CSE 6140: Final Project

Ayush Mohanty
Georgia Tech
Atlanta, USA
ayush.mohanty@gatech.edu

Hyun Woo Kim
Georgia Tech
Atlanta, USA
hyun_kim@gatech.edu

Hyunju Kim
Georgia Tech
Atlanta, USA
hkim3239@gatech.edu

Neil Barry
Georgia Tech
Atlanta, USA
nbarry7@gatech.edu

ABSTRACT

The Minimum Set Cover (MSC) problem is a classic NP-complete problem with wide-ranging applications in network design, resource allocation, and data mining. This paper investigates four algorithmic approaches for solving the MSC problem, balancing between optimality and computational efficiency: (1) an exact Branch-and-Bound (BnB) algorithm leveraging LP-relaxation and greedy-derived upper bounds, (2) a greedy Approximation algorithm with a logarithmic approximation guarantee, and two heuristic Local Search methods— (3) Simulated Annealing and (4) Hill Climbing—tailored to explore the solution space efficiently. We conduct comprehensive empirical evaluations across standardized test instances, including small and large-scale datasets. Results demonstrate that while Branch-and-Bound (BnB) guarantees optimal solutions on all small and some medium-sized instances, its scalability is limited—requiring several hundred seconds on larger problems. The greedy approximation algorithm consistently provides the fastest solutions, typically completing in under a second, but with higher relative errors, especially in large instances where the solution size can exceed the optimum by up to 81%. Our local search variants, particularly Simulated Annealing (LS1), achieve strong empirical performance on large instances—often matching or closely approximating the optimum with low relative error and runtimes under one second—substantially outperforming the approximation baseline. This study highlights the trade-offs between accuracy, runtime, and scalability across the four types of algorithmic paradigms.

KEYWORDS

Algorithms, Minimum Set Cover, NP-Complete

ACM Reference Format:
Ayush Mohanty, Hyun Woo Kim, Hyunju Kim, and Neil Barry. 2025. CSE 6140: Final Project. In *CSE 6140*. ACM, New York, NY, USA, 14 pages.

1 INTRODUCTION

The Minimum Set Cover (MSC) problem is a fundamental NP-complete problem in computational complexity theory with significant applications across numerous domains including resource allocation, facility location, database query optimization, and computational biology. Given its theoretical intractability and practical importance, MSC problem has become a cornerstone benchmark in the study of combinatorial optimization and approximation algorithms.

At its core, the problem asks: *given a finite universe of elements and a collection of subsets whose union covers the universe, what is the smallest number of these subsets that together cover all elements*? While the decision version of the problem is NP-complete, the optimization version is equally challenging, driving the need for efficient algorithmic strategies that trade off between exactness and scalability. In real-world applications, the MSC problem naturally arises in scenarios such as minimizing the number of test cases to cover all code paths, selecting representative features for machine learning models, or deploying sensors to monitor environmental variables. As datasets grow in complexity, solving MSC problem instances exactly becomes computationally prohibitive, motivating the development of heuristic and approximation-based approaches that produce decent quality solutions within a feasible time.

This paper presents a comprehensive investigation of algorithmic approaches for solving the MSC problem through rigorous theoretical analysis and systematic empirical evaluation. We formulate and analyze multiple solution methodologies: (1) an exact branch-and-bound algorithm that guarantees optimality while managing computational complexity; (2) a greedy approximation algorithm with provable logarithmic approximation bounds; and (3) two distinct local search heuristics designed to establish an effective balance between solution quality and computational efficiency. Our methodology demonstrates the comparative advantages of different algorithmic paradigms when addressing the intrinsic complexity of the MSC problem.

Through methodical experimentation on standardized benchmark instances of varying complexity, we conduct a comparative performance analysis of these algorithms across multiple dimensions: solution quality relative to known optima, runtime efficiency, and scalability characteristics. Our experimental results demonstrate that while the branch-and-bound approach achieves provably optimal solutions for instances of moderate size, the proposed local search methods consistently provide near-optimal solutions with substantially reduced computational requirements for larger problem instances. The greedy approximation algorithm, despite its theoretical guarantees, exhibits inferior solution quality in practice but serves as an efficient initialization mechanism for more sophisticated metaheuristic approaches.

Main Contributions. In this paper, we make the following technical contributions:

- We implement and evaluate an exact branch-and-bound algorithm for MSC problem that incorporates LP-relaxation and approximation-based bounding for efficient pruning.
- We implement a classical greedy approximation algorithm with provable $O(\log n)$ approximation guarantees and analyze its practical performance on diverse datasets.

Conference'17, July 2017, Washington, DC, USA

- We design and implement two local search heuristics—Simulated Annealing and Hill Climbing with tabu enhancements—to explore high-quality solutions under time constraints.
- We conduct a systematic empirical study comparing all four approaches across small- and large-scale real-world benchmark datasets, analyzing their trade-offs in accuracy, runtime, and scalability.
- We provide open-source code, dataset preprocessing, and result visualizations for reproducibility and future benchmarking: github.com/hhyy0401/CSE6140.

2 PROBLEM DEFINITION

In this paper, we explore the Minimum Set Cover problem.

- **Input:** (1) A finite set (called the *universe*): $U = \{x_1, x_2, \ldots, x_n\}$, (2) a collection of subsets of U: $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, where $S_i \subseteq U$ for all $i = 1, \ldots, m$
- **Goal:** Find a subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that $\bigcup_{S_i \in \mathcal{T}} S_i = U$ and the number of sets in $\mathcal{T}$ is minimized.
- **Objective:** Minimize $|\mathcal{T}|$ subject to the constraint that every element of U is contained in at least one subset in $\mathcal{T}$.

This problem is one of Karp's original 21 NP-complete problems [3] and is fundamental in the study of combinatorial optimization and computational complexity.

3 RELATED WORK

A seminal approximation algorithm for MSC problem is the greedy heuristic introduced by Chvátal [1], which iteratively selects the subset covering the largest number of uncovered elements. This algorithm achieves an approximation ratio of H_n, the n^{th} harmonic number, which is approximately $\ln n$.

To improve upon the greedy approach, Hassin and Levin [2] developed an algorithm that achieves a better approximation ratio for specific instances of the MSC problem. Their method refines the selection process by considering additional problem structure, leading to improved performance over the classical greedy algorithm.

Further advancements include the work of Saket and Sviridenko [5], who provided enhanced approximation bounds for the MSC problem. Their research contributes to a deeper theoretical understanding of the problem's complexity and approximation limits.

4 PROPOSED METHODS

We introduce four algorithmic solutions to the Minimum Set Cover problem—Branch-and-Bound (BnB), Greedy Approximation (Approx), Simulated Annealing (LS1), and Hill Climbing (LS2). For each, we provide pseudocode, detailed explanations, and theoretical analyses including time and space complexity. For the local search methods, we also describe our tuning strategies and discuss their respective strengths and limitations.

4.1 Exact branch-and-bound solution

In this section, we discuss the Branch-and-Bound (BnB) methodology used to efficiently solve the Minimum Set Cover problem. Specifically, we address the limitations of naive enumeration and detail how bounding solutions using LP-relaxation and approximation algorithms provides a significantly more efficient exploration of the solution space.

