Project Report - Team No.31

WENWEN LI, XINYI MAO, and HUI QIAO

The Minimum Set Cover problem is a famous NP-complete problem which is widely applied in areas such as resource allocation, data mining, and network security. In this paper, we solve the Minimum Set Covering problem with four different algorithmic approaches and compare their performance. The algorithms we implemented include a Branch and Bound Algorithm, a Greedy Algorithm with logarithmic approximation guarantees, and two different variants of Local Search algorithms - Simulated Annealing and Random Restart Hill Climbing. We evaluate these algorithms both theoretically and empirically, with respect to solution quality and computation time, across small, and large datasets. The results show the practical trade-offs between accuracy and efficiency, providing insights into the suitability of different algorithmic strategies for solving large-scale instances of the Minimum Set Cover problem. The Greedy Algorithm excels in running time across both small and large datasets, but it has a high relative error. BnB provides an optimality guarantee, but its running time becomes extremely high on large datasets. Hill Climbing offers the best balance between running time and solution quality for large datasets. BnB is the preferred choice when the dataset is small, while Hill Climbing is the preferred choice when the dataset is large.

1 Introduction

The Minimum Set Cover problem is a foundational challenge in combinatorial optimization that arises in various domains, including bioinformatics, machine learning, urban planning, and operations research. Despite its simple formulation, the problem is known to be NP-complete, making it computationally intractable for large instances when using naive brute-force methods. For example, we may want to build as few libraries as possible to ensure that every resident in a city has access to one. Similarly, in cybersecurity, we aim to select the minimum number of detection tools needed to cover all known vulnerability types, thereby ensuring safety while minimizing resource costs.

The goal of this project is to explore the practical and theoretical aspects of solving the Minimum Set Cover problem through a diverse set of algorithmic paradigms. Given m subsets and U with n elements, we want to find the minimum number of subsets such that the union of all subsets covers the entire universe U. We implement four algorithms: a Branch-and-Bound exact solver, a greedy approximation algorithm with $ln(n)$ approximation guarantees, and two heuristic local search strategies - Simulated Annealing and Random Restart Hill Climbing.

Our work involves both algorithm design and empirical evaluation. We run our implementations on real-world and synthetic datasets of small and large sizes to understand how these algorithms perform under time constraints and to analyze the trade-offs they offer in terms of solution quality and runtime. We define small instances as those where n and m are less than 20, where n is the number of elements in U, and m is the number of subsets. Large instances are those where m and n are greater than 20 but less than 2000.

For small instances ($m, n \leq 20$), the running time for all four algorithms is 0 seconds, which is extremely fast. In terms of solution quality, the Branch and Bound (BnB) and Local Search algorithms achieve the optimal solution, with a relative error of 0. However, the Greedy Algorithm has an average relative error of 0.36.

For large instances ($20 \leq m, n \leq 2000$), the Greedy Algorithm has the lowest running time, followed by Simulated Annealing, Hill Climbing, and finally Branch and Bound. The Branch and Bound algorithm reaches the cutoff time and exhibits extremely high running time. In terms of solution quality for large instances, the Greedy Algorithm has a relative error of 0.36, BnB has 0.33, Simulated Annealing has 0.26, and Hill Climbing achieves the best result with a relative error of 0.20.

Authors' Contact Information: Wenwen Li; Xinyi Mao; Hui Qiao.

Overall, for small instances, all four algorithms achieve both high accuracy and efficiency, but BnB is the best choice due to its optimality guarantees. For large instances, Hill Climbing offers the best balance between solution quality and runtime efficiency. This paper serves as a comprehensive case study on the algorithmic treatment of the minimum set covering problem, highlighting the strengths and limitations of exact, approximate, and heuristic methods.

2 Problem Definition

Let $U = \{x_1, x_2, \ldots, x_n\}$ be a universe of n elements and let $S = \{S_1, S_2, \ldots, S_m\}$ be a collection of m subsets of U, where each $S_i \subseteq U$. The Minimum Set Cover problem is to find a subcollection $T \subseteq S$ such that the union of all subsets in T covers the entire universe U, i.e.,

$$\bigcup_{S_i \in T} S_i = U,$$

and the cardinality of T, denoted $|T|$, is minimized.

Formally, the objective is:

$$\min |T| \quad \text{subject to} \quad \bigcup_{S_i \in T} S_i = U.$$

This problem is NP-complete, as proven by Richard Karp in his seminal list of 21 NP-complete problems. It remains a cornerstone problem in complexity theory and algorithm design due to its broad applicability and theoretical richness.

3 Related work

3.1 Branch-and-Bound

Several exact algorithms have been proposed to solve the MSC problem, among which branch-and-bound remains a widely studied approach. Three components of Branch-and-Bound are often used to improve the performance of the algorithm, the search strategy, the branching strategy, and the pruning strategy [17]. Early works solved MSC exactly for small to moderate instances, using an upper bound from the greedy algorithm result, and a lower bound from a fast heuristic or from the LP relaxation [4, 15] to tune the pruning strategy. More recently, modern developments on the branch-and-bound algorithm have been focusing on larger instances and more different techniques under these three strategies. For search strategy, modifications have been made to depth-first search (DFS), breadth-first search (BrFS), and best-first search (BFS). Meanwhile, a Distributed best-first search (DBFS), considered a hybrid of DFS and BFS, has been applied to successfully solve a single machine scheduling problem where the objective is to find a schedule with the minimum total tardiness, outperforming the other approaches [12]. For branching strategy, an alternative branching strategy that generates many subproblems at each node in the branch-and-price tree, significantly reduces the length of any path through the search tree [18]. For the pruning strategy, some new approaches use some learning methods to effectively prune a large set of unexplored nodes as early as possible and achieve a solution with an optimality gap guarantee were proposed [9, 19]. Recent advancements in branch-and-bound algorithms for the MSC problem have also introduced learning-based techniques to enhance search, branching, and pruning strategies. Machine learning models, particularly those employing imitation learning, have been utilized to guide node selection, improving search efficiency [8].

3.2 Approximation Algorithm

There are several approximation algorithms used to solve the minimum set covering problem, such as greedy algorithm, LP relaxation and rounding, Primal-Dual methods. Chvatal(1979)[7] adopts greedy algorithm to solve weighted set-covering problem by iteratively picks the subset that covers the largest number of uncovered elements and proves that the Greedy achieves an approximation factor of $ln(n)$. The second approximation algorithm is LP relaxation and rounding that formulate the problem as an integer linear problem, relax it to an LP and round the solution. Hochbaum(1997) [10] show the LP rounding method can match the greedy algorithm's performance. Bar-Yehuda and Even (1981) [3] used Primal-Dual methods, an alternative to LP rounding, to build a feasible solution to the dual LP and construct a cover simultaneously. It is particularly useful in weighted version of the set-covering problem.

3.3 Local Search Algorithm

Commonly used local search algorithms includes: simulated annealing, tabu search, genetic algorithms, hill climbing with randomized bad moves, variable neighborhood search. Jabalameli et al. (2009)[11] and kirkpatrick et al.(1985)[14] used simulated annealing which allows for non-improving moves to escape local minima to solve the set covering problem. It starts from an initial solution, simulated annealing iteratively accepts neighboring solutions based on a temperature schedule. The temperature starts high and gradually reduced, lowering the probability of accepting worse solutions as the iterations progresses. Similarly to simulated annealing, hill climbing with randomized bad moves also accept non-improving moves at certain probabilities and the probability of accepting bad moves typically decreases over time. Bagherinejad et al.(2017) [2] used hill clibing and genetic algorithm to solve the maximal covering location problem and find hill climbing has a better performance. Iterated local search is an algorithm that applies local search iteratively by modifying the current solution to escape from local minima. After each local search, a perturbation is applied to explore a new part of the solution space, and local search is applied again. Chiarandini and St¨utzle(2002) [6] and Lourenço et al.(2003) [16] proved that algorithm performance can be improved with the iteration.

