Exact and Approximation Algorithms for the Minimum Set Cover Problem

Saketh Chaluvadi
schaluvadi@gatech.edu
Georgia Institute of Technology
School of Computer Science
Atlanta, GA, USA

Adityaa Magesh Kumar
akumar773@gatech.edu
Georgia Institute of Technology
School of Computer Science
Atlanta, GA, USA

Arjun Tammishetti
atammishetti3@gatech.edu
Georgia Institute of Technology
School of Computer Science
Atlanta, GA, USA

Abstract

The Minimum Set Cover (MSC) problem is a fundamental NP-complete problem with broad applications in optimization and computer science. In this project, we implement and evaluate several algorithms for solving MSC, including a greedy approximation algorithm, a branch-and-bound (BnB) exact solver, and local search variants such as simulated annealing and hill climbing. Our evaluation uses benchmark datasets of varying sizes to assess solution quality and runtime behavior. Results show that while the greedy algorithm performs surprisingly well across many instances—often producing near-optimal covers—local search methods offer meaningful trade-offs between accuracy and performance. BnB guarantees optimality but suffers from scalability issues on large instances. We discuss these algorithms and their trade-offs and practical implications.

Keywords

Set Cover, Branch and Bound, Greedy Algorithm, Approximation Algorithms, Local Search, Simulated Annealing, Hill Climbing

ACM Reference Format:
Saketh Chaluvadi, Adityaa Magesh Kumar, and Arjun Tammishetti. 2025. Exact and Approximation Algorithms for the Minimum Set Cover Problem. In *Proceedings of CSE 6140: Algorithms (CSE 6140 Project Report)*. ACM, New York, NY, USA, 15 pages. https://doi.org/https://project-report-placeholder.doi

1 Introduction

The Minimum Set Cover (MSC) problem is a well-known NP-Complete problem and was originally referenced in Karp's original 21 NP-Complete problems [9]. Given a universe of elements and a family of subsets, the goal is to find the smallest collection of subsets that together cover the entire universe or contain all the elements in the universe.

This project investigates the various approaches of solving the MSC problem, including an exact branch-and-bound algorithm, a greedy approximation algorithm, and heuristic algorithms without approximation guarantees.

2 Problem Description

Given a universe $U = \{x_1, x_2, \ldots, x_n\}$ and a collection of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ of U (i.e., $S_i \subseteq U$ for all $i = 1, \ldots, m$), the objective is to select a minimum-cardinality subfamily $C \subseteq \mathcal{S}$ such that

$$\bigcup_{S \in C} S = U.$$

3 Related Work

The Minimum Set Cover (MSC) problem is NP-hard, requiring exponential time to solve in the worst case. Classical exact algorithms for MSC are primarily based on branch-and-bound search. Balas and Carrera developed a branch-and-bound procedure that uses subgradient optimization to compute tighter lower bounds, enabling more effective pruning of the search tree [1]. Modern exact methods may incorporate cutting planes (branch-and-cut) or column generation techniques to further tighten bounds, but these approaches remain limited to medium-sized instances due to the problem's inherent complexity.

Due to the exponential time required by exact algorithms, a simple greedy approach was proposed as an approximation method. This algorithm repeatedly selects the set that covers the largest number of uncovered elements, yielding an H_n-approximation, where H_n is the nth harmonic number. This ensures a solution that uses at most $O(\log n)$ times more sets than the optimum. Johnson first analyzed the greedy algorithm's performance for the unweighted case [8], and Chvátal extended the analysis to the weighted version [4]. Their results show that the greedy algorithm achieves the best possible approximation ratio, up to constant factors. Furthermore, Feige proved that no polynomial-time algorithm can achieve a better approximation than $\ln n$, unless NP-complete problems can be solved in sub-exponential time [6].

In practice, heuristic and local search methods are often preferred, as exact algorithms are too slow for large instances. Beasley introduced a Lagrangian-relaxation heuristic that solves a relaxed version of the problem and then rounds or repairs the solution to ensure feasibility [2]. Local search methods have also proven effective: Jacobs and Brusco [7] showed that an initial solution (e.g., generated by the greedy algorithm) can be substantially improved by iteratively adding and removing sets to reduce the overall cover size.

Other heuristics have also been applied to the MSC problem. A notable example is the genetic algorithm developed by Beasley and Chu [3], which evolves a population of solutions using problem-specific crossover and mutation operations to explore the solution space more effectively.

As seen from these prior works, a wide range of techniques have been developed to tackle the MSC problem. In this work, we explore four algorithms and evaluate their performance across a range of problem instances.

4 Algorithms

4.1 Branch and Bound

4.1.1 Algorithm Description. The implemented Branch-and-Bound (BnB) algorithm is designed to solve the Set Cover Problem. The algorithm uses a best-first search approach along with a lower bound and upper bound to effectively prune subproblems that are not "promising".

Algorithm 1 Branch-and-Bound for Set Cover

```
Require: Set cover instance $(n, subsets)$, time limit
Ensure: Best solution found and trace of improvements
1: $start \leftarrow$ current time
2: $trace \leftarrow [(0.0, cost(\text{GreedySolution}))]$
3: $best \leftarrow$ Greedy initial solution
4: $best\_size \leftarrow$ size of $best$
5: $root\_uncovered \leftarrow \{1, 2, ..., n\}$
6: $subset\_ids \leftarrow \{0, 1, ..., |subsets| - 1\}$
7: $max\_set\_size \leftarrow \max_{S \in subsets} |S|$
8: $root\_lb \leftarrow \lceil \frac{|root\_uncovered|}{max\_set\_size} \rceil$
9: $frontier \leftarrow [(root\_lb, 0, \emptyset, root\_uncovered, subset\_ids)]$
10: $visited \leftarrow \emptyset$
11: while $frontier$ is not empty and time limit not exceeded do
12:     Pop $(lb, k, chosen, uncovered, rem\_sets)$ from $frontier$
13:     if $lb \geq best\_size$ then
14:         continue
15:     end if
16:     if $uncovered = \emptyset$ then
17:         $best \leftarrow chosen$
18:         $best\_size \leftarrow k$
19:         Append (elapsed time, $best\_size$) to $trace$
20:         continue
21:     end if
22:     $state\_key \leftarrow (uncovered, chosen)$
23:     if $state\_key \in visited$ then
24:         continue
25:     end if
26:     Add $state\_key$ to $visited$
27:     Pick arbitrary $e \in uncovered$
28:     for all $i \in rem\_sets$ such that $e \in subsets[i]$ do
29:         $new\_chosen \leftarrow chosen \cup \{i\}$
30:         $new\_uncovered \leftarrow uncovered \setminus subsets[i]$
31:         $new\_rem \leftarrow rem\_sets \setminus \{i\}$
32:         $new\_lb \leftarrow (k+1) + \lceil \frac{|new\_uncovered|}{max\_set\_size} \rceil$
33:         if $new\_lb < best\_size$ and $|frontier| < MAX\_FRONTIER$ then
34:             Push $(new\_lb, k + 1, new\_chosen, new\_uncovered, new\_rem)$ into $frontier$
35:         end if
36:     end for
37: end whilereturn $best, trace$
```

4.1.2 Algorithm Background: Branch and Bound.

Initial Solution. The branch-and-bound (BnB) algorithm begins by constructing an initial upper bound using the greedy approximation algorithm, which is much better than the naive upper bound. Using this upper bound, the algorithm attempts to find a better solution through a best-first-search approach.