4.1.1 Overview. A straightforward or naive solution to the MSC problem is to enumerate all possible combinations of subsets S_i, systematically deciding whether each subset is included or excluded from the solution. Each potential combination would then be checked for feasibility (covering the universe) and optimality. However, such an approach has severe limitations:

- It requires exploration of 2^m possible subset combinations, where m is the number of subsets.
- Feasibility checking for each subset combination is computationally expensive, leading to impractical runtimes even for modest problem sizes.

Thus, we must employ a more efficient strategy to explore solution sub-structures effectively.

4.1.2 Efficient Exploration through Branch-and-Bound. BnB improves upon naive enumeration by systematically exploring subsets of solutions, building a search tree, and leveraging bounds to prune large portions of the solution space without exhaustive exploration. The key ideas behind our approach include:

- Identifying promising solution sub-structures early.
- Computing tight lower and upper bounds to prune suboptimal or infeasible solution branches quickly.
- Systematically branching on critical decisions, guided by heuristic rules for efficiency.

4.1.3 Bounding Solutions to Improve Efficiency. The fundamental advantage of BnB arises from its ability to prune entire substructures of the solution tree by effectively bounding the quality of partial solutions. Specifically, two bounds are crucial to our implementation:

(1) **Lower Bound (LP-relaxation)**: Establishes the minimum possible cost required to complete a partial solution, helping identify and eliminate suboptimal branches early.
(2) **Upper Bound (Approximation Algorithm)**: Provides an initial feasible solution and a practical benchmark for the best-known solution at any point, further refining the pruning criterion.

4.1.4 Deriving a Reasonable Lower Bound. To derive a suitable lower bound, we relax the MSC problem to a Linear Program (LP). Specifically, for the subset of elements that remain uncovered at any node in the search tree, we solve the LP-relaxation formulated as follows:

$$\begin{aligned} \min \quad & \sum_{j=1}^{m} x_j \\ \text{s.t.} \quad & \sum_{j:e \in S_j} x_j \geq 1, \quad \forall e \in \text{uncovered}, \\ & 0 \leq x_j \leq 1, \quad \forall j \in \{1, \ldots, m\}. \end{aligned}$$

Since the LP-relaxation allows fractional solutions, its optimal solution value provides a tight and computationally efficient lower bound for the integer optimal solution.

Justification: The LP-relaxation solution space includes all integer feasible solutions. Hence, its fractional optimal objective value is guaranteed to be no greater than the true integer optimal value, making it a valid and often tight lower bound for pruning.

4.1.5 Deriving a Reasonable Upper Bound. We derive an initial feasible solution and thus an upper bound using a greedy approximation algorithm, well-known for its $O(\log n)$ approximation ratio, where n is the number of elements in the universe. The greedy algorithm proceeds iteratively by:

(1) Selecting the subset covering the largest number of uncovered elements at each step.
(2) Adding it to the solution and repeating the process until all elements are covered.

This upper bound sets an initial benchmark, allowing the Branch-and-Bound algorithm to prune any solutions that cannot improve upon the currently known best solution.

4.1.6 Time Complexity. While naive enumeration has exponential complexity $O(2^m)$, the BnB approach reduces practical runtime dramatically. Although worst-case complexity remains exponential, the strategic use of LP-based pruning and approximation-based bounds frequently results in significant pruning of the solution tree, rendering the algorithm practically efficient for most realistic instances.

4.1.7 Space Complexity. The space complexity primarily depends on the recursion stack and bookkeeping data structures maintained during execution. Practically, space usage is proportional to $O(m \cdot d)$, with d denoting the recursion depth. Effective pruning ensures the recursion depth d typically remains manageable in practice. subsubsection

Advantages Over the Naive Approach. The critical advantage of our method is that it leverages structural insights provided by LP-based lower bounds and approximation-derived upper bounds. Unlike naive enumeration, these bounds permit rapid and efficient pruning of large solution subspaces without explicit enumeration, significantly improving computational performance. By integrating LP-relaxation and greedy approximation algorithms into a BnB framework, our methodology efficiently addresses the combinational explosion inherent in the MSC problem, significantly outperforming naive enumeration approaches.

Pseudocode. Algorithm 1 captures our methodology.

Algorithm 1: Branch-and-Bound

Input: Universe U, subsets $\mathcal{S}$, cutoff time T
Compute initial feasible solution S_{init} via greedy approximation;
$S^* \leftarrow S_{init}$ (best known solution);
Function `BranchAndBound`($current\_cover$, $current\_solution$)**:**
  if *elapsed time exceeds* T **then**
    return;
  if $current\_cover = U$ **then**
    if $|current\_solution| < |S^*|$ **then**
      $S^* \leftarrow current\_solution$;
    return;
  Compute lower bound LB via LP-relaxation on uncovered elements;
  if $|current\_solution| + LB \geq |S^*|$ **then**
    return `// prune;`
  Select an uncovered element e_{min} with fewest subsets covering it;
  for *each subset* S_j *containing* e_{min}, *prioritized by coverage* **do**
    `BranchAndBound`($current\_cover \cup S_j$, $current\_solution \cup \{j\}$);
return optimal solution S^*;

4.2 Approximation algorithm

4.2.1 Overview. The greedy algorithm for the Minimum Set Cover problem follows an approach that iteratively selects the most "efficient" subset at each step until all elements in the universe are covered.

Initialization:

- Start with an empty solution set $\mathcal{T} = \emptyset$
- Initialize the set of uncovered elements $C = U$ (the entire universe)

Iterative Selection:

- While there are uncovered elements ($C \neq \emptyset$):
  - For each remaining subset $S_i \in \mathcal{S}$, calculate the number of currently uncovered elements it would cover: $|S_i \cap C|$
  - Select the subset S_j that maximizes this value, i.e., the subset that covers the maximum number of remaining uncovered elements
  - Add S_j to the solution: $\mathcal{T} = \mathcal{T} \cup S_j$
  - Update the set of uncovered elements: $C = C \setminus S_j$

Termination:

- When all elements are covered ($C = \emptyset$), return the selected subsets $\mathcal{T}$

Algorithm 2: GreedySetCover

Data: Universe $U = \{x_1, x_2, \ldots, x_n\}$, Collection of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$
Result: A subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that $\bigcup_{S_i \in \mathcal{T}} S_i = U$
$\mathcal{T} \leftarrow \emptyset$;
$C \leftarrow U$; `// Set of elements still to be covered`
while $C \neq \emptyset$ **do**
  Select $S_i \in \mathcal{S}$ that maximizes $|S_i \cap C|$;
  $\mathcal{T} \leftarrow \mathcal{T} \cup \{S_i\}$;
  $C \leftarrow C \setminus S_i$;
end
return $\mathcal{T}$;

4.2.2 Pseudocode of Set Cover Approximation.

4.2.3 Time Complexity.

Theorem 4.1. *The time complexity of the greedy algorithm for the Minimum Set Cover problem is $O(m \cdot n)$, where $m = |\mathcal{S}|$ is the number of subsets and $n = |U|$ is the size of the universe.*

Proof. In each iteration of the while loop, the algorithm examines at most m sets to find the one that covers the maximum number of uncovered elements. For each set, calculating $|S_i \cap C|$ takes $O(n)$ time in the worst case. The while loop executes at most n times (since at least one element is covered in each iteration). Therefore, the total time complexity is $O(m \cdot n)$. □

4.2.4 Space Complexity.

Theorem 4.2. *The space complexity of the greedy algorithm for the Minimum Set Cover problem is $O(m + n)$.*

Proof. The space complexity can be analyzed as follows:

- Storage of the universe U requires $O(n)$ space.
- Storage of the collection of subsets $\mathcal{S}$ requires $O(m+\sum_{i=1}^{m} |S_i|)$ space. In the worst case, this is $O(m \cdot n)$ if each subset contains most elements.
- The solution set $\mathcal{T}$ requires at most $O(m)$ space.
- The set of uncovered elements C requires at most $O(n)$ space.

