CS301

2022-2023 Spring

Project Report

Group 081

Sude Buket Kipri

Damla Yıldıztekin

1. Problem Description

Intuitive Description:

The minimum cover problem asks to find the smallest number of sets from a given collection of sets that can cover all the elements in a finite set. In other words, we want to find the minimum number of sets from C that can completely cover all elements in S. This problem arises in many real-world scenarios, such as scheduling problems, database design, and network optimization. This problem can be thought of as finding the minimum number of resources required to complete a task or the minimum number of tests required to cover all possible scenarios.

Formal Definition:

Given a finite set S and a collection *C = (C1,...,Cm)* of subsets of S, the minimum cover problem is to find the smallest subset *D* = {*D1,D2,...,Dk*} of C such that each element in S belongs to at least one set in D.

Applications:

Applications include operations research, computer science, and bioinformatics. The minimum cover problem has several uses in these areas. In operations research, it is used to find the minimum number of resources required to complete a task or the minimum number of machines required to cover all the jobs. It is used in data compression in computer science, where the challenge is to determine the smallest number of codewords necessary to represent all the symbols in a particular collection. The issue in bioinformatics emerges during the analysis of gene expression data, when the objective is to identify the fewest possible genes that can adequately account for the observed expression patterns. It can also be used to optimize resource allocation in a production line or a transportation system. In database systems, it can help in optimizing the database queries, and in scheduling problems, it can aid in optimizing the schedules of tasks to be performed.

Hardness Theorem:

The minimum cover problem is NP-Complete. The decision version of minimum cover is NP-complete, and the optimization/search version of minimum cover is NP-hard[1]. The one in the annotation website is the decision version, the one on SuCourse is a search problem, therefore if we consider the one on SuCourse, we have the corresponding optimization problem which is NP-hard.

Proof:

To prove that the Minimum Cover problem is NP-complete, we need to show that it is in the class of NP problems and that it is NP-hard. The detailed proofs of NP-completeness for the Minimum Cover problem can be found in the book *"Computers and Intractability: A Guide to the Theory of NP-Completeness" by Garey and Johnson*[2]. And *"Covering a Set with Arithmetic Progressions is NP-Complete" by Heath and Lenwood*[3]. In addition, the solution and proof can also be found in *"Solving the set cover problem and the problem of exact cover by 3-sets in the Adleman–Lipton" model by Chang and Guo*[4]. In the textbook of the CS301 course, *Cormen et al.* also define this problem as *NP-Complete in Introduction to Algorithms*[5]. *Karp* also discusses this problem is NP-Complete in his seminal paper *Reducibility among combinatorial problems by performing polynomial-time reductions.*[6]

---

[1] Korte, Bernhard; Vygen, Jens (2012), *Combinatorial Optimization: Theory and Algorithms* (5 ed.), Springer, ISBN 978-3-642-24487-2

[2] Garey, Michael R., and David S. Johnson. *Computers and intractability*. Vol. 174. San Francisco: freeman, 1979.

[3] Heath, Lenwood S. *Covering a set with arithmetic progressions is NP-complete*. Department of Computer Science, Virginia Polytechnic Institute & State University, 1989.

[4] Chang, Weng-Long, and Minyi Guo. "Solving the set cover problem and the problem of exact cover by 3-sets in the Adleman–Lipton model." *BioSystems* 72, no. 3 (2003): 263-275.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

[6] Karp, R. M. (1972). Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher, & J. D. Bohlinger (Eds.), Complexity of Computer Computations (pp. 85-103). Springer US.

2. Algorithm Description

a. Brute Force Algorithm

The minimum set cover problem's brute force algorithm entails creating all possible subsets of the collection C and determining which of them covers the complete set S. This algorithm is inefficient and has an exponential time complexity.

Pseudo-code for the algorithm:

```
function brute_force_minimum_set_cover(S, C):
    min_cover = []
    min_cover_size = infinity

    for I in range(2^lenI):
        candidate_cover = []
        for j in range(lenI):
            if I & (1 << j):
                candidate_cover.append(C[j])

        if is_valid_cover(S, candidate_cover):
            if len(candidate_cover) < min_cover_size:
                min_cover = candidate_cover
                min_cover_size = len(candidate_cover)

    return min_cover

function is_valid_cover(S, candidate_cover):
    covered = set()
    for subset in candidate_cover:
        covered != subset

    return covered == S
```

Explanation of the algorithm:

1) Set min_cover_size to infinite and initialize min_cover (D) to be empty.
2) Loop through every integer from 0 to 2m – 1, where m is the collection C's subset count.
3) Create a candidate cover for each integer by include the subsets of C that correspond to the set bits in the integer's binary representation.
4) Use the is_valid_cover function to determine whether the candidate cover is a valid cover for set S. Update the min_cover and min_cover_size if the candidate cover is less than the current minimum cover size.
5) After examining every combination, return the min_cover.