Search Space Exploration. The BnB algorithm explores the space of all subset selections using a best-first search with a priority queue as it's backing data structure. Each node in the search tree represents a partial solution, characterized by a set of chosen subsets, a set of uncovered elements, and the remaining available subsets. At each step, the algorithm selects the node with the lowest lower bound, computed as the current depth plus a fractional estimate of the number of subsets needed to cover the remaining uncovered elements. This ensures that promising candidates are expanded early, while nodes that cannot outperform the best known solution are pruned.

Lower Bound Estimation. The lower bound for a node is calculated by estimating the minimum number of additional subsets required to cover the remaining uncovered elements. Specifically, we divide the number of uncovered elements by the size of the largest subset in the problem and take the ceiling. This heuristic ensures we never overestimate and allow us to discard branches that look like they will not yield any improvements.

4.1.3 Complexity Analysis.

Time Complexity. The worst-case time complexity of BnB is $O(2^m \cdot m \log k)$, where m is the number of subsets and k is the number of nodes in the priority queue (frontier). This arises from the possibility of exploring all 2^m subset combinations, with each expansion involving heap operations ($O(\log k)$) and state updates over the remaining m subsets ($O(m)$). However, in practice, the greedy upper bound and the lower bound heuristic enable us to aggressively prune branches that are unlikely to yield improvements. The most expensive operations during execution include heap manipulations for maintaining the priority queue and state expansion when generating child nodes. Since nodes are only expanded if they are promising with respect to the current best solution, the actual runtime is highly dependent on the input instance and the user-defined time limit.

Space Complexity. The algorithm stores each node in the frontier, with each node maintaining sets of chosen subsets and uncovered elements. In the worst case, space usage grows as $O(k \cdot m)$, where k is the number of frontier nodes and m is the number of subsets.

4.1.4 Strengths and Weaknesses. Branch-and-bound offers the strongest theoretical guarantees among the considered algorithms, as it always returns an optimal solution if allowed to fully complete. The initial greedy upper bound and a fast lower bound heuristic leads to effective pruning, making the method viable for small and moderate sized instances. However, the main weakness of the algorithm is its scalability. The number of possible subset combinations grows exponentially, and for large instances, the algorithm may not complete within practical time limits.

4.1.5 Parameter Selection. We configure the BnB algorithm with a greedy-derived initial upper bound and a simple yet effective lower bound based on the maximum subset size. To limit memory consumption and maintain practical performance, the maximum size of the search frontier is capped at 100,000 nodes. The algorithm is executed with a fixed 900-second time limit, which was empirically chosen after testing longer cutoffs of 1200 and 1800 seconds. These higher cutoffs led to system errors, as the exponential nature of the best-first search method used in the algorithm on large instances caused unbounded memory usage. On our experimental platform—a MacBook Pro with 16 GB RAM—this often resulted in the Python kernel being forcefully restarted by the operating system due to excessive memory pressure. Consequently, we chose 900 seconds as a safe upper bound that allowed the algorithm to explore promising solutions without risking out-of-memory failures.

4.1.6 Implementation Details. The branch-and-bound solver is implemented in Python using a min-heap priority queue for node selection and hash-based state memoization for cycle avoidance. To manage memory usage, the frontier size is capped at a configurable constant (e.g., 100,000). Additionally, a hash set of visited states is maintained to avoid redundant work, contributing $O(v)$ to space, where v is the number of unique state keys. The implementation supports multi-processed execution for processing multiple instances in parallel. Solutions and their corresponding cost traces are output to files for downstream statistical analysis.

4.2 Approximation Algorithm with approximation guarantees

4.2.1 Algorithm Description. The implemented greedy algorithm works through selecting the subset that will cover the largest number of uncovered nodes at each iteration. It iterates until the universe is fully covered through this heuristic.

Algorithm 2 Greedy Approximation for Set Cover

```
Require: Set cover instance $(n, subsets)$
Ensure: Approximate solution covering all elements
1: Initialize $uncovered \leftarrow 1, \ldots, n$, $solution \leftarrow \emptyset$
2: while $uncovered \neq \emptyset$ do
3:     Find subset $S_i$ covering maximum uncovered elements
4:     $solution \leftarrow solution \cup i$
5:     $uncovered \leftarrow uncovered \setminus S_i$
6: end whilereturn $solution$
```

4.2.2 Approximation Guarantee. The approximation guarantee of the greedy solution is what allows usage of it without being subject to solutions extremely different than optimal. If there exists an optimal solution that uses k subsets, the greedy approach will use at most $k\left(1 + \ln \frac{n}{k}\right)$ subsets, where n is universe size [5]. Thus, as shown in the full proof in Dinitz, this implies an $O(\log n)$ approximation factor.

4.2.3 Complexity Analysis.

Time Complexity. Through each iteration, the subset with the greatest intersection over uncovered elements is found - which requires m iterations to go through each subset and n iterations to complete the intersection comparison for each subset, where m is the number of subsets and n is the universe size. Thus, the time complexity is $O(m^2 * n)$ because in the worst case, we iterate m times selecting all m subsets.

Space Complexity. To execute the algorithm, it stores the uncovered portion of the universe, thus $O(n)$ complexity. It also uses the subsets, thus $O(m)$, with a total space complexity of $O(m + n)$.

4.2.4 Strengths and Weaknesses. The strengths of the greedy algorithm are found in its speed (polynomial time) and intuitive heuristics/simplicity. The approximation guarantee allows for an understanding of how close the solution will be compared to the optimal one. The major weakness is that the greedy approach is suboptimal - although the heuristic of choosing the set which covers max uncovered nodes is fine, it doesn't generate an optimal solution. This results in more chosen subsets than optimal.

4.2.5 Implementation Details. The implementation is written in Python, using inbuilt set data structures for efficient intersection/difference computations. The same implementation is used in other discussed algorithms to provide initial solutions.