However, with efficient data structures:

- We can represent the subsets using bit vectors or hash sets for $O(1)$ lookup time for set operations.
- We can track membership of each subset in the solution using a boolean array of size m.
- We can track the coverage status of each element using a boolean array of size n.

Therefore, the overall space complexity is $O(m + n)$. □

4.2.5 Approximation Guarantee.

Theorem 4.3. *Let OPT denote the size of the optimal solution to the Minimum Set Cover problem. The greedy algorithm returns a solution of size at most $OPT \cdot H(d)$, where $d = \max_i |S_i|$ is the maximum size of any set in the collection, and $H(d) = \sum_{i=1}^{d} \frac{1}{i} \leq 1 + \ln d$ is the d-th harmonic number.*

Proof. Let C_j denote the set of elements that remain uncovered after j iterations of the greedy algorithm. Initially, $C_0 = U$. Let OPT be the size of the optimal solution, and let $\{S_1^*, S_2^*, \ldots, S_{OPT}^*\}$ be an optimal solution.

Since the optimal solution covers all elements in C_j with OPT sets, by the pigeonhole principle, there must exist a set S_i^* in the optimal solution that covers at least $\frac{|C_j|}{OPT}$ elements of C_j. That is:

$$\max_i |S_i^* \cap C_j| \geq \frac{|C_j|}{OPT}$$

The greedy algorithm selects the set S_{j+1} that maximizes $|S_{j+1} \cap C_j|$. Therefore:

$$|S_{j+1} \cap C_j| \geq \max_i |S_i^* \cap C_j| \geq \frac{|C_j|}{OPT}$$

This implies that:

$$|C_{j+1}| = |C_j| - |S_{j+1} \cap C_j| \leq |C_j| - \frac{|C_j|}{OPT} = |C_j| \cdot \left(1 - \frac{1}{OPT}\right)$$

By recursively applying this inequality:

$$|C_j| \leq |U| \cdot \left(1 - \frac{1}{OPT}\right)^j$$

The greedy algorithm terminates when $C_j = \emptyset$, which happens when $|C_j| < 1$ (since we're dealing with integers). This condition is satisfied when:

$$|U| \cdot \left(1 - \frac{1}{OPT}\right)^j < 1$$

Taking the natural logarithm of both sides:

$$\ln |U| + j \cdot \ln\left(1 - \frac{1}{OPT}\right) < 0$$

Since $\ln(1 - x) \leq -x$ for $0 < x < 1$, we have:

$$\ln |U| - j \cdot \frac{1}{OPT} < 0$$

Solving for j:

$$j > OPT \cdot \ln |U|$$

This means that the greedy algorithm terminates after at most $OPT \cdot \ln |U| + 1$ iterations. Since each iteration adds one set to the solution, the size of the greedy solution is at most $OPT \cdot \ln |U| + 1$.

Since $|U| \leq d \cdot OPT$ (as each of the OPT sets in the optimal solution covers at most d elements), we have:

$$OPT \cdot \ln |U| + 1 \leq OPT \cdot \ln(d \cdot OPT) + 1 = OPT \cdot (\ln d + \ln OPT) + 1$$

For $OPT \geq 1$, this is at most $OPT \cdot H(d)$, where $H(d)$ is the d-th harmonic number. □

4.2.6 Tightness of the Approximation Ratio.

Theorem 4.4. *The approximation ratio of $O(\log n)$ for the greedy algorithm is asymptotically tight. That is, there exist instances of the Minimum Set Cover problem for which the greedy algorithm produces a solution whose size is $\Theta(\log n)$ times the optimal solution size.*

Proof. Consider a universe $U = \{1, 2, \ldots, n\}$ where $n = 2^k - 1$ for some integer $k \geq 1$. We construct a collection of subsets $\mathcal{S}$ as follows:

1. For each $j \in \{1, 2, \ldots, k\}$, create $\binom{n}{2^{j-1}}$ sets, each containing 2^{j-1} consecutive elements from U. 2. Add one more set $S_0 = U$.

The optimal solution consists of just the set S_0, so $OPT = 1$.

The greedy algorithm, however, will first choose a set containing 2^{k-1} elements, then a set containing 2^{k-2} elements from the remaining uncovered elements, and so on. It will select exactly $k = \log_2(n + 1)$ sets.

Therefore, the ratio of the greedy solution size to the optimal solution size is $\frac{\log_2(n+1)}{1} = \Theta(\log n)$. □

4.2.7 Strengths of Greedy Relative to Local Search.

- **Theoretical Guarantees**: The greedy algorithm provides a provable approximation ratio, while many local search methods offer no such guarantees.

- **Determinism**: The greedy algorithm is deterministic, producing the same solution for a given input, which can be advantageous for reproducibility and debugging.
- **Parameter-Free**: The greedy algorithm requires no parameter tuning, unlike many local search methods that require careful calibration.
- **Efficiency**: The greedy algorithm typically requires fewer iterations than local search methods to reach a reasonable solution.

4.2.8 Weaknesses of Greedy Relative to Local Search.

- **Solution Quality**: Local search methods can often find better solutions in practice by exploring a larger portion of the solution space.
- **Flexibility**: Local search methods can be more easily adapted to handle additional constraints or objective function components.
- **Refinement Capability**: Local search methods can refine existing solutions, while the greedy algorithm builds a solution incrementally without reconsideration.
- **Hybridization**: Local search can be more easily hybridized with other metaheuristics to improve performance.

4.3 Local search algorithm I: Simulated Annealing

4.3.1 Overview. The Simulated Annealing (SA) algorithm is an optimization technique inspired by the physical process of annealing in metallurgy. It is particularly well-suited for combinatorial optimization problems, such as the Set Cover Problem, due to its ability to escape local minima and explore the search space quickly.

4.3.2 Algorithm Description. The implemented Simulated Annealing (SA) algorithm begins by constructing an initial solution using a greedy covering strategy. This initial solution is obtained by greedily attempting to select a minimal set of subsets to cover the universe. The SA algorithm then iteratively improves upon this solution through a series of perturbations and probabilistic decision-making.

In each iteration, the algorithm performs the following steps:

- **Perturbation:** The current solution is modified using the `Perturb` function. This function creates a neighboring solution by removing a random element from the current set cover solution and adding enough subsets to cover any remaining uncovered elements. The goal of this perturbation is to generate small, controlled changes to the solution, which helps the algorithm explore the search space and avoid getting stuck in local optima.
- **Acceptance Criterion:** After the perturbation, the new solution (denoted C') is compared to the current solution C. If the new solution has a lower cost, it is immediately accepted as the new solution. If the new solution is worse, it may still be accepted with a certain probability, which depends on the temperature and the difference in cost between the current and neighboring solutions. This probabilistic acceptance helps the algorithm escape local minima by allowing it to explore potentially better solutions that initially appear worse.