4 Algorithms

4.1 Exact branch-and-bound

4.1.1 Algorithm Description The Branch-and-Bound (BnB) algorithm solves the Set Cover Problem by exploring a binary decision tree using a breadth-first search strategy. Each node in the tree represents a partial solution, determined by whether a specific subset is included or excluded. To improve efficiency, the algorithm first computes an initial upper bound using a greedy approximation: at each step, the greedy method selects the subset that covers the largest number of uncovered elements. This provides a feasible, though not necessarily optimal, reference solution, which sets the initial upper bound on the number of subsets used. As BnB algorithm explores the tree, it performs following checks at each node:

- If the current solution already covers the universe, and uses fewer subsets than the current best solution, it updates the best solution.
- Else, it estimates a lower bound on the number of additional subsets required. This estimate is based on the number of uncovered elements and the size of the largest remaining subset.

If the lower bound is exceeds the upper bound, the node is pruned, which means we skip the further exploration, because we won't get a better solution. If the lower bound is still promising, we will continue to explore the inclusion of the current subset.

4.1.2 Complexity Analysis The time complexity is $O((m+n)2^m)$, where m represents the number of subsets, and n represents the size of the universal set. This complexity is from the binary branching of the search tree. For m subsets, m is the depth of the tree, which creates a search tree with up to $O(2^m)$ nodes in the worst case[5]. For each node, the algorithm performs operations:

- We test whether the current coverage bitmask equals the target mask ($covered == all\_covered$), which takes $O(n)$ time complexity in the bit-length n.
- We compute how many elements are still missing. Converting to a string of bits and counting the ones takes $O(n)$ time.
- To pass the current best solution down the recursion, we need to copy the selected subsets list, like $new\_selected = selected + [this\_subset]$. It takes $O(k)$ to copy a list of size k, and the worst case is $O(m)$.
- We need to compute the bounds to decide whether to prune. We scan all the subsets and elements, which takes $O(m+n)$.

Thus, we sum up the above costs for each node and know that it takes $O(m+n)$ for each node in the worst case. However, in practice, the complexity can be significantly reduced due to effective pruning with high-quality upper and lower bounds. The space complexity, dominated by the breadth-first search queue, is $O(m2^m)$, but can benefit similarly from effective pruning.

4.1.3 Algorithm Strengths First, the algorithm is guaranteed to find the optimal solution if the search tree is fully explored or if all remaining nodes are pruned. Second, it offers flexibility in design, allowing for the incorporation of diverse search strategies, branching heuristics, and pruning or dominance rules. Furthermore, the approach is well-suited for parallel computing, as independent subtrees can be distributed across processing cores with minimal communication overhead, owing to the use of a shared global upper bound.

4.1.4 Algorithm Weaknesses A primary weakness of the Branch-and-Bound algorithm is its worst-case exponential complexity, making it computationally impractical for large problem instances or for those hard to find effective pruning. The algorithm's performance heavily depends on the quality of heuristic bounds and pruning rules.

Algorithm 1 Branch-and-Bound

Require: Universe U, subsets $S = \{S_1, \dots, S_m\}$, cutoff time T_{cutoff}

Ensure: Minimum index set that covers all elements in U

```
 1: $k^* \leftarrow$ GreedyCover($S$)                                ▷ Initial upper bound
 2: Sort $S$ in descending order of $|S_i|$
 3: Compute max_remaining[$1 \dots m$]
 4: Initialize queue with ($i = 1$, covered = $\emptyset$, $c = 0$, sel = $\emptyset$)
 5: while queue not empty and time $< T_{\text{cutoff}}$ do
 6:     ($i$, covered, $c$, sel) $\leftarrow$ Dequeue
 7:     if covered $= U$ then
 8:         if $c < k^*$ then
 9:             $k^* \leftarrow c$
10:             Record current solution sel
11:         end if
12:         continue
13:     end if
14:     if $i > m$ then
15:         continue
16:     end if
17:     $r \leftarrow |U \setminus \text{covered}|$
18:     $M \leftarrow$ max_remaining[$i + 1$] if $i < m$, else 0
19:     if $M > 0$ then
20:         LB $\leftarrow c + \lceil r/M \rceil$
21:     else if $r > 0$ then
22:         LB $\leftarrow \infty$
23:     else
24:         LB $\leftarrow c$
25:     end if
26:     if LB $\geq k^*$ then
27:         continue
28:     end if
29:                                                                  ▷ Exclude $S_i$
30:     Enqueue(($i + 1$, covered, $c$, sel))
31:                                                   ▷ Include $S_i$ if new coverage
32:     new_cover $\leftarrow$ covered $\cup S_i$
33:     if new_cover $\neq$ covered and $c + 1 < k^*$ then
34:         Enqueue(($i + 1$, new_cover, $c + 1$, sel $\cup \{i\}$))
35:     end if
36: end while
37: return best solution found
```

4.2 Approximation Algorithm

4.2.1 Algorithm Description In this paper, we choose greedy algorithm as the approximation algorithm. The greedy algorithm begins with an empty cover and repeatedly selects the subset that covers the largest number of uncovered elements until the universe is fully covered. We maintain chosen_subsets, best_subset, best_index, and max_newly_covered, and loop through each subset in the collection of subsets. First, we compute the number of newly covered elements. If the newly_covered is greater than max_newly_covered, we update max_newly_covered with newly_covered, set best_subset to the current subset, and update best_index with the current index. After finishing the loop, we update the covered set with the newly added subset and append it to the chosen_subsets list. We repeat this process until the covered set equals U.

Algorithm 2 Greedy Set Cover Algorithm

```
Require: Universe set $U$, list of subsets $S = \{S_1, S_2, \ldots, S_m\}$
Ensure: Indices of selected subsets that cover the universe
1:  $covered \leftarrow \emptyset$                              ▹ Initial empty set of covered elements
2:  $chosen\_subsets \leftarrow [\,]$                           ▹ List to store indices of chosen subsets
3:  while $covered \neq U$ do
4:      $best\_subset \leftarrow$ None                          ▹ Best subset found in this iteration
5:      $best\_index \leftarrow -1$                             ▹ Index of the best subset
6:      $max\_newly\_covered \leftarrow 0$                      ▹ Maximum number of newly covered elements
7:      for each subset $S_i$ in $S$ do
8:          $newly\_covered \leftarrow |S_i \setminus covered|$  ▹ Number of newly covered elements by $S_i$
9:          if $newly\_covered > max\_newly\_covered$ then
10:             $max\_newly\_covered \leftarrow newly\_covered$
11:             $best\_subset \leftarrow S_i$
12:             $best\_index \leftarrow i$
13:         end if
14:     end for
15:     if $best\_subset$ is None then
16:         raise Error: No feasible solution
17:     end if
18:     $covered \leftarrow covered \cup best\_subset$          ▹ Update covered elements
19:     Append $best\_index + 1$ to $chosen\_subsets$           ▹ 1-indexed subset index
20:     Remove $best\_subset$ from $S$    ▹ Remove the chosen subset from the list(optionally), improve efficiency
21: end while
22: return $chosen\_subsets$
```

4.2.2 Approximation ratio Chvatal(1979) [7] proved that the ratio between the greedy and optimal algorithm grows at most logarithmatically in the largest set in the weighted version of minimum set covering problem. Our problem is a simplified version where the cost is one for each subset. To be more specific, he finds that the cost of the cover returned by greedy heuristic is at most H(d) times the cost of an optimal cover, where d is the size of the largest set S_j.

$$H(d) = \sum_{j=1}^{d} \frac{1}{j} \tag{1}$$

$$= \ln d + \gamma + \frac{1}{2d} - \frac{1}{12d^2} + \cdots \tag{2}$$

$$<= \ln d + \gamma \tag{3}$$

Where $\gamma \approx 0.5772$ is the Euler–Mascheroni constant. The size of the largest set can't exceed n. Thus, the greedy algorithm has a approximation guarantee of $O(ln(n))$.

