $no\_improve \leftarrow 0$
  end if
end while
return S^*

$\min(m, n)$)), and coverage counts ($O(m \cdot n)$). The main loop runs for T iterations (based on cutoff time), with each iteration performing exclusive coverage computation ($O(m \cdot n)$), subset removal ($O(m)$), coverage checks ($O(n)$), uncovered elements computation ($O(n)$), subset addition ($O(m \cdot n)$), and validation ($O(m \cdot n)$). The rare `GreedyImprove` step takes $O(m \cdot n \cdot \min(m, n))$. Assuming `GreedyImprove` is infrequent, the total time complexity is dominated by the loop, yielding $O(T \cdot m \cdot n)$.

$$\text{Time Complexity: } O(T \cdot m \cdot n)$$

The input consists of the universe U ($O(n)$) and m subsets $\mathcal{S}$ ($O(m \cdot n)$). The algorithm uses sets S, S^* ($O(m)$), coverage counts, exclusive coverage, and uncovered elements ($O(n)$), plus temporary storage for set operations ($O(n)$). The total space complexity is dominated by the input storage, resulting in $O(m \cdot n)$.

$$\text{Space Complexity: } O(m \cdot n)$$

4.3.4 Strengths and Weaknesses. This approach is flexible and simple to implement, can achieve better solutions than greedy alone in practice, and is easy to integrate with various heuristics. However, it offers no strong theoretical improvement over greedy, is sensitive to parameter tuning (such as swap size and iteration limits), and may get stuck in local minima without global diversification.

4.3.5 Heuristics and Tuning. Several heuristics and tuning strategies are employed to improve performance. The swap size is dynamically increased if no improvement is observed, allowing exploration of larger neighborhoods. Greedy re-optimization is applied to escape stagnation and guide the search toward better solutions. A time cutoff is used to limit total computation, serving as a practical stopping criterion.

4.4 Local Search (Simulated Annealing)

4.4.1 Description and pseudo-code. The Simulated Annealing algorithm begins with an initial feasible solution(here we import the solution from approximation algorithm) and explores the solution space through stochastic local modifications. At each iteration, a neighbor solution is generated by removing one subset from the current solution and greedily repairing coverage until the universe is fully covered. The updated solution is probabilistically accepted based on the Metropolis criterion: if it improves the cost, it is always accepted; otherwise, it is accepted with a probability that decreases exponentially with both the cost difference and the current temperature. To prevent stagnation, the algorithm restarts if the current cost deteriorates by more than 50% relative to the best solution found. The temperature is geometrically decreased after a fixed number of iterations per level. The best solution and a trace of the cost over time are maintained and returned upon reaching the time limit or minimum temperature.

4.4.2 Complexity Analysis. Let $n = |U|$ denote the size of the universe, and $m = |\mathcal{S}|$ the number of subsets. Let N_{iter} be the number of iterations per temperature level, and L the number of cooling levels. In each inner loop iteration, generating a neighbor takes $O(1)$ time, while the greedy repair step may examine all subsets to cover uncovered elements, costing $O(nm)$ in the worst case. With N_{iter} iterations per level and L temperature levels, the overall time complexity is:

$$O(L \cdot N_{\text{iter}} \cdot nm)$$

The temperature is reduced geometrically, then $L = O(\log(1/T_{\min}))$, yielding:

$$O(N_{\text{iter}} \cdot \log(1/T_{\min}) \cdot nm)$$

The algorithm maintains the initial solution S_0(for restart), current solution S and best solution S^*, each of size at most m. The universe and subsets require $O(nm)$ space if stored explicitly. The trace list stores up to $L \cdot N_{\text{iter}}$ entries in the worst case. Thus, the total space complexity is:

$$O(nm + L \cdot N_{\text{iter}})$$

4.4.3 Approximation Guarantees. Unfortunately, simulated Annealing offers no formal approximation bound on the solution quality in polynomial time. While it can theoretically converge to the global optimum under an **infinite-time** schedule with appropriate cooling, it is impractical for real-world instances.

4.4.4 Strengths and weaknesses. Simulated Annealing is effective in escaping local minima by probabilistically accepting worse solutions, making it well-suited for rugged or multi-modal search landscapes. Moreover, the algorithm is simple to implement and offers a tunable balance between exploration and exploitation via its cooling schedule.