Algorithm 3: Simulated Annealing for Set Cover

Input: Universe U, Collection of subsets $S = S_1, \ldots, S_m$, Maximum runtime T, Initial temperature T_0, Cooling rate α

Output: Approximate set cover solution C^*

```
C ← GreedyCover(U, S);
C* ← C;
t ← 0;
D ← T_0
while time elapsed < T do
    C' ← Perturb(C)
    if |C'| < |C| then
        C ← C';
        if |C'| < |C*| then
            C* ← C';
        end
    end
    else if exp((−|C'|+|C|)/D) > rand() then
        C ← C';
    end
    D ← α · D;
end
return C*;
```

- **Temperature Decay:** The temperature is gradually reduced according to the cooling rate α. As the temperature decreases, the likelihood of accepting worse solutions decreases, allowing the algorithm to focus more on refining the current solution. The temperature decay controls the trade-off between exploration and exploitation, with high temperatures encouraging exploration and low temperatures favoring exploitation.

The perturbation function is critical to this algorithm as it introduces diversity in the search space. Relative to a basic hill climbing local search algorithm, this means we might sacrifice some convergence speed to have a much better chance of avoiding local optima and converging to a better solution in the end.

4.3.3 Approximation Guarantee. Simulated Annealing does not offer a formal approximation guarantee for the SCP, unlike greedy algorithms which yield a $\ln n$-approximation. Because of the random nature of the algorithm, and the importance of the temperature parameters, it is entirely possible that the algorithm could end up stuck chasing suboptimal solutions. Especially in problems with extreme cases of local optima, if the cooling occurs too quickly the algorithm could return a result far from the optimal solution. However, in practice, Simulated Annealing can outperform greedy approaches by moving towards better solutions while leveraging randomness to explore different regions of the solution space.

4.3.4 Time and Space Complexity. To assess the practicality of Simulated Annealing for the Set Cover Problem, we examine both time and space requirements under reasonable assumptions about input size and runtime constraints.

Time Complexity. In this local search algorithm, we have no guarantees of convergence and simply run until a cut-off time, and as such our time complexity is dependent on how long we choose to run the algorithm. Let $n = |U|$ be the size of the universe and $m = |S|$ be the number of subsets. Each iteration of the algorithm involves generating a neighboring solution via perturbation, which consists of:

- Removing one random subset from the current solution ($O(1)$);
- Recomputing the set of uncovered elements ($O(n)$, assuming sparse sets);
- Iteratively selecting subsets that maximize coverage of the remaining uncovered elements, which involves scanning all m subsets and computing intersections ($O(mn)$ in the worst case).

If we assume the number of iterations is bounded by a limit T, and considering perturbation dominates the per-iteration cost, the overall time complexity becomes:

$$O(T \cdot m \cdot n)$$

This is a pessimistic upper bound. In practice, the number of uncovered elements after removing one subset is often small, and intersection checks are efficiently handled using set operations, resulting in significantly better empirical performance.

Space Complexity. The algorithm maintains the following data structures:

- The set of subsets S, of size $O(m \cdot s)$, where s is the average subset size (often much less than n);
- One or two current solutions (lists of indices), of size $O(m)$;
- Sets representing the covered and uncovered elements, each requiring $O(n)$ space.

Therefore, the total space complexity is:

$$O(m \cdot s + n)$$

which is generally manageable for moderate values of m and n.

4.3.5 Design Rationale and Tuning. The Simulated Annealing algorithm was chosen for its ability to probabilistically escape local optima, which is a critical limitation in greedy and deterministic approaches to the Set Cover Problem. The algorithm begins with a greedy solution to ensure a solid baseline, then introduces randomness through perturbation to explore alternative regions of the solution space.

The perturbation mechanism removes a randomly chosen subset from the current solution, creating a temporary coverage gap. This gap is then greedily filled by re-selecting subsets that cover the most remaining uncovered elements, effectively re-optimizing part of the solution. This hybrid of randomness and greedy reconstruction helps maintain feasibility while allowing variation.

Parameters were manually tuned based on empirical performance across a variety of instance sizes. A cooling rate of $\alpha = 0.99$ and starting temperature $T_0 = 100.0$ provided a good balance between exploration and convergence. No automated tuning framework was used, but systematic testing on smaller instances helped guide parameter choices.

4.3.6 Strengths and Weaknesses.

Strengths:

- *Escapes local optima:* Through probabilistic acceptance of worse solutions, SA can overcome poor local minima that trap deterministic algorithms.
- *Adaptability:* SA can be applied to many combinatorial problems without problem-specific adaptations.
- *Flexible solution quality-time tradeoff:* By adjusting runtime and cooling schedule, users can balance between fast approximations and higher solution quality.

Weaknesses:

- *No worst-case guarantee:* Unlike greedy algorithms, SA offers no formal bounds on approximation quality, making its performance problem-dependent.
- *Parameter sensitivity:* Solution quality can vary significantly depending on temperature schedule and time budget.
- *Potentially slow convergence:* In some cases, especially large instances with small improvements, SA may converge slowly or get stuck oscillating around suboptimal solutions.

4.4 Local search algorithm II: Hill Climbing

4.4.1 Algorithm Description. We implemented a randomized local search algorithm tailored for the MSC problem. This heuristic approach incorporates randomness in both its initialization and iterative improvement phases, drawing conceptual inspiration from dynamics such as random walks on hypergraphs [4] to navigate the solution space.

The algorithm begins by constructing an initial feasible cover using a *Random Cover* method. Subsets from the collection $\mathcal{S}$ are selected uniformly at random and added to the solution until all elements in the universe U are covered. This randomized initialization serves to diversify the starting points of the search, aiming to mitigate the risk of prematurely converging to suboptimal solutions often encountered by deterministic greedy approaches.

Following initialization, the algorithm enters an iterative local search phase to enhance the current solution. Each iteration involves several steps. First, a subset S_i is chosen randomly from the current cover $\mathcal{T}$. The algorithm then explores the neighborhood of S_i by identifying potential replacement candidates S_j from $\mathcal{S} \setminus \mathcal{T}$. A subset S_j qualifies as a neighbor if it shares at least one element with S_i. A specific neighbor S_j is selected based on certain criteria (the original description mentioned favoring larger neighbors, but other criteria could be employed). This selected S_j then replaces S_i in the solution. Subsequently, a crucial pruning step is performed: the algorithm checks the modified solution for any redundant subsets. A subset is deemed redundant if all elements it covers are also covered by at least one other subset in the current solution. Such redundant subsets are removed to ensure the cover remains minimal with respect to subset inclusion and potentially reduce its size (cardinality).