4.2.3 Complexity Analysis The time complexity is $O(m^2n)$ where m is the number of subsets and n is the number of elements in U. Firstly, in the outer while loop, this loop will run until all elements in U are covered. In the worst case, we will select at most m subsets. So this loop will iterate at most m times. Secondly, in the inner for loop, the inner for loop will iterate over each subset in subsets. For each subset, the algorithm computes the set difference(subset-covered). In the worst case, the set difference operation has a time complexity of O(n). This is because each subset can potentially have up to n elements, and the subset - covered operation requires iterating over all the elements in the subset. So the overall time complexity of algorithm is $O(mxnxn) = O(m^2n)$. The space complexity is $O(mn)$. The covered set stores the elements covered by the chose subsets. In the worst case, this set will store all n elements from the universe, so its space complexity is O(n). the chosen_subsets stores the indices of the selected subsets. In the worst case, all subsets are chosen. The space complexity for this list is $O(m)$. The space used by the subsets depends on the number of subsets and the size of each subset. Each subset can contain up tp n elements, so the space complexity for subsets is $O(mn)$. therefore, the total space complexity is $O(mn)$.

4.2.4 Algorithm Strength Greedy algorithm is easy to understand and implement. It selects subsets that cover the maximum uncovered subsets. The greedy algorithm runs pretty quickly especially on large problems. Even though its solution maybe not optimal, it can provide a good approximation and a starting point for more advanced techniques, like local search. Overall, it is simple, fast, and performs surprisingly well in practice, especially on large datasets where exact methods are infeasible.

4.2.5 Algorithm Weakness The greedy algorithm might select subsets that provide immediate coverage but fail to select subsets that are more beneficial in the long run. Thus, the greedy algorithm can't guarantee the optimal solution and it has a worst-case approximation ration of $ln(n)$, where n is the size of the universe. This means the solution can be up to $ln(n)$ times worse than the optimal solution in the worst case. When the dataset size is small, the greedy algorithm is similar to the optimal solution, but it will become less accurate as the dataset size increases, often resulting in a suboptimal solution with a higher relative error due to the lack of global optimization.

4.3 Local Search 1 – Simulated Annealing

4.3.1 Algorithm Description

Simulated annealing is an optimization algorithm inspired by the process of annealing [13] in metallurgy. The process mainly uses a temperature parameter to control the exploration and exploitation. If the temperature is high, the probability of accepting worse neighboring solution is high, which means the algorithm try to explore a better optimum by trading off current best solution. When the solution is closer to the optimum, the probability will be reduced which means it is less likely to explore other directions. Here I let temperature times cooling parameter in each loop.

The algorithm starts with an initial solution and then find a neighbor based on current solution. If the neighbor has better solution than current, then the current solution will be the neighbor. If the neighbor does not have a better solution than current, then there's a probability that the current solution will accept this worse neighbor. With the difference between the current solution cost and neighbor's cost smaller, this probability becomes smaller. The algorithm is show in algorithm 3.

In addition, I set a parameter to track if the algorithm converges called $track\_convergence$. If the best solution is not updated in 1000 iterations, the algorithm will stop. Although this may reduce the quality of the algorithm, it is helpful to reduce the runtime. I also tuned this parameter to 1000 based on the trade off between quality and runtime by comparing the parameter 100, 500, and 1000.

4.3.2 *The initial solution*

The initial solution is returned from $get\_initial\_solution(universe, subset)$, which is shown in algorithm 4. It is found by keeping choosing the subset which covers the most elements in the universe until all chosen subsets covers the universe. I tried other methods of choosing initial solution such as including all subsets and choosing the first several subsets until the subsets cover all elements in universe, but the results turned out worse than current choice.

4.3.3 *Find the neighbor*

The neighbor is returned from $get\_neighbor(current\_solution, subset, universe)$, which is shown in algorithm 5. It is found by randomly choosing one of three actions which are removing a subset from the current solution, adding a subset to the current solution, and swapping a subset in the current solution to one that is not in the current solution.

In the "removing" action, I randomly choose an index in the current solution to remove until the subset after removing is a valid cover. If there's no choice, then the current solution is the smallest cover. It may stuck or reach the local optimum. If it stucks, then it can leave the stuck by adding a subset or swapping the subset.

In the "adding" action, I randomly choose a subset from the whole list of subsets, if it is not in the current solution, then it will be added to the current solution.

The "swap" action randomly chooses an index to remove from the neighbor and then randomly chooses an index to add into the current solution. For reducing the runtime, I randomly choose the remove index once and then randomly choose an added index in a loop until this swapping can still make a valid solution.

4.3.4 *Weakness and Strength*

Simulated Annealing takes longer time to get the local optimum. Because in the method of getting neighbor, "remove" and "swap" action needs to loop the choice until find a valid solution, or loop all choices and keep the original solution. This is not really efficient. However, it is good at exploring the search space. It can explore more search spaces than without looping and avoid stucking in poor local minimum.

In addition, the performance of simulated annealing depends on the parameter like initial temperature, number of iterations, and $track\_convergence$. although I set a convergence parameter, that does not mean the algorithm actually converges, there's no guarantee of optimality. The more the iterations or larger the $track\_convergence$ is, the higher probability it can find the optimum, but the longer time it needs. Overall, simulated annealing can return a relative good result in reasonable amount of time but there may be better algorithm for solving the same problem.