However, despite its flexibility, simulated annealing lacks theoretical approximation guarantees in bounded time and is highly sensitive to its parameter settings. Convergence to near-optimal solutions may require extensive computation time, and the stochastic nature of the search can lead to inconsistent performance across

Algorithm 4 Local Search (simulated annealing)

Require: cutoff time, seed, universe U, subsets $\mathcal{S} = \{S_1, \ldots, S_m\}$, initial solution S_0, initial cost c_0
Ensure: Best solution found and trace of (time, cost)

Set random seed
Initialize: $S \leftarrow S_0$, $c \leftarrow c_0$, $S^* \leftarrow S$, $c^* \leftarrow c$
Initialize trace $\mathcal{T} \leftarrow [(0, c^*)]$
Set initial temperature $T \leftarrow 1.0$, final temperature $T_{\min} \leftarrow 0.01$, cooling rate $\alpha \leftarrow 0.95$
Set max iterations per temperature level $N_{\text{iter}} \leftarrow 10000$
Start timer
while time elapsed $< T_{\max}$ and $T > T_{\min}$ **do**
    for $i = 1$ to N_{iter} **do**
        if time elapsed $> T_{\max}$ **then**
            break
        end if
        Generate neighbor S' by removing one set from S
        Recompute uncovered elements: $U' \leftarrow U \setminus \bigcup_{j \in S'} S_j$
        while $U' \neq \emptyset$ **do**
            Add subset S_k maximizing $|U' \cap S_k|$ to S'
            Update $U' \leftarrow U' \setminus S_k$
        end while
        Let $c' \leftarrow |S'|$, $\Delta^* \leftarrow c' - c^*$, $\Delta \leftarrow c' - c$
        if $\Delta^*/c^* \geq 0.5$ **then**
            Restart from S_0: $S \leftarrow S_0$, $c \leftarrow c_0$
        else if $\Delta < 0$ **or** $\exp(-\Delta/T) > \text{rand}(0, 1)$ **then**
            Accept: $S \leftarrow S'$, $c \leftarrow c'$
            if $c < c^*$ **then**
                Update best: $S^* \leftarrow S$, $c^* \leftarrow c$
                Append to trace: $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\text{elapsed}, c^*)\}$
            end if
        end if
    end for
    Update temperature: $T \leftarrow \alpha T$
end while
return $(S^*, \mathcal{T})$

runs. Another issue stems from the general-purpose nature of simulated annealing. It is designed to be applicable to a wide range of optimization problems without incorporating problem-specific heuristics. As a result, it lacks tailoring to the structural properties of the minimum set cover problem, which hinder our abilities to improve its performance.

4.4.5 *Heuristics and Tuning.*

- **Cooling rate:**The temperature schedule follows a geometric cooling scheme, starting from an initial temperature of 1.0 and decreasing by a factor of $\alpha = 0.95$ until reaching a minimum threshold of 0.01.
- **neighbor size:** At each temperature level, the algorithm performs up to 10,000 neighbor evaluations, where a neighbor is constructed by removing a randomly selected subset and greedily adding new subsets to restore full coverage.
- **Cutoff time:** After several experiments, 60 seconds is enough to cut off the algorithm.
- **Restart threshold:** To avoid stagnation in poor regions of the solution space, the algorithm incorporates a restart mechanism: if a candidate solution's cost deteriorates by more than 50% relative to the best-known solution, the search is reset to the initial configuration.
- **Acceptance criterion:**Acceptance of worse solutions is governed by the Metropolis criterion, where the probability of acceptance decays exponentially with the cost increase and inversely with the current temperature.

These hyperparameters significantly influence the balance between exploration and exploitation. For large datasets, A smaller α, a larger neighborhood size, a wider temperature range, a more relaxed restart condition, and a longer cutoff time can help improve the algorithm's performance — particularly the cooling rate α, the neighborhood size, and the temperature spread.

5 EMPIRICAL EVALUATION

5.1 Platform and Environment

- **Branch-and-Bound Algorithm**
  All experiments were conducted on a MacBook Pro with an Apple M2 chip (8-core CPU), 16 GB of RAM, using Python 3.11 with the default CPython compiler. The Branch-and-Bound (BnB) algorithm was implemented in Python and executed within a Jupyter Notebook environment. To manage memory usage, the priority queue used in the algorithm was explicitly capped at 800,000 entries. Each instance was run with a time cutoff of 900 seconds.
- **Approximation Algorithm**
  All experiments were conducted on a machine running macOS Sequoia 15.3.1 with an Apple M1 Pro chip and 16 GB unified memory. The implementation was written in Python 3.8.18. Code was run in a single-threaded setting to ensure consistency across runs.
- **Local Search (Hill Climbing)**
  The Hill Climbing algorithm experimental setup utilized a macOS Sequoia 15.3.1 system powered by an Apple M3 Pro chip with 18 GB of unified memory. Implemented in Python 3.8.18, the code was executed in a single-threaded configuration to ensure consistent and reproducible results.
- **Local Search (Simulated Annealing)**
  The Simulated Annealing algorithm was tested on a Lenovo ThinkBook with 32 GB of RAM and an Intel® Core™ Ultra 7 155H 1.40 GHz processor. The code was executed using Python 3.12 (64-bit).