To guide the search and prevent cycling, a *tabu set* is maintained, temporarily prohibiting the re-selection of recently modified (e.g., removed) subsets. Additionally, a stagnation counter tracks consecutive iterations without improvement in the best solution size found so far. If this counter exceeds a predefined threshold, the search may terminate early, or trigger a diversification mechanism

Algorithm 4: Hill Climbing for Minimum Set Cover

Input: Universe $U = \{x_1, \ldots, x_n\}$, collection of subsets $\mathcal{S} = \{S_1, \ldots, S_m\}$, cutoff time T
Output: A minimal set cover $\mathcal{T} \subseteq \mathcal{S}$

Initialize $\mathcal{T} \leftarrow$ RandomCover$(U, \mathcal{S})$
Construct count table C recording coverage frequency of each $x \in U$
Initialize tabuSet $\leftarrow \emptyset$
Set $S_i \in \mathcal{T}$ randomly as current candidate
Initialize noImprovementCount $\leftarrow 0$
while *time elapsed* $< T$ **do**
  Compute neighbors of S_i not in $\mathcal{T}$ that share elements with S_i
  Sort neighbors S_j in increasing order of $|S_j|$
  for *each* S_j *not in* tabuSet **do**
    Simulate replacing S_i with S_j in $\mathcal{T}$
    Update count table C
    if *all* $x \in U$ *still covered in* C **then**
      $\mathcal{T} \leftarrow (\mathcal{T} \setminus \{S_i\}) \cup \{S_j\}$
      Add S_i, S_j to tabuSet
      $S_i \leftarrow S_j$
      while *some* $S_k \in \mathcal{T}$ *is redundant* **do**
        Remove S_k from $\mathcal{T}$ and update C
      end
      Reset noImprovementCount to 0
      break
    end
  end
  if *no replacement succeeded* **then**
    Increment noImprovementCount
    Reset tabuSet
    Pick new $S_i \in \mathcal{T}$ randomly
  end
  if noImprovementCount $= min(|\mathcal{T}|, 100)$ **then**
    break
  end
end
return $\mathcal{T}$

like a restart. The algorithm concludes when either a specified time limit is reached or the stagnation condition is met, returning the best cover found during the search. While optimality is not guaranteed, this randomized local search approach is generally simple, scales reasonably well, and often demonstrates strong empirical performance, particularly on large or sparse MSC problem instances.

4.4.2 Approximation Guarantee. This randomized local search algorithm, like most local search heuristics for NP-hard problems such as MSC problem, does not come with a formal approximation guarantee. The quality of the final solution relative to the true optimum is not bounded by a theoretical factor. Its performance is empirically driven and depends significantly on factors like the instance characteristics, the specific implementation of neighborhood exploration and selection, the effectiveness of the randomization, the parameter settings (tabu tenure, stagnation limits), and the computational budget (time limit). While it might potentially outperform algorithms with known guarantees (like the standard greedy $O(\log n)$-approximation) on certain instances, its worst-case performance is unknown.

4.4.3 Complexity Analysis. Let $n = |U|$ be the number of elements in the universe and $m = |\mathcal{S}|$ be the total number of available subsets. Let $s_{max} = \max_{S \in \mathcal{S}} |S|$ be the maximum size of any subset, and s_{avg} be the average subset size. Let $N = \sum_{S \in \mathcal{S}} |S|$ be the total size of the input representation of subsets. Let k denote the number of subsets in the current solution $\mathcal{T}$ (typically $k \ll m$). Let d_{avg} be the average degree of an element (number of sets it belongs to), and d_{max} be the maximum degree.

Time Complexity:

- **Initialization (Random Cover):** Building the initial cover requires selecting subsets randomly until all n elements are covered. Efficient tracking of covered elements (e.g., using a boolean array or hash set and coverage counts) takes roughly $O(n)$ space. Adding a set S and updating counts takes $O(|S|)$ time. If k_{init} subsets are added, the time is roughly proportional to the sum of their sizes, $O(\sum_{i=1}^{k_{init}} |S_i|)$. In the worst case, k_{init} could be large. A loose upper bound considering potentially many selections could be $O(m \cdot s_{max})$ or $O(k_{init} \cdot s_{max})$. The initial pruning step using 'PruneRedundant' adds complexity, see below. The $O(nm)$ bound mentioned earlier might arise from less efficient coverage checking or worst-case scenarios where nearly all sets must be examined multiple times. A more typical estimate focusing on the work done is $O(k_{init} \cdot s_{avg} + n)$ plus pruning time.
- **Each Local Search Iteration:**
  - *Selecting S_i*: $O(1)$ if $\mathcal{T}_{current}$ is an indexed structure.
  - *Finding Neighbors S_j*: Requires identifying subsets $S \notin \mathcal{T}_{current}$ sharing elements with S_i. If pre-computed element-to-subset indices exist (taking $O(N)$ preprocessing time and space), this involves looking up sets for each $u \in S_i$. The cost is roughly $\sum_{u \in S_i} O(d_u) \approx O(|S_i| \cdot d_{avg})$, bounded by $O(s_{max} \cdot d_{max})$. Filtering non-tabu neighbors adds minor overhead. Without pre-computation, it might require checking against many subsets in $\mathcal{S}$, potentially $O(m \cdot s_{max})$.
  - *Replacement*: Modifying the solution representation is typically $O(1)$.
  - *PruneRedundant Function*: Calculating initial coverage counts for the current solution $\mathcal{T}'$ takes $O(\sum_{S \in \mathcal{T}'} |S|) = O(k \cdot s_{avg})$, bounded by $O(k \cdot s_{max})$. Checking each of the k' sets S_k in $\mathcal{T}'$ for redundancy takes $O(|S_k|)$ time using the counts. If a set is removed, updating counts takes $O(|S_k|)$. The total time for pruning is dominated by the initial count calculation and the sum of checks/updates, roughly $O(k \cdot s_{max})$.
  - *Tabu List Update*: Usually $O(1)$ or $O(\log(\text{TabuSize}))$ depending on implementation.

Assuming efficient neighbor finding (with preprocessing), the complexity of one iteration is dominated by neighbor identification and redundancy pruning: approximately $O(s_{max} \cdot d_{avg} + k \cdot s_{max})$. If k is significantly smaller than d_{avg}, the neighbor finding might dominate, otherwise pruning might.
- **Total Time:** The algorithm runs for a number of iterations constrained by T_{limit} and $MaxStagnation$. The total runtime is the number of iterations multiplied by the average time per iteration. The controlled termination ensures predictable runtime behavior.

Space Complexity:

- Storing the input (U and $\mathcal{S}$): Requires $O(n + N)$ space. Precomputed element-to-subset mappings add $O(N)$ space.
- Storing the current solution $\mathcal{T}_{current}$ and best solution $\mathcal{T}_{best}$: $O(k)$ space for subset references.
- Element coverage counts (for pruning and initialization): $O(n)$ space.
- Tabu Set: Space proportional to tabu tenure, O(TabuSize).

The overall space complexity is dominated by the input representation and auxiliary structures like indices and counts, typically $O(N + n)$.

4.4.4 Strengths and Weaknesses. The primary strength of this randomized local search approach lies in its potential to find high-quality solutions for MSC problem, often surpassing simple greedy methods by effectively escaping local optima through randomized initialization and neighborhood exploration. Its core mechanics are relatively straightforward to implement. The algorithm exhibits good scalability, especially on instances where subsets and element memberships are sparse (s_{avg}, d_{avg} are small). Furthermore, it functions as an anytime algorithm: it produces a valid solution quickly and refines it over time, making it suitable for scenarios with strict time constraints, as it can be terminated early while providing the best solution discovered up to that point. Its flexibility allows for adaptation by modifying neighborhood definitions, selection heuristics, or incorporating other metaheuristic components.

However, the approach has notable weaknesses. Critically, it offers no guarantee of finding the optimal solution, nor does it provide a theoretical bound on the solution quality relative to the optimum (approximation ratio). The performance can be sensitive to the choice of parameters, such as the tabu tenure length and the stagnation threshold, often requiring careful tuning for specific problem classes. Due to its stochastic nature, multiple runs on the same instance might yield different results, introducing variability in solution quality and runtime to convergence. While the overall runtime is bounded, the time needed to reach a near-optimal or satisfactory solution can be unpredictable.