Algorithm 3 Simulated Annealing

```
Require: Universe $U$, subsets $S = \{S_1, \ldots, S_m\}$,initial_temp $T$, cooling_rate $C$, max_iterations $itr$, seed $seed$, cutoff
    time $T_{\text{cutoff}}$
Ensure: Minimum index set that covers all elements in $U$
 1: $current\_solution \leftarrow$ get_initial_solution$[U, S]$
 2: $best\_solution \leftarrow current\_solution$
 3: $track\_convergence \leftarrow 0$
 4: for $t \leftarrow 1$ to $itr$ or $T_{\text{cutoff}}$ do
 5:     if $T == 0$ then
 6:         break
 7:     end if
 8:     if $track\_convergence > 1000$ then
 9:         break
10:     end if
11:     $track\_convergence = track\_convergence + 1$
12:     $neighbor \leftarrow$ get_neighbor$[current\_solution, S, U]$
13:     $\delta \leftarrow$ cost$[current\_solution]$ − cost$[neighbor]$
14:     if $\delta \geq 0$ then
15:         $current\_solution \leftarrow neighbor$
16:     else if with the probability less than $e^{\frac{\delta}{T}}$ then
17:         $current\_solution \leftarrow neighbor$
18:     end if
19:     if cost$[current\_solution]$ < cost$[best\_solution]$ then
20:         $track\_convergence \leftarrow 0$
21:         $best\_solution \leftarrow current\_solution$
22:     end if
23:     $T = T * C$
24: end for
25: return best_solution
```

Algorithm 4 get_initial_solution

```
Require: Universe $U$, subsets $S = \{S_1, \ldots, S_m\}$
Ensure: The initial solution of the problem
1: $uncovered \leftarrow U$
2: $solution \leftarrow []$
3: while uncovered do
4:     $best\_index \leftarrow$ the index of the subset that covers the most uncovered set
5:     add $best\_index$ to $solution$
6:     $uncovered = uncovered - S[best\_index]$
7: end while
8: return solution
```

Algorithm 5 get_neighbor

```
Require: current_solution $current\_solution$, Universe $U$, subsets $S = \{S_1, \ldots, S_m\}$
Ensure: The next neighbor
 1: $neighbor \leftarrow current\_solution$
 2: $action \leftarrow$ randomly choose from [add, remove, swap]
 3: if $action == remove$ and $len(neighbor) > 1$ then
 4:     $choice \leftarrow neighbor$
 5:     $to\_remove$ = randomly choose one from choice
 6:     $test \leftarrow$ the list after removing $to\_move$ from $neighbor$
 7:     $choice \leftarrow test$
 8:     while $test$ is not a valid solution and len($choice$) $> 0$ do
 9:         $to\_remove$ = randomly choose one from choice
10:         $test \leftarrow$ the list after removing $to\_move$ from $neighbor$
11:         $choice \leftarrow$ the list after removing $to\_move$ from $choice$
12:     end while
13:     if the test is a valid solution then
14:         $neighbor \leftarrow test$
15:     end if
16: else if $action == add$ then
17:     $candidate \leftarrow$ randomly choose one index from subsets
18:     if $candidate$ is not in the $neighbor$ then
19:         add $candidate$ into $neighbor$
20:     end if
21: else if $action == swap$ and len($neighbor$) $> 0$ and len($neighbor$) $<$ len($S$) then
22:     $to\_remove$ = randomly choose one from $neighbor$
23:     $not\_neighbor \leftarrow$ the list that in $S$ but not in the $neighbor$
24:     $to\_add \leftarrow$ randomly choose an index of subset from $not\_neighbor$
25:     $test \leftarrow$ the list that remove $to\_remove$ from $neighbor$ and add $to\_add$ into $neighbor$
26:     while $test$ is not a valid solution and len($not\_neighbor$) $> 0$ do
27:         $to\_add \leftarrow$ randomly choose an index of subset from $not\_neighbor$
28:         $test \leftarrow$ the list that remove $to\_remove$ from $neighbor$ and add $to\_add$ into $neighbor$
29:         $not\_neighbor \leftarrow$ the list that removes the $to\_add$ from $not\_neighbor$
30:     end while
31:     if the test is a valid solution then
32:         $neighbor \leftarrow test$
33:     end if
34: end if
35: return $neighbor$
```

4.4 Local Search 2 – Random Restart Hill Climbing

4.4.1 Algorithm Description

Hill climbing is an algorithm that keeps searching for the neighbor with the most improvement. In this algorithm, I use the random restart to escape from the poor local minimum. I choose random restart hill climbing because original hill climbing always stucks in a local optimum. Random restart is helpful to find another better local optimum and there's chance to find the optimum based on a random initial.

The algorithm starts from an choosing a random initial solution and then I set it as a best solution. Then the algorithm generates a random initial solution as a current solution and then find the best neighbor based on it. If there's a better neighbor than current solution then keeps searching for the next best neighbor until there's no better neighbor. If the cost of final neighbor i.e.the local optimum is better than the cost of best solution, the best solution becomes the neighbor. Then another random initial solution is generated and the process is repeated until reach the max iterations or the cut off time. The algorithm is provided in algorithm 6. In addition, I also have $tracking\_convergence$ to cut off the algorithm if the best solution is not updated in 100 iterations.

4.4.2 Random Initial Solution

The random initial solution is return from method $get\_random\_initial[subsets, universe]$, which is shown in algorithm 7. It has an uncovered set which records all element that the set does not cover the universe. Then it first randomly chooses a subset from the list of all subsets into the solution and the uncovered set will remove the element in this subset. Then it starts the process as getting initial solution in Simulated Annealing, which is besides the chosen subset, it keeps adding the subset that has the most element in the uncovered set into the solution and removing the element covered from uncovered set until all elements in the universe are covered. Finally it returns the solution.

4.4.3 Find the best neighbor

The best neighbor is returned from $get\_best\_neighbor[current\_solution, subsets, universe]$, which is shown in algorithm 8. It is found by keeping randomly choosing a subset from the current solution to remove until it finds a valid solution. If there's no choice then it will return the original solution. Because we know that the neighbor is either one more than current solution, the same length of current solution, or one less than current solution, the best neighbor is one less than current solution. Therefore, as long as it finds a valid solution after removing a subset, it is the best neighbor.

4.4.4 Weakness and Strength

Random Restart Hill Climbing can escape from poor local optimum and try to find the best local optimum. Because it finds a random initial solution, it may find the optimum faster than simulated annealing because it does not follow a path to get the optimum. However, it takes chance. Sometimes it may not jump to a right path leading to the optimum. So it depends on the choice of random start. But based on the performance, this algorithm works relative well than other algorithms in runtime and quality. The algorithm also depends on the parameter like $max\_iteration$ and $track\_convergence$. With the larger $max\_iteration$ and $track\_convergence$, it will have a better result but also longer runtime.

Algorithm 6 Random Restart Hill Climbing

```
Require: Universe $U$, subsets $S = \{S_1, \ldots, S_m\}$,max_iteration $itr$,probability $P$, seed $seed$, cutoff time $T_{\text{cutoff}}$
Ensure: Minimum index set that covers all elements in $U$
 1: $current\_solution \leftarrow$ get_initial_solution$[U, S]$
 2: $best\_solution \leftarrow current\_solution$
 3: $track\_convergence \leftarrow 0$
 4: for $t \leftarrow 1$ to $itr$ or $T_{\text{cutoff}}$ do
 5:     if $track\_convergence > 100$ then
 6:         break
 7:     end if
 8:     $track\_convergence = track\_convergence + 1$
 9:     $current\_solution \leftarrow$ get_random_initial$[S, U]$
10:     $neighbor \leftarrow$ get_best_neighbor$[current\_solution, S, U]$
11:     while cost$[neighbor]$ < cost$[current\_solution]$ do
12:         $current\_solution \leftarrow neighbor$
13:         $neighbor \leftarrow$ get_best_neighbor$[current\_solution, S, U]$
14:     end while
15:     if cost$[neighbor]$ < cost$[best\_solution]$ then
16:         $track\_converge \leftarrow 0$
17:         $best\_solution \leftarrow neighbor$
18:     end if
19: end for
20: return best_solution
```