4.3 Simulated Annealing Local Search

4.3.1 Algorithm Description. The implemented simulated annealing algorithm (LS1) is designed to solve the Set Cover Problem. The algorithm uses a temperature-based approach to escape local optima and explore the solution space effectively.

Algorithm 3 Simulated Annealing for Set Cover

```
Require: Set cover instance $(n, subsets)$, time limit, initial temperature $T_0$, cooling rate $\alpha$
Ensure: Best solution found and trace of improvements
1: $current\_sol \leftarrow$ Greedy initial solution
2: $best \leftarrow current\_sol$
3: $T \leftarrow T_0$
4: $trace \leftarrow [(0.0, cost(best))]$
5: $no\_improvement \leftarrow 0$
6: while time limit not reached do
7:     $T \leftarrow T \times \alpha$
8:     $neighbor \leftarrow$ Random neighbor of $current\_sol$
9:     $\Delta \leftarrow cost(current\_sol) - cost(neighbor)$
10:    if $\Delta > 0$ or $random() < e^{\Delta/T}$ then
11:        $current\_sol \leftarrow neighbor$
12:        if $cost(current\_sol) < cost(best)$ then
13:            $best \leftarrow current\_sol$
14:            Add $(time, cost(best))$ to $trace$
15:            $no\_improvement \leftarrow 0$
16:        else
17:            $no\_improvement \leftarrow no\_improvement + 1$
18:        end if
19:        if $no\_improvement \geq max\_no\_improvement$ then
20:            Break
21:        end if
22:    end if
23: end whilereturn $best, trace$
```

Initial Solution. The simulated annealing procedure begins by constructing a greedy initial solution. Starting with an empty cover, the algorithm repeatedly selects the subset that covers the largest number of currently uncovered elements, adding it to the solution until every element in the universe is covered. This approach provides a feasible starting point that balances speed and solution quality, ensuring that all elements are addressed before refinement begins.

Neighbor Generation. To explore the solution space, each neighbor is generated by randomly selecting one subset and toggling its inclusion status in the current cover. If this modification results in an infeasible solution—that is, some elements become uncovered—the algorithm repairs the solution by invoking the same greedy selection process used for the initial construction. This repair maintains feasibility while still allowing the search to traverse regions of the space that may lead to improved covers.

Temperature Schedule. Temperature control follows an exponential cooling schedule. We initialize the temperature at $T_0 = 1.0$, which permits the acceptance of solutions that are up to one unit worse in cost during the early search phase, thus encouraging exploration. At each iteration, the temperature is multiplied by a cooling factor $\alpha = 0.98$. This gradual reduction balances exploration and exploitation, allowing the algorithm to transition smoothly from a broad search to fine-tuned improvements as the temperature diminishes.

4.3.2 *Complexity Analysis.*

Time Complexity. Generating the initial greedy solution requires $O(m \log m)$ time for sorting or selecting among m subsets. During each iteration of the annealing loop, neighbor generation runs in $O(1)$, solution validation in $O(n)$ (where n is the number of elements), and, when necessary, repair via the greedy approach in $O(m \log m)$. Since the total number of iterations is bounded by the user-defined time limit and a maximum number of consecutive non-improvements, the overall runtime scales linearly with these stopping criteria.

Space Complexity. The primary storage requirements consist of the current solution itself, which is $O(m)$, and the improvement trace, which grows in $O(t)$ where t is the number of recorded improvements. Consequently, total space usage is $O(m + t)$, which remains modest even for large instances provided the number of improvements is managed.

4.3.3 Strengths and Weaknesses. Simulated annealing offers the advantage of escaping local optima through its probabilistic acceptance of worse solutions, and its exponential cooling schedule effectively balances exploration and exploitation over time. By maintaining a trace of each improvement, the algorithm also supports post-hoc analysis of convergence behavior. However, performance is highly sensitive to the choice of temperature parameters: poor tuning can lead to excessive randomness or premature convergence. Moreover, there are no strict theoretical guarantees on solution quality, and the computational cost may become significant for very large instances due to repeated repairs and evaluations.

4.3.4 Parameter Selection. In our experiments, we set $T_0 = 1.0$ to allow acceptance of cost increases up to one unit early on, $\alpha = 0.98$ to ensure gradual cooling, and max-no-improvement = 10,000 to limit the number of consecutive non-improving steps. These values were chosen based on characteristics of set cover instances, a 600-second overall time budget, and empirical testing across various instance sizes to strike a balance between thorough search and timely termination. The initial inspiration for simulated annealing was derived from the professor's instruction in lecture.

4.3.5 Implementation Details. The implementation is written in Python and supports multi-processed execution for processing multiple instances in parallel. Solutions and their corresponding cost traces are output to files for downstream statistical analysis. Performance visualization is facilitated through Quantile-Runtime Distribution (QRTD) and Solution-Quality Distribution (SQD) plots, enabling comparison across different parameter settings and problem instances.

4.4 Hill Climbing Local Search

4.4.1 Algorithm Description. This algorithm starts with a randomly generated feasible solution that covers the entire universe. It then repeatedly selects the best neighboring solution—defined as the one with the greatest improvement in cost—until no better neighbor exists, at which point it restarts the process from a new random initialization.

Algorithm 4 Hill Climbing with Random Restarts for Set Cover

```
Require: Set cover instance $(n, subsets)$, time limit, max-no-improvement
Ensure: Best solution found and trace of improvements
 1: $start \leftarrow$ current time
 2: $trace \leftarrow [(0.0, \text{cost of best})]$
 3: $current \leftarrow$ random initialization
 4: while not covers_all($current$) do
 5:     $current \leftarrow$ random initialization
 6: end while
 7: $best \leftarrow current$
 8: $no\_improvement \leftarrow 0$
 9: while true do
10:     if elapsed time > $time\_limit$ then
11:         return $best, trace$
12:     end if
13:     $current \leftarrow$ random initialization
14:     while true do
15:         if elapsed time > $time\_limit$ then
16:             return $best, trace$
17:         end if
18:         $neighbor \leftarrow$ get_best_neighbor($current$)
19:         if $neighbor$ is None then
20:             break
21:         end if
22:         $current \leftarrow neighbor$
23:         if cost($current$) < cost($best$) then
24:             $best \leftarrow current$
25:             Append (time, cost($best$)) to $trace$
26:             $no\_improvement \leftarrow 0$
27:         else
28:             $no\_improvement \leftarrow no\_improvement + 1$
29:         end if
30:         if $no\_improvement \geq max\_no\_improvement$ then
31:             return $best, trace$
32:         end if
33:     end while
34: end while
```

Initial Solution. The hill climbing local search algorithm begins by generating an initial feasible solution through a random subset selection process. To ensure that this initialization is valid, random initializations are repeatedly sampled until one is found that fully covers the universe.