4.4.5 Automated Tuning and Configuration Considerations. While formal automated tuning frameworks (like irace or SMAC) were not explicitly detailed as used in the provided context, the configuration of such a local search algorithm often involves significant manual tuning and experimentation. Key considerations during this phase would include setting the tabu tenure and the stagnation limit. For instance, the stagnation parameter might trigger diversification strategies, such as random restarts from the best-known solution perturbed slightly, or even a full restart using the 'RandomCoverInitialization' again, conceptually similar to how Random Walk with Restart periodically returns to source nodes to maintain focus while exploring.

Another important configuration choice involves the neighbor selection strategy within the local search step. Instead of always selecting a random valid neighbor, heuristics could be employed. For example, one might prioritize selecting a neighbor S_j that is larger than the removed set S_i, or has a similar size. This could be motivated by the goal of finding a minimum cardinality cover, potentially making the pruning step more effective. These heuristic choices represent manual configuration decisions aimed at balancing exploration, exploitation, and computational efficiency, often refined through empirical evaluation on benchmark instances.

4.4.6 Design Choices and Justification. The choice of a *Random Cover Initialization* instead of a deterministic greedy start was made to promote diversity and explore different regions of the search space from the outset, aiming to avoid the potential biases of greedy choices that might lead directly to poor local optima. This aligns with the inspiration cited from random processes on hypergraphs [4]. Defining the neighborhood based on *shared elements* between the removed set S_i and candidate replacement S_j provides a localized and relevant way to explore alternatives while maintaining some structural similarity.

The core search operators combine *replacement* (swapping S_i for S_j) for exploration and potential improvement, with *redundancy pruning* for intensification and ensuring solution minimality (in terms of subset inclusion). This combination is standard and effective for set covering heuristics. The *random selection of* S_i from the current cover adds another layer of stochasticity, preventing deterministic cycling and encouraging broader exploration of removal possibilities. The inclusion of a *Tabu List* is a standard technique to prevent immediate reversal of moves and avoid short-term cycling, thereby pushing the search towards new areas. Finally, using a *stagnation counter* provides a practical termination condition based on search progress, complementing the overall time limit and allowing for early stopping or triggering diversification mechanisms when the search appears to be stuck.

5 EXPERIMENTS

5.1 Datasets

We evaluate our algorithms on a standardized dataset provided with the course assignment. The dataset is divided into three sections: test, small, and large instances. All files follow a consistent format and are used to assess correctness, scalability, and solution quality.

- **Test instances** (`test#.in`, `test#.out`): These include both the input data and the optimal set of selected subsets. They are used to validate the correctness of our algorithms.
- **Small instances** (`small#.in`, `small#.out`): These have up to 20 elements and 20 subsets. Only the optimal value (minimum cover size) is provided, allowing us to assess the relative accuracy of our solutions.
- **Large instances** (`large#.in`, `large#.out`): These scale up to 2000 elements and 2000 subsets. Like the small instances,

only the optimal value is given. These instances are used to test algorithmic scalability and runtime performance.

Each input file begins with two integers, n and m, representing the number of elements in the universe and the number of subsets, respectively. Each subsequent line describes a subset: the first integer denotes its size, followed by the elements it contains.

5.2 Experimental setting

All experiments were conducted on a MacBook Pro featuring Apple's M1 Pro system-on-chip. The machine is equipped with a 10-core CPU (8 performance cores and 2 efficiency cores), along with 16 GB of unified memory. The software environment was set up as follows:

- **Language:** Python 3.12.17
- **Compiler/Interpreter:** CPython 3.12.17
- **Libraries:**
  - `NumPy` for array operations
  - `SciPy` (v1.12+) for linear programming (LP-relaxation)
  - `time` and `math` for timing and utility functions

For solving LP relaxations at each node of the Branch-and-Bound (BnB) algorithm, we used `scipy.optimize.linprog` with the `HiGHS` solver backend.

We evaluated all algorithms on three types of problem instances: *small*, *large*, and *test* benchmarks. Each algorithm was given a specified time budget per instance:

- **Branch-and-Bound (BnB):** Executed with a maximum cutoff time of 600 seconds per instance. This generous budget allowed for deep exploration of the solution tree and comprehensive use of LP-based pruning.
- **Local Search Algorithms (LS1, LS2):** Each local search algorithm was run with a 30-second cutoff per instance. Due to their randomized nature, we conducted 20 independent runs per instance.
- **Approximation:** This deterministic algorithm was run once per instance with a nominal 30-second limit, though in practice it terminated well within that time due to its greedy and efficient structure.

5.3 Experimental Results

We evaluated all four algorithms—Branch-and-Bound (BnB), Greedy Approximation (Approx), Simulated Annealing (LS1), and Hill Climbing (LS2)—across the complete benchmark suite of small and large problem instances. Table 1 through Table 4 in **Appendix** summarize the average performance of each algorithm, reporting runtime, solution size, and relative error with respect to known optima. Each local search algorithm was run **10 times** with different random seeds, results were averaged for the table.

BnB consistently achieved the optimal solution on all small instances and most large instances, but its runtime increased significantly with problem size. In contrast, the Approximation algorithm achieved the lowest runtimes across all instances, demonstrating strong computational efficiency. This performance gain, however, was accompanied by a degradation in solution quality, as reflected in its higher relative error rates. Empirical results for the Approx algorithm, summarized in Table 5, are consistent with the theoretical approximation bound established earlier. The maximum observed approximation ratio was 1.8105, calculated using the bound proven earlier, with an average of 1.3431 across all instances, which indicates that the approximation algorithm often performs better than its theoretical bound. Both LS1 and LS2 struck a balance between runtime and solution quality. LS1 generally outperformed LS2 in terms of solution accuracy, while LS2 demonstrated slightly better average runtime. For large instances such as `large1` and `large10`, LS1 achieved near-optimal solutions within a fraction of the time required by BnB.

Overall, BnB provides a gold standard for small and moderate-sized problems. Approx is effective as a fast baseline or initializer. LS1 and LS2 offer practical trade-offs for large-scale scenarios where optimality is less critical than runtime.

5.3.1 Role of lower bound on the quality of solution. The approximation algorithm provides an empirical means of estimating a lower bound on the optimal solution quality. By dividing the greedy solution size by the observed approximation ratio, we can derive a lower bound: for instance, on `large1`, the greedy solution is 83 and the approximation ratio is 1.66, yielding a lower bound of approximately $83/1.66 \approx 50$, which matches the known optimum. Across all instances, the average approximation ratio was around 1.34, and the worst-case was 1.81. These ratios suggest that the approximation algorithm, despite lacking guarantees on optimality, consistently yields solutions that are within 20–80% of the optimum. Thus, in cases where the true optimum is unknown, this ratio gives a practical and reasonably tight lower bound estimate.

The branch-and-bound algorithm explicitly computes a lower bound using LP-relaxation at each node in the search tree. For instances where the algorithm terminates within the time budget and finds the optimal solution, this bound equals the true optimum. In incomplete runs (e.g., `large1`), the best solution found provides an upper bound, while the LP-relaxation serves as a formal lower bound. These bounds are typically much tighter than those inferred from the approximation ratio. In practice, even when the full tree is not explored, BnB often produces high-quality solutions with smaller optimality gaps, and the LP-derived lower bounds offer more rigorous evidence for how close the current solution is to optimal.

Maximum approximation ratio: 1.8105
Average approximation ratio: 1.3431
Theoretical approximation guarantee: $H_n \leq \ln(n) + 0.57721\ldots$
Empirical lower bound on solution quality: $0.5523 \cdot \text{OPT}$

5.4 QRTDs and SQDs Plot

Figures 1 and 2 present the *Qualified Runtime Distribution (QRTD)* and the *Solution Quality Distribution (SQD)* for the `large1` and `large10` instances, focusing on the LS1 and LS2 algorithms.