Algorithm 7 get_random_initial

```
Require: Universe $U$, subsets $S = \{S_1, \ldots, S_m\}$
Ensure: The initial solution of the problem
 1: $uncovered \leftarrow U$
 2: $solution \leftarrow []$
 3: $left\_subsets \leftarrow subsets$
 4: $random\_choice \leftarrow$ randomly choose an index of the subset
 5: add $random\_choice$ to $solution$
 6: $uncovered = uncovered - S[random\_choice]$
 7: remove $S[random\_choice]$ from $left\_subsets$
 8: while uncovered do
 9:     $best\_index \leftarrow$ the index of the subset that covers the most uncovered set
10:     add $best\_index$ to $solution$
11:     $uncovered = uncovered - S[best\_index]$
12: end while
13: return solution
```

Algorithm 8 get_best_neighbor

```
Require: current_solution $current\_solution$, Universe $U$, subsets $S = \{S_1, \ldots, S_m\}$
Ensure: The best neighbor
1: $neighbor \leftarrow current\_solution$
2: $choice \leftarrow neighbor$
3: $to\_remove$ = randomly choose one from choice
4: $test \leftarrow$ the list after removing $to\_move$ from $neighbor$
5: $choice \leftarrow test$
6: while $test$ is not a valid solution and len($choice$) $> 0$ do
7:     $to\_remove$ = randomly choose one from choice
8:     $test \leftarrow$ the list after removing $to\_move$ from $neighbor$
9:     $choice \leftarrow$ the list after removing $to\_move$ from $choice$
10: end while
11: if the test is a valid solution then
12:     $neighbor \leftarrow test$
13: end if
14: return $neighbor$
```

5 Empirical evaluation

In this section, we tested our algorithms based on the small-size and large-size data given. All experiments were implemented in Python 3.7 and were performed on macOS Sonoma14.1 system, with Apple M3 Pro chip, 18GB RAM. The development environment used was Visual Studio Code. The result is shown in table 1.

5.1 Experimental Procedure

Each algorithm is tested on 18 small-size problems and 12 large-size problems. Each problem provides the number of items and a list of subsets. Small-size problems have less than or equal to 20 elements and less than or equal to 20 subsets. Large-size problems have less than or equal to 2000 elements and less than or equal to 2000 subsets. Branch and Bound and Approximation ran each problem once. While two local search algorithms ran each problem 10 times with different seeds and then the average time, collection size, and relative error are recorded.

The time is calculated based on the difference of the time before the algorithm starts and the time after the algorithm start. However, the running time of local search algorithm are recorded based on the time of finding the best solution, which is the time recorded in the last line of trace file. Because algorithms stop if the best solution is not updated after $track\_convergence$ iterations. The total running time would be the found time of best solution plus the time running $track\_convergence$ iterations which is almost a constant around 20s for 1000 iterations. The table1 records the found time of best solution for two local search algorithms.

The relative error is used to compare the performance of each algorithms. It is computed as $\frac{Alg-OPT}{OPT}$ where Alg is the best solution returned from the algorithm i.e. the collection size in the table1, and OPT is the given optimum.

There are several parameters we set initially. The cutoff time is 1200s. The seeds used for local research algorithms are $[42, 100, 500, 20, 30, 50, 10, 90, 130, 190]$. Simulated Annealing has $initial\_temperature = 1$, $max\_iterations = 10000$,$track\_convergence = 1000$. Random Restart Hill Climbing has $max\_iterations = 10000$, $track\_convergence = 100$.

The $track\_convergence$ was tuned based on algorithms. Simulated Annealing has better solution when $track\_convergence = 1000$ while 100 is enough for Random Restart Hill Climbing i.e most solutions do not change if $track\_convergence = 1000$.

Table 1. Comprehensive table to compare the results of each algorithms

	Branch and Bound			Approximation			Hill Climbing			Simulated Annealing		
Dataset	Time (s)	collection size	RelErr	Time (s)	collection size	RelErr	Time (s)	collection size	RelErr	Time (s)	collection size	RelErr
small 1	0.0	5	0.0	0.0	5	0.0	0.00	5	0.0	0.0	5	0.0
small 2	0.0	3	0.0	0.0	4	0.33	0.0	3	0.0	0.00	3	0.0
small 3	0.00	5	0.0	0.0	6	0.2	0.00	5	0.0	0.00	5	0.0
small 4	0.0	4	0.0	0.0	5	0.25	0.0	4	0.0	0.00	4	0.0
small 5	0.00	5	0.0	0.0	6	0.2	0.00	5	0.0	0.00	5	0.0
small 6	0.0	3	0.0	0.0	4	0.33	0.00	3	0.0	0.00	3	0.0
small 7	0.0	3	0.0	0.0	4	0.33	0.0	3	0.0	0.00	3	0.0
small 8	0.0	2	0.0	0.0	3	0.5	0.0	2	0.0	0.00	2	0.0
small 9	0.0	3	0.0	0.0	4	0.33	0.0	3	0.0	0.000	3	0.0
small 10	0.0	2	0.0	0.0	3	0.5	0.0	2	0.0	0.00	2	0.0
small 11	0.00	4	0.0	0.0	5	0.25	0.0	4	0.0	0.00	4	0.0
small 12	0.00	3	0.0	0.0	4	0.33	0.00	3	0.0	0.00	3	0.0
small 13	0.0	2	0.0	0.0	3	0.5	0.0	2	0.0	0.00	2	0.0
small 14	0.0	2	0.0	0.0	3	0.5	0.0	2	0.0	0.00	2	0.0
small 15	0.0	2	0.0	0.0	3	0.5	0.00	2	0.0	0.00	2	0.0
small 16	0.0	2	0.0	0.0	3	0.5	0.0	2	0.0	0.00	2	0.0
small 17	0.0	2	0.0	0.0	3	0.5	0.00	2	0.0	0.00	2	0.0
small 18	0.0	2	0.0	0.0	3	0.5	0.0	2	0.0	0.00	2	0.0
large 1	1441.90	83	0.66	0.22	83	0.66	2.49	50	0.0	3.22	50	0.0
large 2	1297.21	21	0.11	0.00	21	0.11	0.01	20	0.05	0.01	20	0.05
large 3	1283.56	17	0.13	0.00	17	0.13	0.10	15.1	0.01	1.66	16.1	0.07
large 4	1538.70	153	0.68	0.07	153	0.68	4.04	147.1	0.62	12.32	150.8	0.66
large 5	1202.27	7	0.17	0.00	8	0.33	0.05	6.3	0.05	0.38	7	0.17
large 6	1212.03	7	0.17	0.04	7	0.17	0.17	6.1	0.02	0.00	7	0.17
large 7	1280.96	172	0.81	0.19	172	0.81	10.14	167.7	0.77	18.19	170.8	0.80
large 8	1.17	5	0.0	0.00	6	0.2	0.00	5	0.0	0.26	5.3	0.06
large 9	1357.18	16	0.14	0.00	16	0.14	0.05	15	0.07	0.77	15.6	0.11
large 10	1354.09	319	0.44	0.15	319	0.44	13.96	309.2	0.40	82.04	312.4	0.41
large 11	1382.43	56	0.4	0.04	56	0.4	1.15	53.8	0.35	0.04	56	0.4
large 12	1444.83	18	0.2	0.00	18	0.2	0.04	16	0.07	0.23	17.4	0.16

5.2 Empirical results

5.2.1 Branch and Bound

For small instances, the BnB method's RelErr are all zeroes and the running time is almost zero too, showing a good performance in solution quality and algorithm efficiency. For large instances, the biggest disadvantage for the algorithm appears - 11 out of 12 instances hit the cutoff time and the solution quality decreases with the maximum RelErr of 0.81. We should notice that instance large 8 also has a large dataset size, but BnB finds the optimal solution within 1.2 s, implying that the subset structure is of great importance.