Search Space Exploration. The search proceeds by iteratively refining the current solution via steepest ascent in cost . At each step, the algorithm evaluates all neighboring solutions—each differing by the inclusion or exclusion of a single subset—and moves to the neighbor with the best improvement. This local optimization continues until a local minimum is reached, i.e., no neighbor yields a better solution. To ensure we encounter minimum poor local minima, the algorithm performs repeated random restarts, re initializing the search from a new random solution whenever no progress is made.

Termination Condition. The algorithm terminates under two conditions: if the allotted time limit is reached, or if the number of consecutive non-improving steps exceeds a user-defined threshold. In our experiments, we set this maximum number of non-improvements to 10,000, beyond which the algorithm ends to prevent excessive time spent in whihc no better solution is found.

4.4.2 *Complexity Analysis.*

Time Complexity. For each restart, the algorithm begins with a valid random initialization in $O(m)$ time and proceeds by exploring the neighborhood of the current solution. Evaluating each neighbor involves checking subset coverage and cost in $O(n)$ time, and with m potential neighbors, each local descent phase has a worst-case cost of $O(mn)$. If r restarts are performed and each descent involves t iterations before convergence or cutoff, the total time complexity is $O(r \cdot t \cdot m \cdot n)$ in the worst case.

Space Complexity. The algorithm stores the current and best solution sets, each of size $O(m)$, and a trace list recording improvements, which grows as $O(t)$ where t is the number of recorded improvements. No large auxiliary structures are required, making the method memory-efficient and ideal for large-scale instances.

4.4.3 *Strengths and Weaknesses.* Hill climbing is fast and simple to implement, with strong performance on small and moderately sized instances. Its greedy nature allows rapid convergence toward local optima, and the addition of random restarts improves performance by reducing the chance of getting stuck in poor-quality solutions. However, the algorithm lacks a mechanism for accepting worse solutions, which limits its ability to escape local minima compared to more exploratory methods like simulated annealing. Furthermore, performance may vary depending on the quality of initial solutions and the how the instance data is organized.

4.4.4 *Parameter Selection.* We configure the hill climbing algorithm with a maximum of 10,000 consecutive non-improving steps per run and a global time cutoff of 900 seconds. This limit was empirically determined to strike a balance between sufficient exploration and stability on our experimental platform.

4.4.5 *Implementation Details.* The hill climbing algorithm is implemented in Python and supports randomized multi-start search with configurable cutoff parameters. At each improvement step, timestamps and solution costs are recorded for downstream analysis and trace visualization. Neighbor evaluations are performed via a dedicated function that returns the best feasible modification to the current solution. The algorithm outputs both the final solution and its corresponding cost trace.

5 Experimental Setup

We tested our algorithms on benchmark instances from the course dataset, including small and large cases. Each run was limited to 600 seconds, and a different seed was used for each run to ensure robustness.

5.1 Environment

Experiments were conducted on a MacBook Pro with an M1 chip and 16 GB RAM. Python 3.11 was used for implementation.

5.2 Evaluation Metrics

We report average solution size, average runtime, and relative error against the known optimum for all small and large instances.

5.2.1 Upper Bound Comparison. Through our results and as shown earlier in the discussion on the approximate algorithm, we see an $O(\log n)$ approximation guarantee. This means the solution generated by the greedy algorithm will be within a logarithmic factor of the optimal solution. Our results for the algorithms are empirically sound and maintain this guarantee. If we compare the RelErr (relative error) between the greedy approach and the branch and bound approach, we notice consistently higher errors relative to optimal in the greedy results. While branch and bound is an exact algorithm that can reach optimal solutions (relative error approaches zero) given enough time, greedy settles at an approximation through its heuristic, resulting in solutions farther from optimal. This is consistent with our assertion of greedy as the upper bound to optimal, with branch and bound being a closer upper bound.

Table 1: Branch and Bound Performance Summary

Instance	AvgTime(s)	AvgSize	RelErr
large2	900.000	21.0	0.105
large3	900.000	17.0	0.133
large4	900.000	153.0	0.681
large5	900.000	8.0	0.333
large6	900.000	7.0	0.167
large7	900.000	172.0	0.811
large8	128.387	5.0	0.000
large9	900.000	16.0	0.143
large10	900.000	319.0	0.443
large11	900.000	56.0	0.400
large12	900.000	18.0	0.200
small1	0.056	5.0	0.000
small2	0.047	3.0	0.000
small3	0.047	5.0	0.000
small4	0.047	4.0	0.000
small5	0.046	5.0	0.000
small6	0.046	3.0	0.000
small7	0.047	3.0	0.000
small8	0.046	2.0	0.000
small9	0.046	3.0	0.000
small10	0.046	2.0	0.000
small11	0.051	4.0	0.000
small12	0.048	3.0	0.000
small13	0.046	2.0	0.000
small14	0.046	2.0	0.000
small15	0.047	2.0	0.000
small16	0.046	2.0	0.000
small17	0.046	2.0	0.000
small18	0.046	2.0	0.000