No specific algorithm design method, such as divide-and-conquer, dynamic programming, or greedy algorithms, is used in the brute force algorithm. It simply checks each and every combination of subsets in collection C. The algorithm ensures a solution, but because of its exponential time complexity, it is computationally expensive.

b. Heuristic Algorithm

The greedy algorithm is a natural choice for the set cover problem, as it makes the locally optimal choice at each step in the hope of finding a global optimum. It is also considerably more efficient than the brute force algorithm. The greedy algorithm doesn't guarantee the absolute minimum set cover, but it does provide a good approximation.

Algorithm description:

The greedy algorithm for the minimum set cover problem works as follows:

1) Start with an empty cover.

2) While there are elements in the universal set S that are not yet covered:

a. Choose a subset C_i from the collection of subsets C that maximizes the number of uncovered elements in S.

b. Add C_i to the cover and remove the elements of C_i from S.

3) Return the cover.

This algorithm makes a greedy choice in each iteration by choosing the subset that covers the most uncovered elements. It continues until all elements in the set S are covered.

Pseudo-code for the algorithm:

```
function GreedySetCover(S, C):
    D = {}  # Initialize an empty set D
    while S is not empty:
        C_i = set in C that covers the maximum number of uncovered elements in S # Choose the subset with the maximum number of uncovered elements
        D = D ∪ {C_i} # Add the chosen subset to D
        S = S - C_i # Remove the covered elements from U
    return D
```

Explanation of the pseudo-code:

1) Initialize the cover as an empty set.

2) While there are still uncovered elements in the set U (initially all S):

a. Choose the subset in the collection C that covers the most uncovered elements (using the select_maximal_subset function).

b. Remove the elements of the chosen subset from the set U.

c. Add the chosen subset to the cover.

3) Return the cover.

Time complexity:

The greedy algorithm is more efficient than the brute force algorithm. It has a time complexity of O(n*m) in the worst case, where n is the size of the universal set S and m is the size of the collection C.

Approximation Ratio:

The greedy algorithm for the minimum set cover problem is a log(n)-approximation algorithm, where n is the size of the universal set S. This means that the size of the cover found by the algorithm is at most log(n) times the size of the minimum possible cover.

Proof:

Let OPT be the size of the minimum cover, and let ALG be the size of the cover found by the greedy algorithm. Consider the iteration in which the greedy algorithm selects a subset. At this point, at least 1/OPT of the elements are still uncovered, because otherwise there would be a subset in the optimal cover that covers only already covered elements. Therefore, the greedy algorithm selects a subset that covers at least a 1/OPT fraction of the remaining elements. After at most OPT log(n) iterations, the number of uncovered elements is reduced by a constant factor, and all elements are covered. Therefore, ALG <= OPT log(n), which proves the log(n)-approximation ratio of the greedy algorithm. Stern demonstrates the proof of the greedy heuristic algorithm with further detail in the *"MIT Seminar of Theoretical Computer Science: Set Cover Problem"*. Chvatal discussed the proof of the greedy heuristic algorithm in the article *"A Greedy Heuristic for the Set-Covering Problem."*[7] Akhtar also provided the algorithm along with the proof in her article "A heuristic approach for minimum set cover problem."[8] Williamson and Shmoys covered this topic in their book *The Design of Approximation Algorithms*[9], showing the greedy algorithm and proof.

3. Algorithm Analysis

a. Brute Force Algorithm

Theorem:

The brute force algorithm provided for the minimum set cover problem correctly finds the smallest cardinality subset D of C such that every element in S belongs to at least one set in D.

Proof:

By contradiction, we establish the theorem. Think about a scenario in which the Brute Force Set Cover technique does not provide the smallest cardinality subset D of C that completely covers the set S. This indicates that a subset D' of C exists that has a less cardinality than the answer the

[7] Chvatal, V. "A Greedy Heuristic for the Set-Covering Problem." Mathematics of Operations Research 4, no. 3 (1979): 233–35. http://www.jstor.org/stable/3689577.

[8] Akhter, Fatema. "A heuristic approach for minimum set cover problem." Int. J. Adv. Res. Artif. Intell 4 (2015): 40-45.

[9] Williamson, David P., and David B. Shmoys. *The design of approximation algorithms.* Cambridge university press, 2011.

algorithm produced and whose union of subsets encompasses the whole set S. However, because our algorithm loops through all combinations of the C subsets, it must have also come across the combination D'. The algorithm should have discovered that the cardinality of D' is smaller than the best cover discovered so far and that the union of subsets in D' covers the complete set S when it came across D'. This goes against our presumption that the procedure does not return the smallest cardinality subset D of C that covers the complete collection S. Therefore, the smallest cardinality subset D of C that completely covers the set S is returned by the Brute Force Minimum Set Cover technique.