5.2.2 Approximation

The greedy algorithm used for approximation excels in running time. It is pretty fast, no matter the test sample size is small or large. We can see that from large 1 to large 12, the running time is under 0.3 s which is really efficient. We can use approximation algorithm as a good starting point for other advanced algorithm. However, the greedy algorithm uses the maximum uncovered set as the criteria to select the subset which can't guarantee the optimal selection. When the size of subsets is large, the relative error is large.

5.2.3 Random Restart Hill Climbing

Random restart hill climbing shows a pretty good performance. Based on the relative error, most solution can reach near optimum or optimum. This shows that random restart is efficient jump out the poor local optimum and finds the optimum. I also ran hill climbing with random restart which has worse performance, which shows the needs of random restart although it takes longer time. Only problem large 10, large 11, large 4, and large 7 have relative large relative error. It may because the algorithm can not meet a good local optimum or it needs more iterations to find the optimum.

5.2.4 Simulated Annealing

The performance of Simulated Annealing is worse than expected. The runtime is larger than expected but the relative error is not better than expected. It may be because the swap and remove action in finding the neighbor need to loop for choosing an index until there's a valid solution but if it is on path far away from optimum, it may not reach the optimum in initialized iterations. Although the temperature allows the algorithm to explore a different direction other than the current best direction, for some problems with large size and more local optimums, it can not help a lot. For better results, we need to tune parameters or allow larger iterations or $track\_convergence$, but it also takes longer time.

5.3 Evaluation Plots for Local Search

Evaluation plots analyze the run time of local search on different problems. Each of the problem large 1 and large 10 was run 20 times with seeds
in $[42, 100, 500, 20, 30, 50, 10, 90, 130, 190, 1, 22, 46, 83, 93, 41, 105, 164, 194, 34]$ for Simulated Annealing and another 20 times with Random Restart Hill Climbing. Qualified Runtime for various solution qualities measures the distribution with respect to the run time with vary relative error. It is helpful to analyze how long to reach quality q^*. Solution Quality Distribution for various run times also shows how good is the solution after a fixed amount of time. Because there are only 20 times for each algorithm, the plots of QRTD and SQD are not smooth but more like step function. We may need to run more times to get smoother line. The box plots of run times show the mean and variance of the runtime for both problem.

5.3.1 Qualified Runtime for various solution qualities(QRTD)

Figure 1 and 2 compares runtime behavior between Simulated Annealing and Hill Climbing for problem large 1. Simulated Annealing shows a steeper curve after 2.5s which means it can reach the same quality with higher probability after 2.5s, while hill climbing has steeper curve before 2.5. So simulated annealing can reach certain quality faster after 2.5s, but hill climbing can reach certain quality faster before 2.5s. Hill Climbing shifts less widely, which shows that most cases reach a certain quality with lower time than Simulated Annealing. But the longer tail of hill climbing shows that several cases using hill climbing may take longer time to get the optimum. Therefore the upper bound of finding the optimum using Simulated Annealing is smaller.

Figure 3 and 4 shows the QRTD for problem large 10. We can see that the upper bound of relative error for local search on problem 10 is around 44% because both algorithm can reach quality 44% in almost 0s. In addition, the plot of Hill Climbing is steeper than Simulated Annealing, so Hill Climbing reaches a certain quality in faster time. The hill climbing can reach the quality 42% with probability 1 in 15s, while Simulated Annealing can not. Therefore, in this case, Hill Climbing is better than Simulated Annealing. Compared to problem large 1, it can not find the optimum, the smaller the relative error, the more time it needs to reach it.

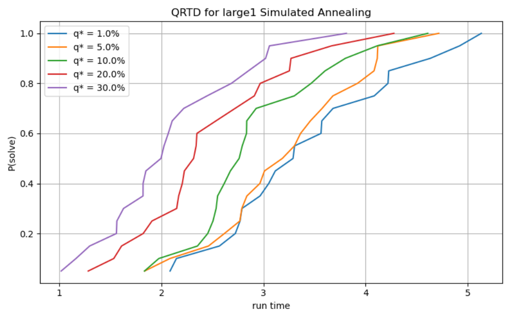

Fig. 1. QRTD for large1 using Simulated Annealing

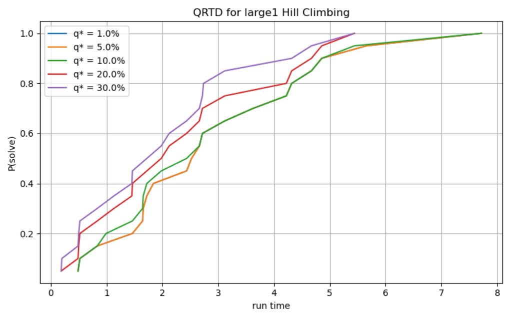

Fig. 2. QRTD for large1 using Hill Climbing

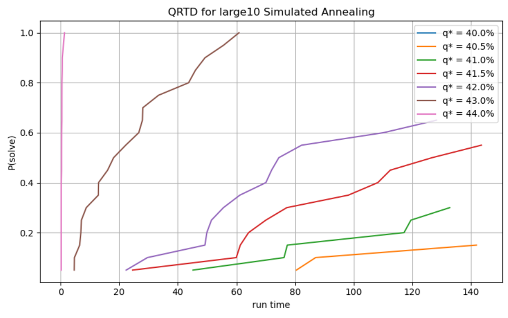

Fig. 3. QRTD for large10 using Simulated Annealing

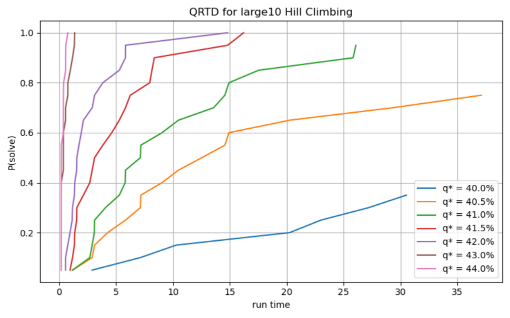

Fig. 4. QRTD for large10 using Hill Climbing

5.3.2 *Solution Quality Distribution for various run-times(SQD)*

Figure 5 and 6 are the SQD for problem large 1 using Simulated Annealing and Hill Climbing. It shows an upper bound of 8s to solve the problem. Simulated Annealing reach near optimal quality in around 4s. The lines of Simulated Annealing is steeper which means that it can find good solution faster. While Hill Climbing reaches a certain quality sooner for example, 0.3s can reach quality 20% with some probability and at 2s it can reach the near optimal solution with probability 40%, which is pretty good. But Simulated Annealing can not reach the near optimal solution in 2s. Therefore, Hill Climbing has probability to reach good solution with some probability faster due to the random start, but it takes less time for Simulated Annealing to reach the optimum with probability 1.

Figure 7 and 8 shows the SQD for problem large 10 using Simulated Annealing and Hill Climbing. Different from problem large 1, they have similar behavior. Both algorithm reach a lower bound of relative error that can reach with probability 1. The lower bound for Simulated Annealing is around 43%, and for is around 46% for Hill Climbing. Hill Climbing reaches a lower relative error with some probability within the same time. Therefore, in this case, Hill Climbing can reach a lower relative error at the same time, but when time is really close to 0, its quality is relatively low. This may because the random start of Hill Climbing would have a really bad quality solution.