Table 2: Greedy Approximation Algorithm Performance Summary

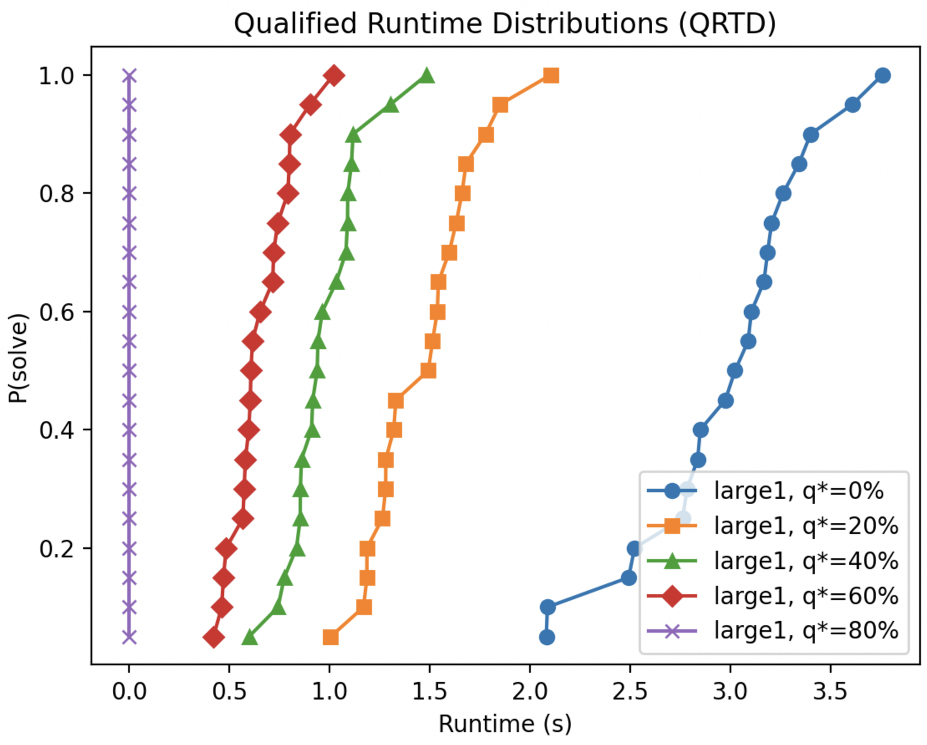
Instance	AvgTime(s)	AvgSize	RelErr
large1	0.543	83.000	0.660
large2	0.041	21.000	0.105
large3	0.046	17.000	0.133
large4	0.206	153.000	0.681
large5	0.043	8.000	0.333
large6	0.048	7.000	0.167
large7	0.532	172.000	0.811
large8	0.039	6.000	0.200
large9	0.047	16.000	0.143
large10	0.441	319.000	0.443
large11	0.133	56.000	0.400
large12	0.047	18.000	0.200
small1	0.037	5.000	0.000
small2	0.036	4.000	0.333
small3	0.037	6.000	0.200
small4	0.037	5.000	0.250
small5	0.037	6.000	0.200
small6	0.037	4.000	0.333
small7	0.038	4.000	0.333
small8	0.036	3.000	0.500
small9	0.036	4.000	0.333
small10	0.037	3.000	0.500
small11	0.037	5.000	0.250
small12	0.036	4.000	0.333
small13	0.037	3.000	0.500
small14	0.037	3.000	0.500
small15	0.036	3.000	0.500
small16	0.037	3.000	0.500
small17	0.037	3.000	0.500
small18	0.036	3.000	0.500

Table 3: Simulated Annealing Performance Summary

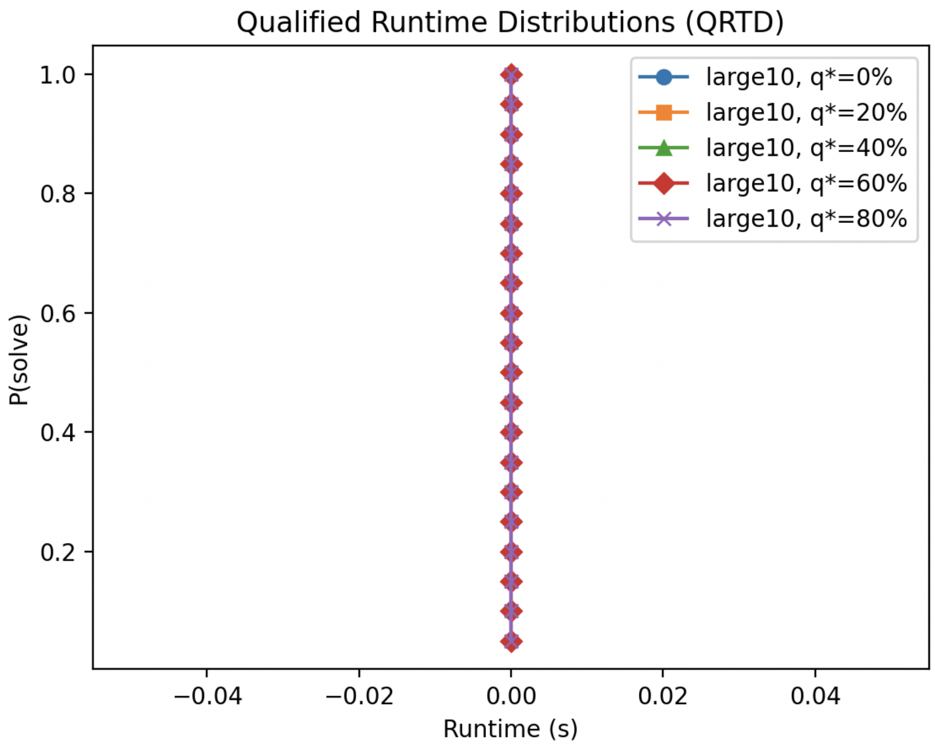
Instance	AvgTime (s)	AvgSize	RelErr
large1	56.161	50.000	0.000
large2	1.540	20.600	0.084
large3	8.054	17.000	0.133
large4	39.700	152.900	0.680
large5	6.735	8.000	0.333
large6	13.009	7.000	0.167
large7	145.901	172.000	0.811
large8	3.856	6.000	0.200
large9	7.698	16.000	0.143
large10	226.088	317.600	0.437
large11	36.018	56.000	0.400
large12	3.777	18.000	0.200
small1	0.201	5.000	0.000
small2	0.151	3.900	0.300
small3	0.153	5.800	0.160
small4	0.173	4.900	0.225
small5	0.214	5.000	0.000
small6	0.187	4.000	0.333
small7	0.118	4.000	0.333
small8	0.146	2.800	0.400
small9	0.177	3.800	0.267
small10	0.166	3.000	0.500
small11	0.227	4.800	0.200
small12	0.215	3.900	0.300
small13	0.149	3.000	0.500
small14	0.168	2.600	0.300
small15	0.187	3.000	0.500
small16	0.190	3.000	0.500
small17	0.283	3.000	0.500
small18	0.262	2.700	0.350