5.2 Experimental Setup and Metrics

We evaluated algorithm performance based on the *collection size*, defined as the number of subsets selected in the final cover. To assess solution quality, we compared the BnB output to known optimal values (`OPT`) and computed the *relative error* as $(\text{ALG} - \text{OPT})/\text{OPT}$. Running time was recorded in seconds. Tables 1 and 2 summarize the results for large and small datasets, respectively.

For local search algorithms,i.e. hill climbing and simulated annealing, as required we test on `large1` and `large10` datasets for 20 runs to plot QRTD and SQD to analyze the running time and solution quality.

5.3 Results

The results from all algorithms were compiled into a comprehensive table summarized in Table 1 and Table 2 for large cases and small cases. The plots for each algorithm's relative error and running time are plotted in Figure 1-4.:

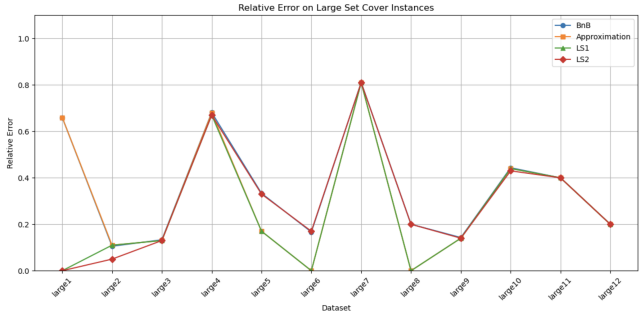


Figure 1: Comparison of relative errors on large dataset

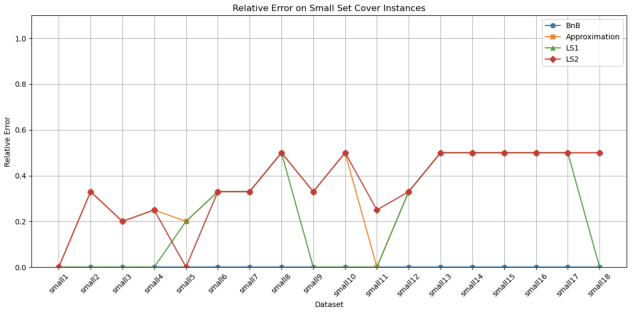


Figure 2: Comparison of relative errors on small dataset

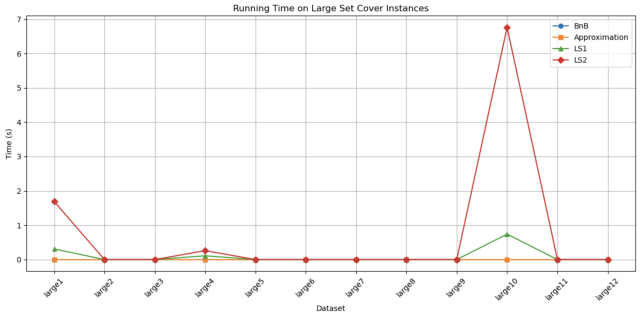


Figure 3: Comparison of running time on large dataset

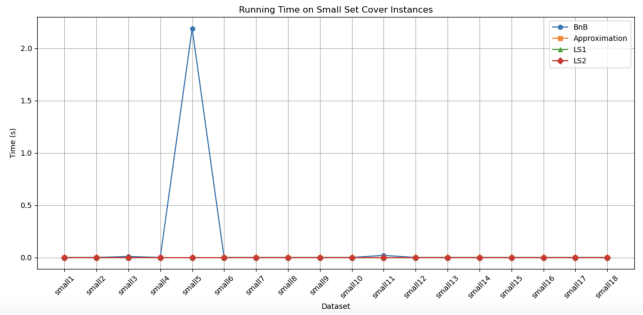


Figure 4: Comparison of running time on small dataset

- **Branch-and-Bound Algorithm**
  As shown in Table 1, the BnB algorithm reaches the cut-off time on all large instances, and in several cases (e.g., `large1`, `large4`, `large7`), the resulting solutions exhibit high relative errors exceeding 0.6. This indicates that the algorithm was unable to sufficiently explore the search space due to the combination of memory constraints and time limits.
  In contrast, Table 2 shows that all small instances were solved optimally, with relative error of 0.0000 across the board. Most of these were completed within milliseconds or a few seconds, demonstrating that BnB performs effectively on small input sizes.
- **Approximation Algorithm** From Tables 2 and 1, we observed that the greedy approximation algorithm demonstrated excellent runtime performance, consistently completing in under 0.01 seconds across all datasets. Solution quality was high for smaller instances, with several cases (e.g., `small1`, `small11`) matching the optimal solution exactly. However, accuracy declined on larger datasets, where relative errors exceeded 0.65 in cases like `large7` and `large4`, reflecting the algorithm's tendency to over-select in the presence of overlapping subsets. These results highlight the greedy method's utility as a fast and reasonable baseline, particularly when used as an initializer for more advanced refinement techniques.
- **Local Search (Hill Climbing)**