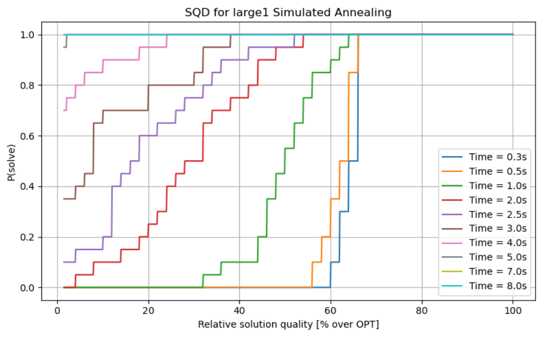

Fig. 5. SQD for large1 using Simulated Annealing

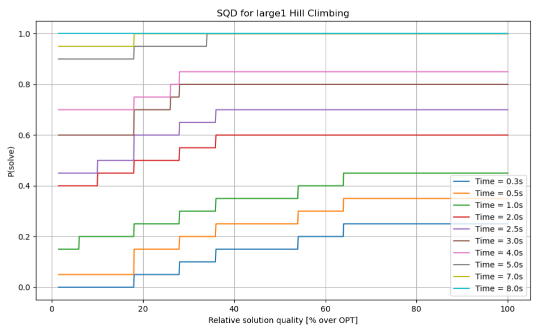

Fig. 6. SQD for large1 using Hill Climbing

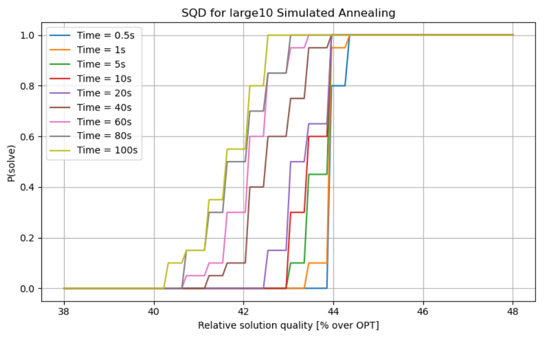

Fig. 7. SQD for large10 using Simulated Annealing

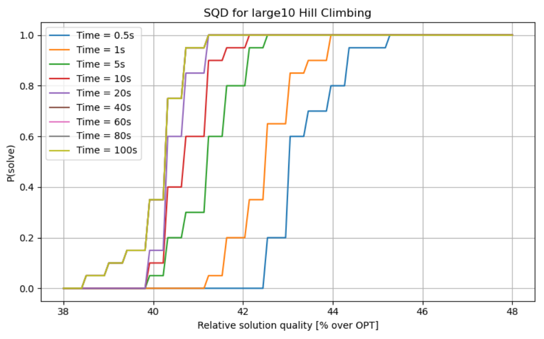

Fig. 8. SQD for large10 using Hill Climbing

5.3.3 Box plots for running times

The box plot 9 shows the runtime difference between simulated annealing and hill climbing for problem large 1. We can see that the mean of run time for both algorithms are similar, but the variance of climbing hills is larger. While the box plot 10 shows that the mean and variance of run time for both algorithm have large difference for problem large 10. The average runtime and variance of simulated are both larger than the average runtime of hill climbing. I think it is because large 1 can be solved optimally while large 10 can not. When the problem can be solved optimally, the seed can cause hill climbing jumps to different random start, some may jump to the random start that leads to the optimum quickly, some may take more jumps to to random start leads to the optimum. So the variance of hill climbing is larger. While if a problem can not be solved optimally, most random starts take similar time for reaching local optimum and it never reach the optimum, so it keeps jumping to random start no matter what the seed is. Therefore the variance of hill climbing is smaller in this case. Overall, the average time of solving the large size problem using hill climbing is smaller than that using simulated annealing and solving large 10 takes more time than solving large 1.

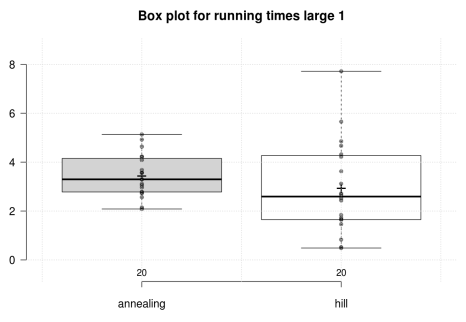

Fig. 9. Box plot for running time for large1

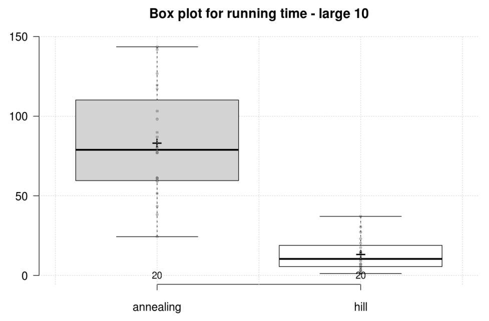

Fig. 10. Box plot for running time for large10

6 Discussion

In this section, we provide a comparative analysis of different algorithms for solving MSC problems with the evaluation criteria mentioned above.

6.1 Accuracy and Quality

Table2 shows the average relative error(RelErr) for each algorithm in our experiment. For small instances ($m, n \leq 20$), BnB and two Local Search algorithms all hit the optimal solution, while the Greedy Approximation algorithm has an average RelErr of 0.36. BnB method systematically enumerates all possible combinations of subsets, using lower and upper bounds to prune branches; this guarantees a global optimal solution if run to completion. Local Search - Hill Climbing repeatedly moves to the best "neighbor" solution by swapping sets in and out; on small problem sizes, this process can effectively explore all high-quality regions of the solution space and converge to the global optimal solution. In small instances, Local Search - Simulated Annealing typically visits nearly every promising configuration and ultimately settles on the true optimal solution, although which is not guaranteed. However, the Greedy Approximation method uses a purely myopic rule, choosing the set that covers the most currently uncovered elements without any lookahead or backtracking, which can force suboptimal early choices and thus a relative error even on these small datasets.

For large instances ($m, n \leq 2,000$), all four algorithms produce RelErr to different degrees. The Greedy Approximation method still yields the highest average RelErr (0.36). The BnB is right after the approximation with an average RelErr of 0.33 despite being exact in theory, because this method hits the cut-off time on most of the large instances and returns the current best solution, instead of the optimal solution. The Simulated Annealing method can achieve a relatively lower average RelErr(0.26), benefiting from escaping poor local choices. The Hill Climbing achieves the lowest average RelErr (0.20). At each step, Hill Climbing picks the single move that gives the largest immediate drop in cover size and keep choosing random initial to compare multiple local optimums, which is a distinct feature from other algorithms. By always taking the best possible swap and search for the best local optimum, this method greatly increases the solution accuracy.