Table 4: Hill Climbing Performance Summary

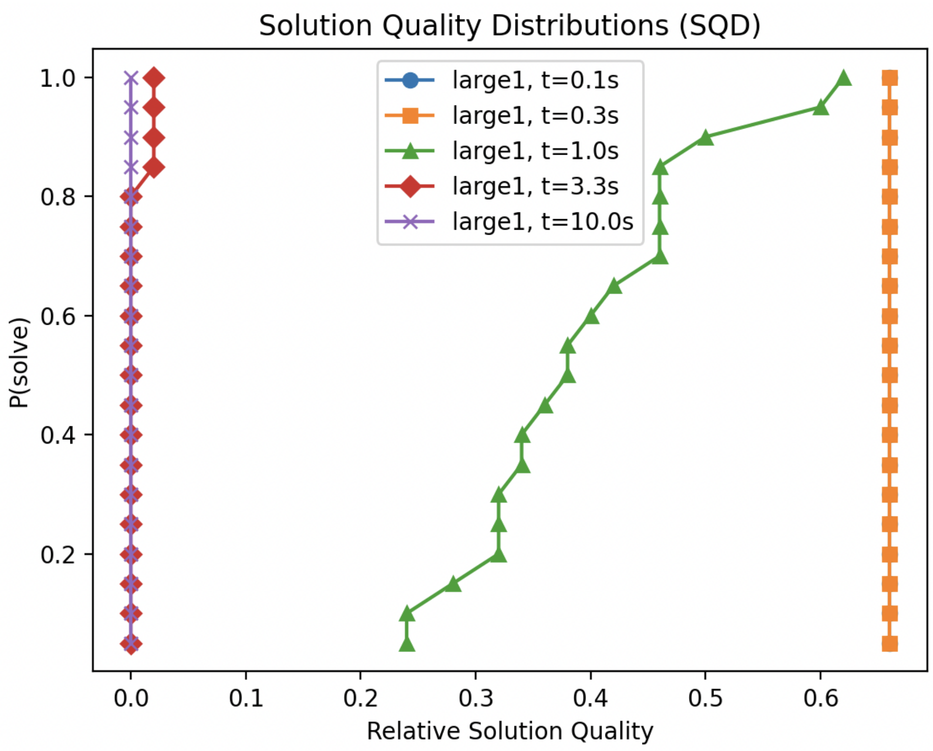
Instance	AvgTime (s)	AvgSize	RelErr
large1	673.135	149.800	1.996
large2	3.966	26.200	0.379
large3	21.593	25.800	0.720
large4	572.973	192.900	1.120
large5	6.401	9.600	0.600
large6	14.316	9.200	0.533
large7	666.747	230.800	1.429
large8	3.047	6.800	0.360
large9	18.839	25.700	0.836
large10	673.333	405.900	0.837
large11	320.516	77.700	0.943
large12	9.434	25.700	0.713
small1	0.154	5.000	0.000
small2	0.087	3.000	0.000
small3	0.104	5.000	0.000
small4	0.100	4.000	0.000
small5	0.124	5.000	0.000
small6	0.119	3.000	0.000
small7	0.092	3.000	0.000
small8	0.085	2.000	0.000
small9	0.104	3.000	0.000
small10	0.085	2.000	0.000
small11	0.139	4.000	0.000
small12	0.113	3.000	0.000
small13	0.100	2.000	0.000
small14	0.089	2.000	0.000
small15	0.115	2.000	0.000
small16	0.109	2.000	0.000
small17	0.132	2.000	0.000
small18	0.127	2.000	0.000



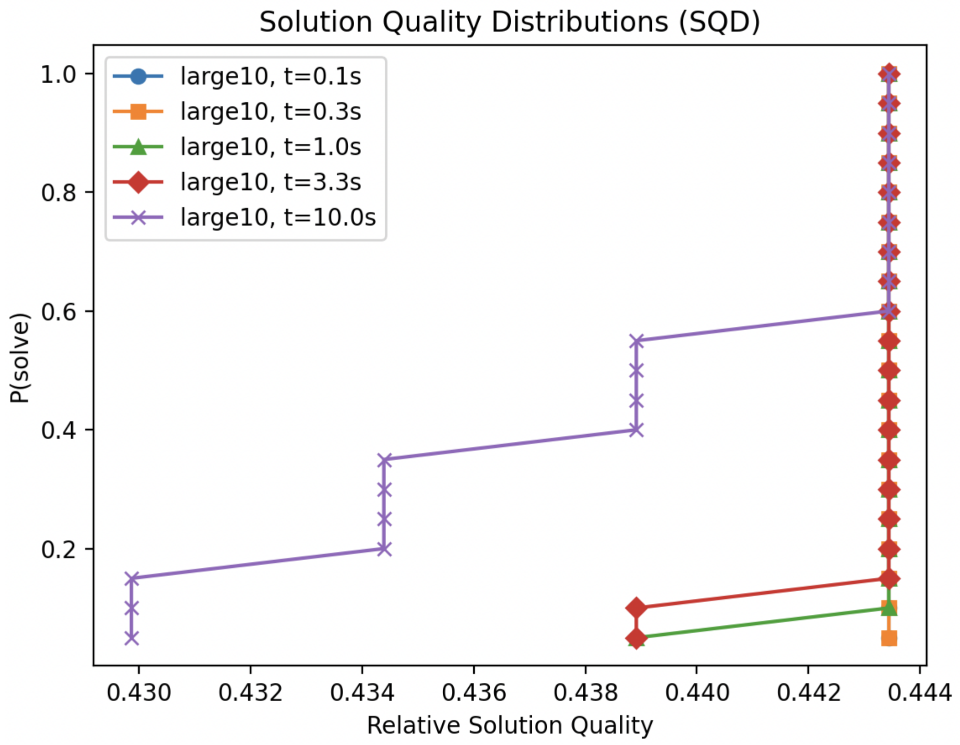
(a) ls1_large1_qrtd



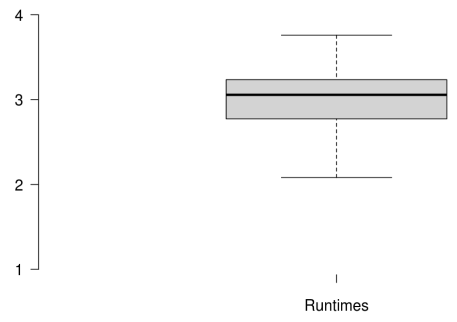
(b) ls1_large10_qrtd



(a) ls1_large1_sqd



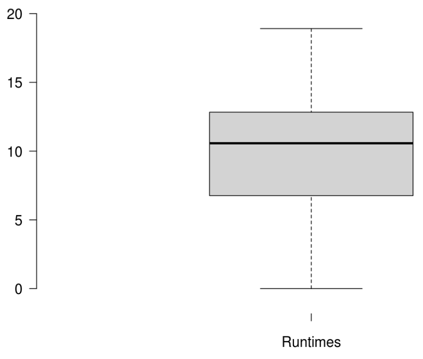
(b) ls1_large10_sqd



Box plot statistics

	Runtimes
Upper whisker	3.76
3rd quartile	3.23
Median	3.06
1st quartile	2.77
Lower whisker	2.08
Nr. of data points	20.00