  For the small datasets (Tables 2), the hill climbing algorithm consistently finds optimal solutions (relative error = 0) in most cases, with a few instances showing modest deviation (e.g., up to 50% relative error). Also, we noted that the runtime are negligible (0 seconds for all), due to the small size and simple structure of these instances. On the large datasets (Tables 1), it performs reasonably well on moderately sized problems (e.g., `large1`, `large2`, `large3`) with low relative errors and short runtimes. However, its performance degrades on more complex instances (e.g., `large4`, `large7`, `large10`), where the relative error exceeds 40% or even 80%. This suggests that while this algorithm is efficient in practice, it can get stuck in poor local optima on large or highly constrained instances. No instance exceeded one second in runtime, showing the method's computational

Table 1: Comparison of All Four Algorithms on Large Set Cover Instances

Dataset	OPT	BnB			Approx			LS1			LS2		
		CollSz	RelErr	Time	CollSz	RelErr	Time	CollSz	RelErr	Time	CollSz	RelErr	Time
Platform / Env	–	Mac M2 Pro, 16GB, Python 3.11			Mac M1 Pro, 16GB, Python 3.8.18			Mac M3 Pro, 18GB, Python 3.8.18			Lenovo ThinkBook, Intel i7, 32GB, Python 3.12		
large1	50	83	0.6600	0.00	83	0.66	0.00	50	0.00	0.31	50	0.00	1.69
large2	19	21	0.1053	0.00	21	0.11	0.00	21	0.11	0.00	20	0.05	0.00
large3	15	17	0.1333	0.00	17	0.13	0.00	17	0.13	0.00	17	0.13	0.00
large4	91	153	0.6813	0.00	153	0.68	0.00	152	0.67	0.11	152	0.67	0.26
large5	6	8	0.3333	0.00	7	0.17	0.00	7	0.17	0.00	8	0.33	0.00
large6	6	7	0.1667	0.00	6	0.00	0.00	6	0.00	0.00	7	0.17	0.00
large7	95	172	0.8105	0.00	172	0.81	0.00	172	0.81	0.00	172	0.81	0.00
large8	5	6	0.2000	0.00	5	0.00	0.00	5	0.00	0.00	6	0.20	0.00
large9	14	16	0.1429	0.00	16	0.14	0.00	16	0.14	0.00	16	0.14	0.00
large10	221	319	0.4434	0.00	319	0.44	0.00	318	0.44	0.74	317	0.43	6.76
large11	40	56	0.4000	0.00	56	0.40	0.00	56	0.40	0.00	56	0.40	0.00
large12	15	18	0.2000	0.00	18	0.20	0.00	18	0.20	0.00	18	0.20	0.00

2em

Table 2: Comparison of All Four Algorithms on Small Set Cover Instances

Dataset	OPT	BnB			Approx			LS1			LS2		
		CollSz	RelErr	Time	CollSz	RelErr	Time	CollSz	RelErr	Time	CollSz	RelErr	Time
Platform / Env	–	Mac M2 Pro, 16GB, Python 3.11			Mac M1 Pro, 16GB, Python 3.8.18			Mac M3 Pro, 18GB, Python 3.8.18			Lenovo ThinkBook, Intel i7, 32GB, Python 3.12		
small1	5	5	0.00	0.00	5	0.00	0.00	5	0.00	0.00	5	0.00	0.00
small2	3	3	0.00	0.00	4	0.33	0.00	3	0.00	0.00	4	0.33	0.00
small3	5	5	0.00	0.01	6	0.20	0.00	5	0.00	0.00	6	0.20	0.00
small4	4	4	0.00	0.00	5	0.25	0.00	4	0.00	0.00	5	0.25	0.00
small5	5	5	0.00	2.19	6	0.20	0.00	6	0.20	0.00	5	0.00	0.00
small6	3	3	0.00	0.00	4	0.33	0.00	4	0.33	0.00	4	0.33	0.00
small7	3	3	0.00	0.00	4	0.33	0.00	4	0.33	0.00	4	0.33	0.00
small8	2	2	0.00	0.00	3	0.50	0.00	3	0.50	0.00	3	0.50	0.00
small9	3	3	0.00	0.00	4	0.33	0.00	3	0.00	0.00	4	0.33	0.00
small10	2	2	0.00	0.00	3	0.50	0.00	2	0.00	0.00	3	0.50	0.00
small11	4	4	0.00	0.02	4	0.00	0.00	4	0.00	0.00	5	0.25	0.00
small12	3	3	0.00	0.00	4	0.33	0.00	4	0.33	0.00	4	0.33	0.00
small13	2	2	0.00	0.00	3	0.50	0.00	3	0.50	0.00	3	0.50	0.00
small14	2	2	0.00	0.00	3	0.50	0.00	3	0.50	0.00	3	0.50	0.00
small15	2	2	0.00	0.00	3	0.50	0.00	3	0.50	0.00	3	0.50	0.00
small16	2	2	0.00	0.00	3	0.50	0.00	3	0.50	0.00	3	0.50	0.00
small17	2	2	0.00	0.00	3	0.50	0.00	3	0.50	0.00	3	0.50	0.00
small18	2	2	0.00	0.00	3	0.50	0.00	2	0.00	0.00	3	0.50	0.00