For `large1`, the QRTD (left) indicates that LS1 reaches high-quality solutions more quickly and in a greater fraction of runs compared to LS2, particularly for a relative error threshold of $q^* = 0.25$. The corresponding SQD (right) confirms this trend, showing that LS1 produces more consistent and accurate final solutions than LS2.

For `large10`, the QRTD shows neither LS1 nor LS2 were able to cross the relative error threshold and achieve good solutions. The SQD plot shows LS1 had slightly better performance on this instance, with the LS2 algorithm having more variation in solution quality.

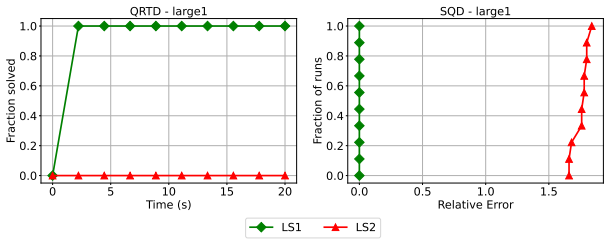

Figure 1: QRTD (left) and SQD (right) for LS1 and LS2 on `large1`.

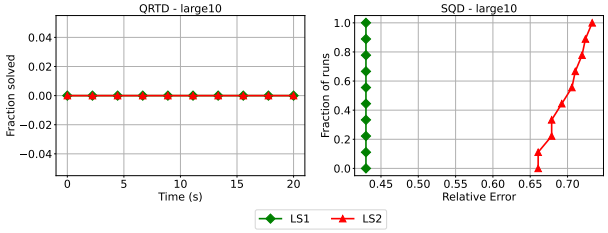

Figure 2: QRTD (left) and SQD (right) for LS1 and LS2 on `large10`.

5.5 Runtime Variability

Figures 3 and 4 show box plots of the running times for each local search algorithm on the `large1` and `large10` instances, based on **20 runs**. LS1 exhibits relatively low variance, while LS2 displays greater variability. This difference stems from their initialization strategies: LS1 begins with a greedy set cover, resulting in more consistent performance, whereas LS2 starts from a random set cover, leading to broader variation between trials. Additionally, LS2 incorporates a random walk with restart mechanism when no improvement is found. Although LS1 outperformed LS2 on `large1`, LS2 was faster on `large10`, indicating that greedy initialization does not always guarantee better runtime performance.

6 DISCUSSION

Branch-and-Bound (BnB) consistently produced optimal solutions for both small and most large instances, establishing itself as the accuracy benchmark. However, this precision came at a computational cost: runtime increased rapidly with instance size due to its exhaustive search nature. Although BnB employs LP-relaxation to derive tight lower bounds and prune the search space, its scalability is limited, especially when full tree exploration becomes infeasible within a time budget.

Regarding Greedy Approximation, our experimental results reveal a significant gap between theoretical and practical performance

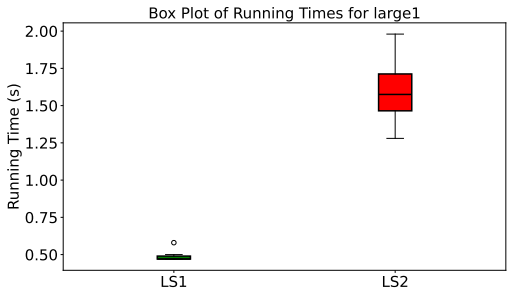

Figure 3: Box plot of running times for LS1 and LS2 on `large1`.

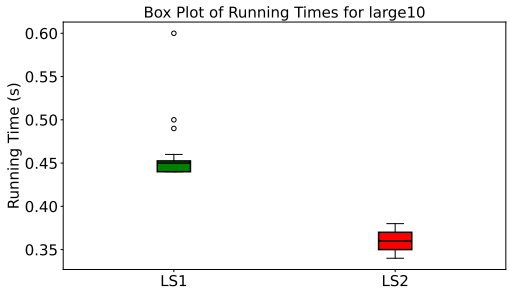

Figure 4: Box plot of running times for LS1 and LS2 on `large1`.

of the greedy algorithm for Set Cover. While the theoretical approximation guarantee ($H_n \approx \ln(n) + 0.57721$) suggests ratios between 1.68 and 8.18 for our instances, the observed performance is substantially better, with a maximum ratio of only 1.81 and an average of 1.34. This translates to an empirical lower bound of $0.5523 \times \text{OPT}$, indicating solutions at least 55% as good as optimal in practice. The largest approximation ratios (1.81, 1.68, 1.66) occur on the largest problem instances (large7, large4, large1), showing the expected correlation between instance size and approximation quality, though the performance degradation is much slower than theoretical bounds suggest. These findings show the benefits of using the greedy approach in practical applications, as it delivers near-optimal results with polynomial time complexity, making it a good choice for instances where exact optimality is not required or when initializing more time consuming algorithms.

The local search algorithms (LS1 and LS2) strike a middle ground between runtime and solution quality, with performance closely tied to their initialization and exploration strategies. LS1, initialized with a greedy solution, exhibited low variance across trials and consistently achieved near-optimal results in many large instances. Its structured starting point constrained the search space, making it more predictable and robust. On the other hand, LS2, which starts from a random set cover and employs a random walk with restart when progress stalls, demonstrated higher variance but often found competitive solutions faster. This stochasticity enables broader exploration but introduces performance fluctuations, especially on problem instances where the greedy heuristic aligns poorly with the optimal structure.

In summary, each algorithm offers a distinct trade-off. BnB guarantees optimality but is computationally expensive. Approx is fast but less accurate. LS1 is more reliable due to deterministic initialization, while LS2 leverages randomness for adaptability at the cost of stability. The choice of algorithm should therefore be guided by the problem scale and the relative importance of runtime versus solution quality.

7 CONCLUSION

In this work, we present a comprehensive evaluation of four algorithms for the Minimum Set Cover problem: an exact Branch-and-Bound (BnB) approach, a classical Greedy Approximation algorithm, and two local search heuristics—Simulated Annealing (LS1) and Hill Climbing with restarts (LS2). Our BnB implementation leverages LP-relaxation and bounding strategies for efficient pruning, while the Approximation algorithm provides fast, provably good solutions. The local search algorithms are designed for practical scenarios where solution quality and runtime must be balanced, with LS1 using a greedy initializer and LS2 exploring the search space more broadly via randomness and restarts.

Experimental results across real-world benchmarks demonstrate clear trade-offs. BnB ensures optimality but suffers from poor scalability. Approx achieves near-optimal solutions much faster than theory suggests. LS1 consistently yields high-quality solutions with low variance due to its deterministic start, while LS2 converges faster on certain instances by embracing stochasticity, albeit with more variability. Overall, our study highlights the practical strengths of greedy and heuristic methods for large-scale problems and offers actionable insights for selecting algorithms based on problem size, time constraints, and solution quality needs. All code and data are open-sourced to support reproducibility and future research at github.com/hhyy0401/CSE6140.