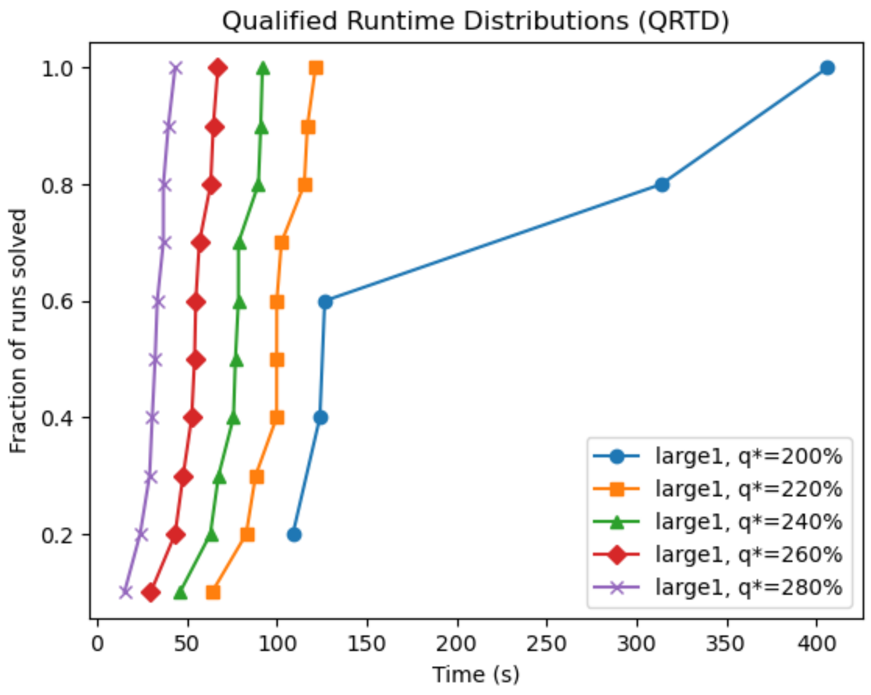
(a) ls1_large1_boxplot



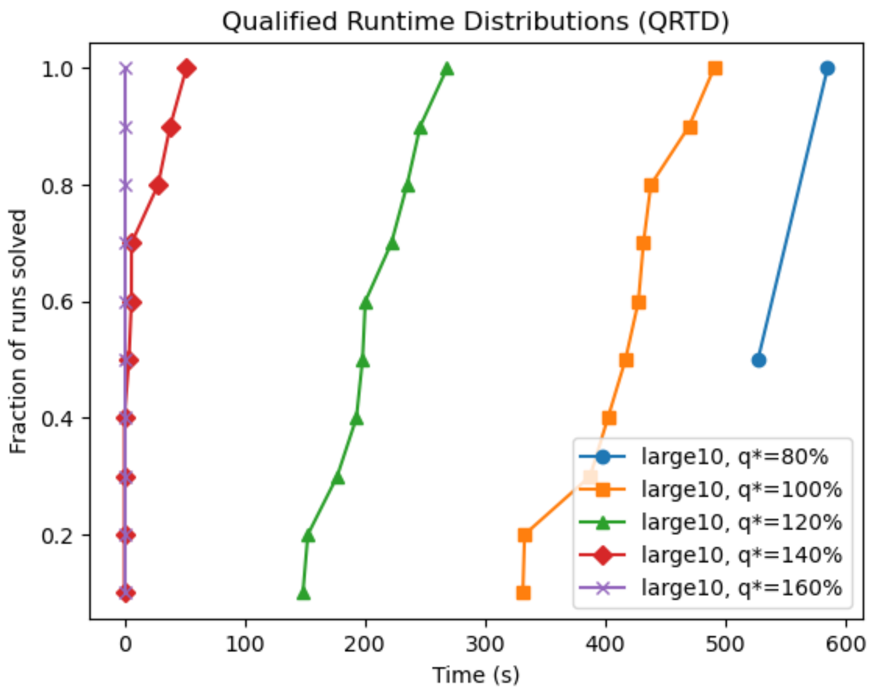
Box plot statistics

	Runtimes
Upper whisker	18.91
3rd quartile	12.83
Median	10.57
1st quartile	6.76
Lower whisker	0.00
Nr. of data points	20.00