efficiency, but highlighting a trade-off between speed and solution quality on harder problems.

We also observed that the hill climbing algorithm demonstrates contrasting performance on `large1` and `large10`, as evidenced by QRTD and SQD plots. On `large1`, the algorithm consistently achieves the global optimum (RelErr = 0.00, Collection Size = 50 = OPT) in all 20 runs within 0.25 seconds, with runtimes ranging from 0.05 to 0.25 seconds. This rapid convergence suggests that `large1` has a favorable structure, likely with uniform coverage, allowing the algorithm to efficiently prune redundant subsets and find the optimal solution. Conversely, on `large10`, the algorithm fails to reach the global optimum in any run, achieving a best solution quality of 43.5% deviation (RelErr = 0.44, Collection Size = 318 vs. OPT = 221) after 1.305 seconds. The longer runtimes, ranging from 0.125 to 1.305 seconds with an average around 1.0 second, indicate that `large10`'s structure—likely sparse with limited coverage—poses significant challenges, trapping the algorithm in local optima despite extended iterations. These results underscore the algorithm's sensitivity to instance structure, excelling on instances that support efficient optimization while struggling with those having constrained coverage options, consistent with earlier performance analyses comparing sparse and dense instances.

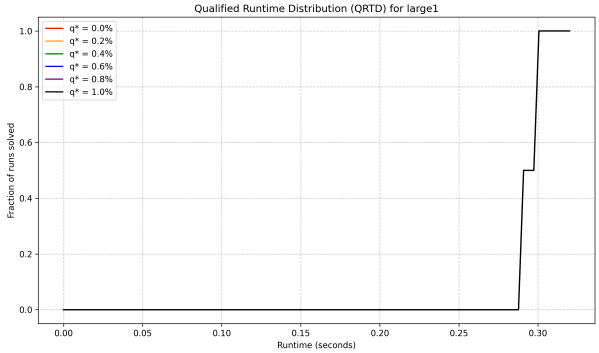
- **Local Search (Simulated Annealing)**

  For small datasets, the simulated annealing algorithm runs quickly and often finds a local optimum. However, as observed, when the global optimum solution size decreases (e.g., from `small1` to `small18`), the algorithm's relative error increases. Even with a large neighborhood size and a slow cooling rate, the algorithm still gets trapped in local optima. The largest relative error reaches 50% for `small13`–`small18`, where the global optimum value is as small as 2. On average, the relative error across all small datasets rises to 35%.

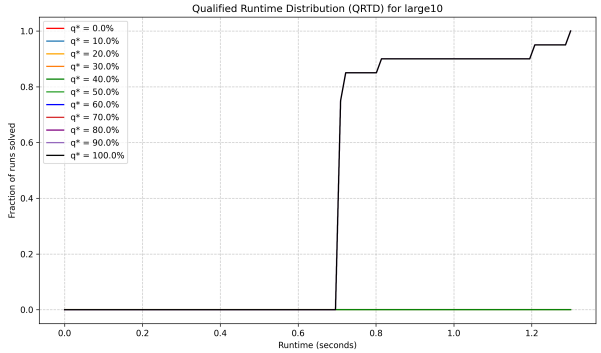
  For large datasets, the simulated annealing algorithm runs longer and generally achieves better performance, with an average relative error of 29.5%. More specifically, it performs well on `large1` (relative error: 0%), `large2` (5%), `large3` (13%), `large6` (17%), and `large9` (14%).

Now consider the running time analysis (see the QRTD, SQD, and boxplot figures). Since the algorithm successfully finds the global optimum in all 20 runs for `large1`, the QRTD plot for varying quality thresholds becomes a line at one, indicating consistent success. The average running time for `large1` is 1.69 seconds, and the maximum is 2.73 seconds. From the SQD plot for `large1`, the running times range from 0.27 to 2.73 seconds.