Table 2. Average Relative Error for Each Algorithm

Algorithm	Small	Large	Total
Branch and Bound	0.00	0.33	0.13
Approximation	0.36	0.36	0.36
Hill Climbing	0.00	0.20	0.08
Simulated Annealing	0.00	0.26	0.10

6.2 Scalability and Efficiency

No running time difference has been found for a small instance experiment. However, when the dataset size grows to a large set, an obvious difference appears. We ran all four algorithms on large instances under a hard cutoff of 1,200 seconds. The number of the subsets varies from 200 to 2,000, and the universe size varies from 50 to 2,000. Plot 11 and 12 show the the four algorithms' scability performance with number of subsets *m* and the universe size *n*.
For the scalability performance, we have following key observations:

- BnB's running time climbs dramatically with the increase of either the number of subsets or the universe size, compared to small instances. It only found the optimum in one large instance within the cutoff time, others all hit the cutoff time, showing this algorithm's exponential inefficient behavior as we expected from an exact search method.
- The Greedy Approximation method takes the shortest time and finishes under 1s for every large instance, as it grows marginally, consistent with its near-linear dependence on the total number of set-element pairs.
- Hill Climbing only examines a few neighbor swaps for each iteration, thus, its runtime scales mildly with *m* and *n*.
- Simulated Annealing generally is slower than Approximation and Hill Climbing, but is still not inefficient. Its scalability plot shows a 'hump' shape, indicating that moderate-sized problems require many cooling-schedule steps to escape local traps, whereas very large ones often hit the iteration cap[1].

Table 3. Average Runtime on Large Instances (in seconds)

Algorithm	Runtime
Branch and Bound	1233.03
Approximation	0.059
Hill Climbing	2.68
Simulated Annealing	9.93

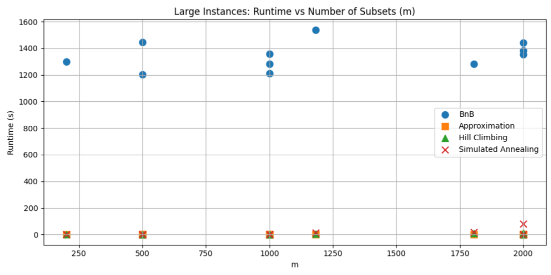

Fig. 11. Runningtime vs Number of Subsets (m) - Large Instances

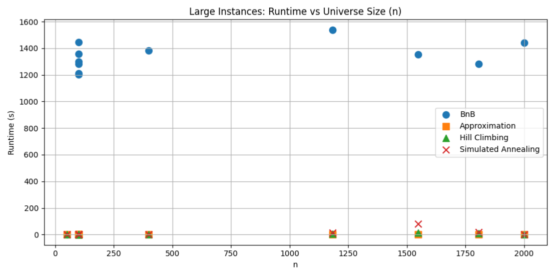

Fig. 12. Runningtime vs Universe Size (n) - Large Instances

Complexity vs. Empirical Runtime

Local Search algorithms don't always have a well-defined time complexity because their runtime depends on how the search progresses. Therefore, we compared the expected time complexity with the actual running time only for the BnB and Greedy Approximation algorithms. We convert the time complexity of these two algorithms into idealized clock times using CPU 4.05 GHz core. Both BnB and Greedy Approximation algorithms run faster than the theoretical expected time. The theoretical time for BnB is exponential and too large to visualize on the plot, therefore, we adopt logarithmic for the y-axis. Due to the design of the algorithms - like pruning, and the actual subset structure, the actual running efficiency is better than the expected time complexity.

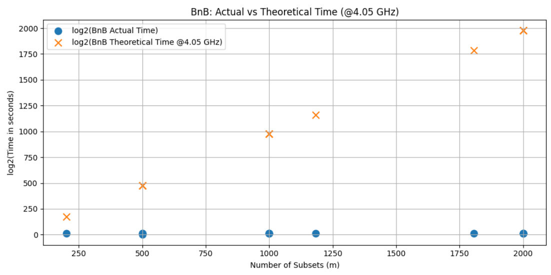

Fig. 13. BnB Runningtime vs Number of Subsets (m) - Large Instances

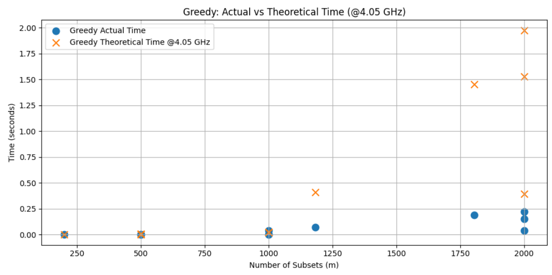

Fig. 14. Approximation Runningtime vs Number of Subsets (m) - Large Instances

7 Conclusion

For small instances, all four algorithms can achieve both high accuracy and efficiency, but BnB has a distinct advantage that it guarantees optimality. For large instances, Hill Climbing provides a very good balance on both solution quality and scalability efficiency. It consistently finds high-quality covers much faster than other algorithms for large dataset. Simulated Annealing's solution quality and algorithm efficiency processing large instances are both worse than Hill Climbing. The Approximation has the worst empirical quality, however, it is the only method among the four with a provable worst-case bound.

References

[1] David Abramson, Mohan Krishnamoorthy, Henry Dang, et al. Simulated annealing cooling schedules for the school timetabling problem. *Asia Pacific Journal of Operational Research*, 16:1–22, 1999.

[2] Jafar Bagherinejad, Mehdi Seifbarghy, and Mahnaz Shoeib. Developing dynamic maximal covering location problem considering capacitated facilities and solving it using hill climbing and genetic algorithm. *Engineering Review: Međunarodni časopis namijenjen publiciranju originalnih istraživanja s aspekta analize konstrukcija, materijala i novih tehnologija u području strojarstva, brodogradnje, temeljnih tehničkih znanosti, elektrotehnike, računarstva i građevinarstva*, 37(2):178–193, 2017.

[3] Reuven Bar-Yehuda and Shimon Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981.

[4] Koen Bontridder, B. Lageweg, Jan Lenstra, James Orlin, and Leen Stougie. Branch-and-bound algorithms for the test cover problem. pages 223–233, 09 2002.

[5] Koen Bontridder, B. Lageweg, Jan Lenstra, James Orlin, and Leen Stougie. Branch-and-bound algorithms for the test cover problem. pages 223–233, 09 2002.

[6] Marco Chiarandini, Thomas Stützle, et al. An application of iterated local search to graph coloring problem. In *Proceedings of the computational symposium on graph coloring and its generalizations*, pages 112–125. Ithaca New York (USA), 2002.

[7] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[8] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks, 2019.

[9] He He, Hal Daumé, and Jason Eisner. Learning to search in branch-and-bound algorithms. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3293–3301, Cambridge, MA, USA, 2014. MIT Press.

[10] Dorit S Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. *Approximation algorithms for NP-hard problems*, pages 94–143, 1997.

[11] MS Jabalameli, B Bankian Tabrizi, and Mohammad Moshref Javadi. A simulated annealing method to solve a generalized maximal covering location problem. *Int J Ind Eng Comput*, 2:439–448, 2011.

[12] Gio K. Kao, Edward C. Sewell, and Sheldon H. Jacobson. A branch, bound, and remember algorithm for the 1|ri|tischeduling problem. *Journal of Scheduling*, 12(2):163–175, 2009.

[13] Scott Kirkpatrick, Charles D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220:671 – 680, 1983.

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

[15] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[16] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.

[17] David R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe, and Edward C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016.

[18] David R. Morrison, Jason J. Sauppe, Edward C. Sewell, and Sheldon H. Jacobson. A wide branching strategy for the graph coloring problem. *INFORMS Journal on Computing*, 26(4):704–717, 2014.

[19] Kaan Yilmaz and Neil Yorke-Smith. A study of learning search approximation in mixed integer branch and bound: Node selection in scip. *AI*, 2(2):150–178, April 2021.