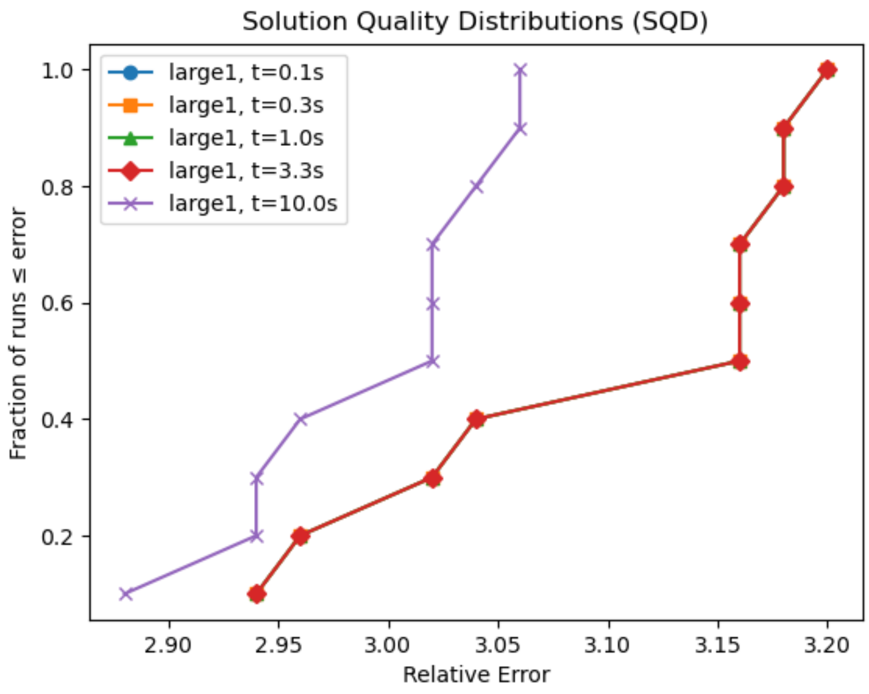
(b) ls1_large10_boxplot



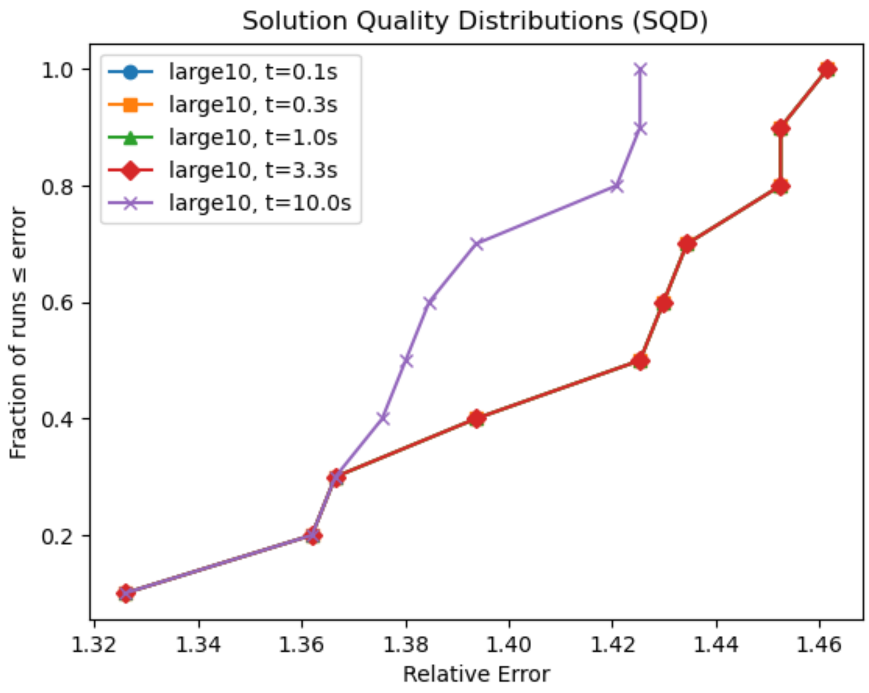
(a) ls2_large1_qrtd



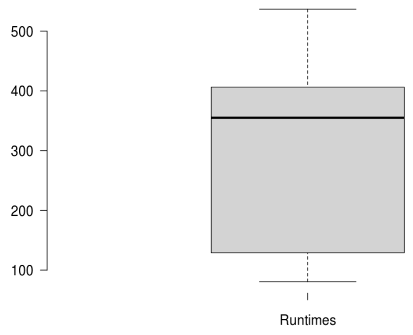
(b) ls2_large10_qrtd

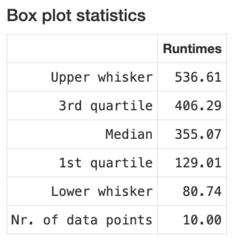


(a) ls2_large1_sqd



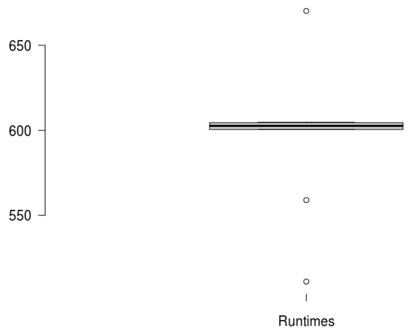
(b) ls2_large10_sqd

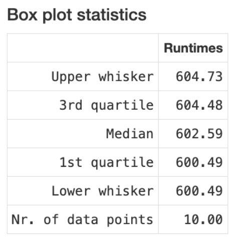



Box plot statistics

	Runtimes
Upper whisker	536.61
3rd quartile	406.29
Median	355.07
1st quartile	129.01
Lower whisker	80.74
Nr. of data points	10.00

(a) ls2_large1_boxplot




Box plot statistics

	Runtimes
Upper whisker	604.73
3rd quartile	604.48
Median	602.59
1st quartile	600.49
Lower whisker	600.49
Nr. of data points	10.00

(b) ls2_large10_boxplot

6 Discussion

Our experiments highlight key trade-offs across the four implemented algorithms. The greedy approximation algorithm performed unexpectedly well across both small and large datasets. On many large instances, its solution was already close to optimal, reducing the observable benefits of more sophisticated algorithms like branch-and-bound. For example, on instances like `large1` and `large10`, the greedy solution was within a few sets of the BnB-optimal value, and in some cases matched it exactly. This strong baseline performance limits the potential for BnB to show dramatic improvements, especially given its significantly higher computational cost and the practical time limit we had to set.
The branch-and-bound algorithm, while theoretically optimal, encountered memory and time scalability issues on large datasets due to the exponential growth of the search space. We capped its runtime at 900 seconds after testing longer cutoffs (e.g., 1200s and 1800s), which frequently led to out-of-memory errors and kernel crashes on our experimental setup. This cap ensured consistent behavior across runs while still allowing BnB to demonstrate its potential on small to medium instances.
Local search methods showed promising results, particularly in scenarios where greedy did not already yield near-optimal results. Simulated annealing offered high-quality solutions with reasonable runtimes. Hill climbing with random restarts was faster and memory-efficient, but more sensitive to local minima.

7 Conclusion

This study compares exact, approximate, and heuristic strategies for the Minimum Set Cover problem. The greedy algorithm proved surprisingly strong, consistently producing solutions close to optimal with minimal computation. While branch-and-bound guarantees optimality, its practical use is constrained by time and memory limits on larger instances. Local search methods, particularly simulated annealing, provide a useful middle ground—offering better solutions than greedy in some cases, without the resource demands of BnB. These results suggest that for many real-world applications where approximate solutions suffice, greedy and local search approaches are highly effective and scalable choices. Future work could explore hybrid methods or improved bounding strategies to scale BnB further while preserving exactness.

Acknowledgments

We thank the CSE 6140 course staff for their guidance and the benchmark data used for this project.

References

[1] Egon Balas and Maria C. Carrera. 1996. A Dynamic Subgradient-Based Branch-and-Bound Procedure for Set Covering. *Operations Research* 44, 6 (1996), 875–890.
[2] J. E. Beasley. 1990. A Lagrangian heuristic for set-covering problems. *Naval Research Logistics* 37, 1 (1990), 151–164.
[3] J. E. Beasley and P. C. Chu. 1996. A genetic algorithm for the set covering problem. *European Journal of Operational Research* 94, 2 (1996), 392–404.
[4] V. Chv'atal. 1979. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research* 4, 3 (1979), 233–235.
[5] Michael Dinitz. 2019. Approximation Algorithms: Set Cover and Max k-Cover. *Lecture Notes for 601.435/635, Johns Hopkins University* (2019). https://www.cs.jhu.edu/~mdinitz/classes/Approx19/lecture4.pdf.
[6] Uriel Feige. 1998. A Threshold of ln n for Approximating Set Cover. *J. ACM* 45, 4 (1998), 634–652.
[7] L. W. Jacobs and M. J. Brusco. 1995. A local search heuristic for large set-covering problems. *Naval Research Logistics* 42, 7 (1995), 1129–1140.
[8] David S. Johnson. 1974. Approximation Algorithms for Combinatorial Problems. *J. Comput. System Sci.* 9, 3 (1974), 256–278.
[9] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, Raymond E. Miller and James W. Thatcher (Eds.). Springer, New York, NY, USA, 85–103.

[9]