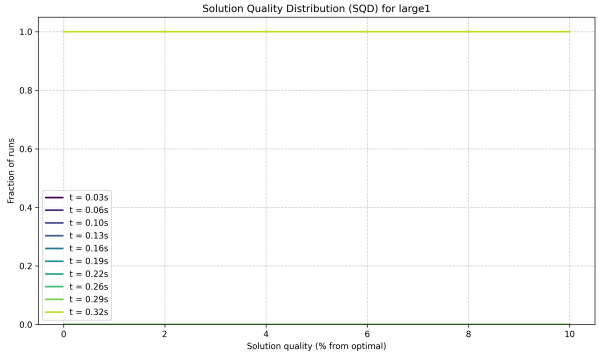
In contrast, the algorithm performs poorly on `large10`. It fails to find the global optimum in any of the 20 runs, even within the 60-second time limit. The average relative error reaches 43%, with an average running time of 6.76 seconds and a maximum of up to 14.35 seconds. This indicates that despite running many iterations for a long time, the algorithm is unable to find the global optimum and struggles to produce high-quality solutions.



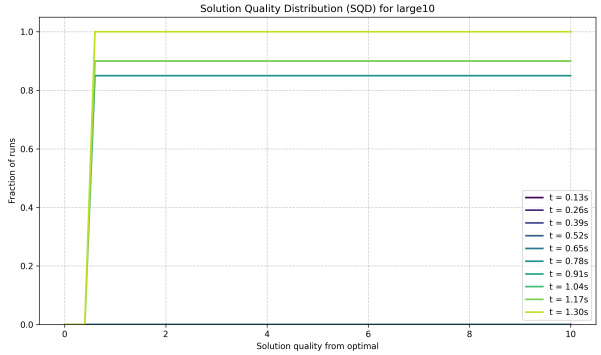
(a) Hill Climbing: QRTD on `large1`



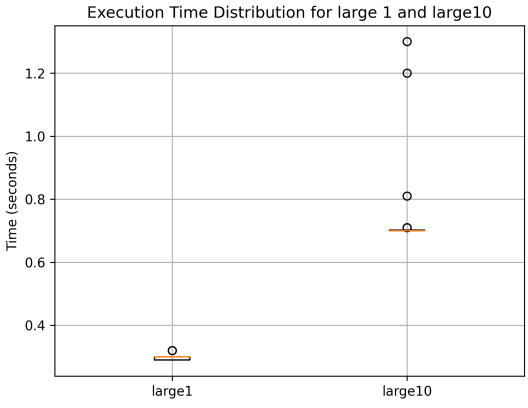
(b) Hill Climbing: QRTD on `large10`



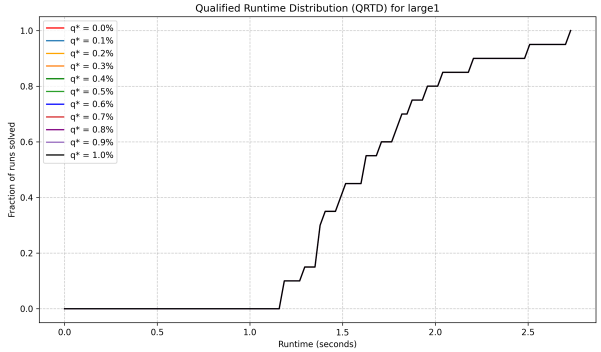
(c) Hill Climbing: SQD on `large1`



(d) Hill Climbing: SQD on `large10`



(e) Hill Climbing: Boxplot on `large1` and `large10`



(f) Simulated Annealing: QRTD on `large1`

Figure 5: Performance plots for Hill Climbing and Simulated Annealing

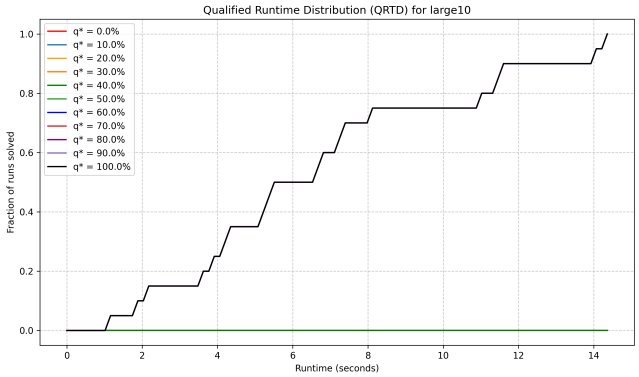


Figure 6: Simulated Annealing: QRTD on `large10`

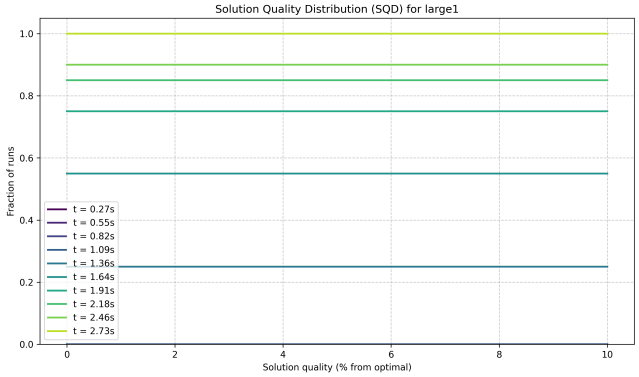


Figure 7: Simulated Annealing: SQD on `large1`

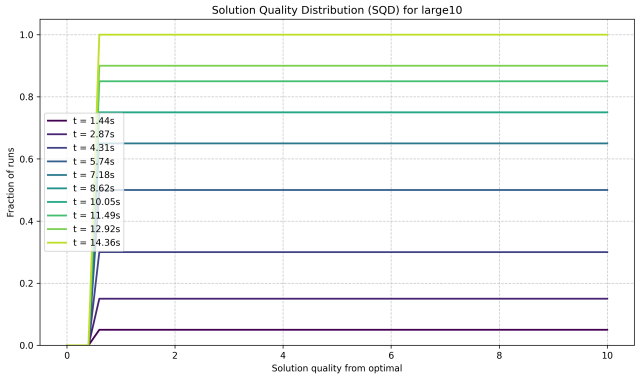


Figure 8: Simulated Annealing: SQD on `large10`

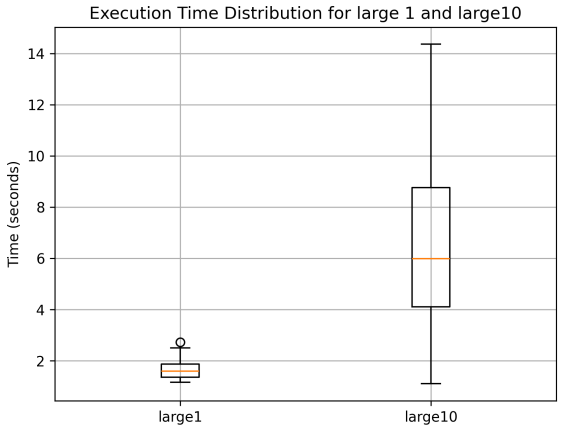


Figure 9: Simulated Annealing: Boxplot on `large1` and `large10`

6 DISCUSSION

- **Branch-and-Bound** Branch-and-Bound (BnB) is the only exact algorithm in our comparison and performs exceptionally well on small instances, consistently producing optimal solutions with zero relative error and minimal runtime. Its strength lies in its ability to fully explore or prune the solution space efficiently when the problem size is limited. In contrast, for large instances, BnB effectively behaves as a greedy algorithm: the final solution is found in 0.00 seconds, identical to the initial greedy upper bound, and no further improvement is made. This outcome stems from the algorithm's early pruning or inability to find better solutions within the time and memory constraints. These results confirm that while the BnB algorithm guarantees optimality in theory, its practical scalability is limited. For small to moderately sized instances, it remains a reliable and interpretable benchmark for evaluating heuristic or approximate methods.
  Interestingly, local search methods (LS1 and LS2) occasionally outperform BnB in large cases by slightly improving upon the greedy baseline, albeit without guarantees. This suggests that while BnB guarantees optimality, its practical usefulness on large datasets is limited unless advanced bounding or extended runtime is employed. In scenarios where solution quality is more important than optimality certification, local search offers a better trade-off between performance and runtime. Nonetheless, BnB remains a powerful baseline and diagnostic tool, especially when paired with a strong greedy initializer.