REFERENCES

[1] Václav Chvátal. 1979. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research* 4, 3 (1979), 233–235.
[2] Refael Hassin and Asaf Levin. 2005. A better-than-greedy approximation algorithm for the minimum set cover problem. *SIAM J. Comput.* 35, 1 (2005), 189–200.
[3] Richard M Karp. 2009. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*. Springer, 219–241.
[4] Hyunju Kim, Heechan Moon, Fanchen Bu, Jihoon Ko, and Kijung Shin. 2025. Estimating Simplet Counts via Sampling. *The VLDB Journal* 34, 2 (2025), 1–26. https://doi.org/10.1007/s00778-024-00890-9
[5] Rishi Saket and Maxim Sviridenko. 2012. New and improved bounds for the minimum set cover problem. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 288–299.

8 Appendix

Table 1: Comprehensive Table of BnB Algorithm

Instance	Time (s)	Collection Size	Rel. Error
large1*	0.28	83.00	0.66
large2	282.56	19.00	0.00
large3	335.97	15.00	0.00
large4	0.17	153.00	0.68
large5	0.04	7.00	0.17
large6	0.00	7.00	0.17
large7	0.36	172.00	0.81
large8	0.00	6.00	0.20
large9	0.00	16.00	0.14
large10	0.68	319.00	0.44
large11	0.08	56.00	0.40
large12	0.00	18.00	0.20
small1	0.00	5.00	0.00
small2	0.00	3.00	0.00
small3	0.00	5.00	0.00
small4	0.00	4.00	0.00
small5	0.00	5.00	0.00
small6	0.00	3.00	0.00
small7	0.00	3.00	0.00
small8	0.00	2.00	0.00
small9	0.00	3.00	0.00
small10	0.00	2.00	0.00
small11	0.00	4.00	0.00
small12	0.00	3.00	0.00
small13	0.00	2.00	0.00
small14	0.00	2.00	0.00
small15	0.00	2.00	0.00
small16	0.00	2.00	0.00
small17	0.01	2.00	0.00
small18	0.00	2.00	0.00

Table 2: Comprehensive Table of Approx Algorithm

Instance	Time (s)	Collection Size	Rel. Error
large1	0.29	83.00	0.66
large2	0.00	21.00	0.11
large3	0.00	17.00	0.13
large4	0.16	153.00	0.68
large5	0.00	8.00	0.33
large6	0.00	7.00	0.17
large7	0.37	172.00	0.81
large8	0.00	6.00	0.20
large9	0.00	16.00	0.14
large10	0.68	319.00	0.44
large11	0.07	56.00	0.40
large12	0.00	18.00	0.20
small1	0.00	5.00	0.00
small2	0.00	4.00	0.33
small3	0.00	6.00	0.20
small4	0.00	5.00	0.25
small5	0.00	6.00	0.20
small6	0.00	4.00	0.33
small7	0.00	4.00	0.33
small8	0.00	3.00	0.50
small9	0.00	4.00	0.33
small10	0.00	3.00	0.50
small11	0.00	5.00	0.25
small12	0.00	4.00	0.33
small13	0.00	3.00	0.50
small14	0.00	3.00	0.50
small15	0.00	3.00	0.50
small16	0.00	3.00	0.50
small17	0.00	3.00	0.50
small18	0.00	3.00	0.50

Table 3: Comprehensive Table of LS1 Algorithm

Instance	Time (s)	Collection Size	Rel. Error
large1	0.47	50.00	0.00
large2	0.00	20.00	0.05
large3	0.00	17.00	0.13
large4	0.15	152.00	0.67
large5	0.00	8.00	0.33
large6	0.00	7.00	0.17
large7	0.00	172.00	0.81
large8	0.00	6.00	0.20
large9	0.00	16.00	0.14
large10	0.44	316.00	0.43
large11	0.00	56.00	0.40
large12	0.00	18.00	0.20
small1	0.00	5.00	0.00
small2	0.00	4.00	0.33
small3	0.00	6.00	0.20
small4	0.00	5.00	0.25
small5	0.00	5.00	0.00
small6	0.00	4.00	0.33
small7	0.00	4.00	0.33
small8	0.00	3.00	0.50
small9	0.00	4.00	0.33
small10	0.00	3.00	0.50
small11	0.00	5.00	0.25
small12	0.00	4.00	0.33
small13	0.00	3.00	0.50
small14	0.00	3.00	0.50
small15	0.00	3.00	0.50
small16	0.00	3.00	0.50
small17	0.00	3.00	0.50
small18	0.00	3.00	0.50

Table 4: Comprehensive Table of LS2 Algorithm

Instance	Time (s)	Collection Size	Rel. Error
large1	1.58	137.60	1.75
large2	0.01	23.30	0.23
large3	0.03	19.20	0.28
large4	0.22	183.30	1.01
large5	0.01	8.50	0.42
large6	0.03	7.90	0.32
large7	0.58	220.70	1.32
large8	0.00	6.10	0.22
large9	0.03	19.00	0.36
large10	0.37	374.90	0.70
large11	0.20	70.50	0.76
large12	0.02	20.10	0.34
small1	0.00	5.50	0.10
small2	0.00	3.10	0.03
small3	0.00	5.00	0.00
small4	0.00	4.00	0.00
small5	0.00	5.50	0.10
small6	0.00	3.30	0.10
small7	0.00	3.60	0.20
small8	0.00	2.30	0.15
small9	0.00	3.10	0.03
small10	0.00	2.20	0.10
small11	0.00	4.20	0.05
small12	0.00	3.20	0.07
small13	0.00	2.30	0.15
small14	0.00	2.10	0.05
small15	0.00	2.90	0.45
small16	0.00	3.00	0.50
small17	0.00	2.90	0.45
small18	0.00	2.10	0.05

Table 5: Approximation: Output Versus Theoretical Bound

Instance	n	m	Optimal	Greedy	Ratio	Theoretical Bound
large1	2000	2000	50	83	1.66	8.1781
large10	1547	2000	221	319	1.44	7.9213
large11	400	2000	40	56	1.40	6.5687
large12	100	500	15	18	1.20	5.1824
large2	100	200	19	21	1.11	5.1824
large3	100	1000	15	17	1.13	5.1824
large4	1183	1183	91	153	1.68	7.6530
large5	100	500	6	8	1.33	5.1824
large6	100	1000	6	7	1.17	5.1824
large7	1805	1805	95	172	1.81	8.0755
large8	50	500	5	6	1.20	4.4892
large9	100	1000	14	16	1.14	5.1824
small1	20	20	5	5	1.00	3.5729
small10	7	15	2	3	1.50	2.5231
small11	20	20	4	5	1.25	3.5729
small12	10	20	3	4	1.33	2.8798
small13	10	10	2	3	1.50	2.8798
small14	5	10	2	3	1.50	2.1866
small15	15	15	2	3	1.50	3.2853
small16	7	15	2	3	1.50	2.5231
small17	20	20	2	3	1.50	3.5729
small18	10	20	2	3	1.50	2.8798
small2	5	10	3	4	1.33	2.1866
small3	15	15	5	6	1.20	3.2853
small4	7	15	4	5	1.25	2.5231
small5	20	20	5	6	1.20	3.5729
small6	10	20	3	4	1.33	2.8798
small7	10	10	3	4	1.33	2.8798
small8	5	10	2	3	1.50	2.1866
small9	15	15	3	4	1.33	3.2853
test1	3	3	2	2	1.00	1.6758
test2	5	7	2	3	1.50	2.1866
test3	10	10	6	7	1.17	2.8798
test4	10	5	4	5	1.25	2.8798
test5	7	10	4	5	1.25	2.5231