- **Greedy Approximate** The greedy approximation algorithm for the set cover problem is fast and effective for dense instances with high subset overlap (`large6`, `large8`, `small1`, `small11`), achieving optimal or near-optimal solutions due to redundant coverage. However, it struggles with sparse instances (`large4`, `large7`, `large10`, `large11`) where limited coverage options lead to high errors (RelErr = 0.40–0.81), and dense instances with complex coverage (small8, small13–18, RelErr = 0.50). Compared to the hill climbing algorithm, the greedy algorithm provides a reasonable but often suboptimal starting point, with hill climbing improving solutions in sparse instances like large1 (RelErr = 0.66 to 0.00) but offering limited gains in others (`large7`). Instance structure significantly influences performance, with sparse instances requiring careful handling of low coverage and dense instances needing navigation of complex patterns. Combining the greedy algorithm with hill climbing or adaptive strategies could enhance its performance across diverse instance types. The greedy algorithm achieved sub-0.01s runtimes across all datasets, confirming its efficiency. On small instances, it often found optimal solutions (e.g., `small1`, `small11`) or maintained low relative error. For larger datasets, solution quality degraded, with relative errors exceeding 0.65 in cases like `large7` and `large4`, highlighting its sensitivity to overlapping or redundant sets. Despite this, the algorithm remains a strong baseline, offering fast approximations suitable for initialization in more sophisticated methods.
- **Local Search (Hill Climbing)**
  The performance of the hill climbing algorithm for the set cover problem is strongly influenced by the structure of the input instances and the algorithm's inherent limitations. Sparse instances such as `large4`, `large7`, and `large10` tend to be challenging due to limited coverage options, often leading the algorithm to select an excessive number of subsets and produce suboptimal solutions. Interestingly, sparsity can also yield optimal results in cases like `large1`, where uniform coverage allows for effective pruning. Conversely, dense instances like `large8` and `small1` generally favor the algorithm, as the abundance of overlapping subsets enables greedy heuristics to perform well. Nonetheless, not all dense instances are easy—examples such as `small8` and `small13–small17` demonstrate how intricate coverage patterns can trap the algorithm in local optima. Overall, the algorithm's myopic search strategy and reliance on greedy decisions make it susceptible to both insufficient coverage in sparse cases and misleading redundancy in dense ones. Its sensitivity to hyperparameters like the swap rate and no-improvement threshold further contributes to inconsistent performance across diverse instance structures.
  On the other hand, a notable observation for the hill climbing algorithm is its performance on instance `large1`, where the objective value improved significantly from the greedy solution (83) to 50, achieving the optimal solution. This improvement stems from the algorithm's initialization with the greedy solution, followed by refinement through pruning redundant subsets. For `large1`, a sparse instance with uniform coverage (each element covered by exactly 40 subsets), the greedy algorithm tends to select redundant subsets, inflating the solution size. The hill climbing algorithm exploited this uniform structure by identifying and removing subsets that provided non-exclusive coverage, resulting in a substantial gain in solution quality.
- **Local Search (Simulated Annealing)**
  Compared to other algorithms, simulated annealing does demonstrate a strong competitive edge in terms of solution quality when applied to large instances of the minimum set cover problem. This advantage is primarily due to its ability to temporarily accept worse solutions and explore the solution space randomly in search of better alternatives. However, this algorithm has some limitations too. First, as discussed in the previous section, the algorithm is highly sensitive to parameter tuning—such as the acceptance probability for worse solutions and the size of the neighborhood—which complicates convergence. Second, the computation time is much longer than others working on large datasets.
  For simulated annealing, running time does not appear to be a major constraint if the cooling rate and neighborhood size are properly configured. Conversely, if these parameters are poorly chosen, extended running time offers little improvement.

7 CONCLUSION

This study compared four algorithms for the Minimum Set Cover Problem. All performed well on small datasets, achieving optimal or near-optimal solutions quickly. On larger instances, performance varied: Branch-and-Bound was exact but limited by time and memory; the Greedy algorithm was fast but less accurate on complex inputs. Hill Climbing improved greedy solutions efficiently but struggled with local optima, while Simulated Annealing offered better robustness at the cost of longer runtimes and parameter sensitivity. Each algorithm has strengths and trade-offs. Combining fast approximations with adaptive local search holds promise for improving performance on challenging instances. Further exploration of dataset characteristics could help fine-tune local search algorithms and enhance their effectiveness on more complex instances.

REFERENCES

[1] Bernhard H Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial optimization*, volume 1. Springer, 2011.

[2] Vijay V Vazirani. *Approximation algorithms*, volume 1. Springer, 2001.

[3] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[4] Mutsunori Yagiura, Shinji Iwasaki, Toshihide Ibaraki, and Fred Glover. A very large-scale neighborhood search algorithm for the multi-resource generalized assignment problem. *Discrete Optimization*, 1(1):87–98, 2004.

[5] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[6] Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. Simulated annealing: From basics to applications. In *Handbook of metaheuristics*, pages 1–35. Springer, 2018.

[7] Isabelle Dupanloup, Stefan Schneider, and Laurent Excoffier. A simulated annealing approach to define the genetic structure of populations. *Molecular ecology*, 11(12):2571–2581, 2002.

[8] Kuo-Chen Chou and Louis Carlacci. Simulated annealing approach to the study of protein structures. *Protein Engineering, Design and Selection*, 4(6):661–667, 1991.

[9] Aris Anagnostopoulos, Laurent Michel, P Van Hentenryck, and Yannis Vergados. A simulated annealing approach to the traveling tournament problem. *Journal of Scheduling*, 9:177–193, 2006.

[10] Steven J D'Amico, Shoou-Jiun Wang, Rajan Batta, and Christopher M Rump. A simulated annealing approach to police district design. *Computers & operations research*, 29(6):667–684, 2002.