Time complexity:

The collection C's subsets can be combined in any way possible according to the Brute Force Minimum Set Cover technique. 2^m is the number of subsets in the collection C, and m is the number of combinations that can be made. The method examines each combination to see if the union of its subsets encompasses the complete set S. The worst-case time complexity of the procedure is O(n * 2^m), assuming that the time complexity of checking the union is O(n), where n is the size of the collection S. This is a tight upper bound because the algorithm must investigate every conceivable subset combination in order to ensure the best outcome.

Space complexity:

The amount of storage space needed to hold the combinations of the subsets in the collection C mostly determines the space complexity of the Brute Force Minimum Set Cover algorithm. The algorithm might, in the worst scenario, require to store all 2 m combinations, resulting in a space complexity of O(2^m). In addition, we must keep track of the min_cover discovered thus far, which, in the worst scenario, may be m subsets. As a result, the total complexity of space is O(2^m + m).

b. Heuristic Algorithm

Theorem:

The greedy algorithm for the minimum set cover problem correctly finds a subset D of C such that every element in S belongs to at least one set in D. It is a (1 + ln n)-approximation algorithm for the set cover problem.

Proof:

The algorithm selects at each step the subset that covers the maximum number of uncovered elements from the set S. It continues until all elements in S are covered. Therefore, at the end of the algorithm, every element in S belongs to at least one set in the cover D. This proves the correctness of the algorithm.

Time Complexity:

The greedy algorithm involves a loop that continues until all elements in S are covered. In each iteration of the loop, it must find the subset in C that covers the maximum number of uncovered elements. This involves iterating over all subsets in C and all elements in the set S, which takes time proportional to the product of their sizes. Therefore, it runs in polynomial time with a worst-case time complexity of the algorithm is O(n*m), where n is the size of the universal set S and m is the size of the collection C. This is a tight upper bound because in the worst case the algorithm must examine all subsets in C and all elements in S.

Space Complexity:

The space complexity of the algorithm is determined by the space required to store the set S and the collection C, as well as the cover D. The space required for S and C is given as input, and the space required for D can be at most the same as the space required for C. Therefore, the space complexity of the algorithm is O(n + m), where n is the size of the universal set S and m is the size of the collection C. The approximation factor of ln(n) is a proven guarantee, so we know that the solution is at most ln(n) times larger than the optimal solution.

The greedy algorithm for the minimum set cover problem has a good balance between efficiency and approximation quality. While it does not guarantee the minimum set cover, it does provide a cover whose size is at most a logarithmic factor larger than the minimum. This makes it a good choice for large instances of the problem where the exact minimum set cover is not required and the brute force algorithm is not feasible due to its high time complexity. The quality of the solution can be impacted by the initial ordering of the sets, which is highly relied upon by the greedy algorithm. Even for modest values of n, the approximation factor of ln(n) can still be significant.

4. Sample Generation (Random Instance Generator)

Pseudo-code for the algorithm:

```
function generateRandomInstance(numSets, numElements):
    elements = generateRandomElements(numElements)
    sets = []
    while not isCovered(sets, elements):
        sets = generateRandomSets(numSets, elements)
    return (sets, elements)

function generateRandomElements(numElements):
    elements = []
    for i in range(numElements):
        element = generateRandomElement()
        elements.append(element)
    return elements

function generateRandomSets(numSets, elements):
    sets = []
    for i in range(numSets):
        subset = generateRandomSubset(elements)
        sets.append(subset)
    return sets
```

```
function generateRandomSubset(elements):
    subset = []
    for element in elements:
        if random() < 0.5:
            subset.append(element)
    return subset

function isCovered(sets, elements):
    covered = set()
    for s in sets:
        covered.update(s)
    return set(elements) == covered
```

Explanation of the Algorithm:

1. Determine the size of the problem (e.g., number of sets, number of elements)
2. Randomly generate a set of elements (e.g., integers, strings) that the sets will be constructed from
3. Randomly generate a set of sets, with each set containing a random subset of the elements
4. Check if the generated sets cover all elements. If not, repeat steps 3-4 until a valid instance is generated.

Python code for the instance generator:

```
import random

def random_set_cover_input(n, m, max_element=100):
    # Generate the universal set S by sampling n unique elements from the
range 1 to max_element
    S = set(random.sample(range(1, max_element + 1), n))

    # Generate the collection of subsets C
    C = []
    for _ in range(m):
        subset_size = random.randint(1, n)
        subset = random.sample(list(S), subset_size)
        C.append(set(subset))

    return S, C
# Generate 20 random set cover problem instances with |S|=10, |C|=5, and
max_element=100
random_instances = [random_set_cover_input(10, 5, 100) for _ in range(20)]

# Print the generated instances
for i, instance in enumerate(random_instances, 1):
    S, C = instance
    print(f"Instance {i}:")
    print("Universal Set S:", S)
    print("Collection of Subsets C:", C)
    print()
```

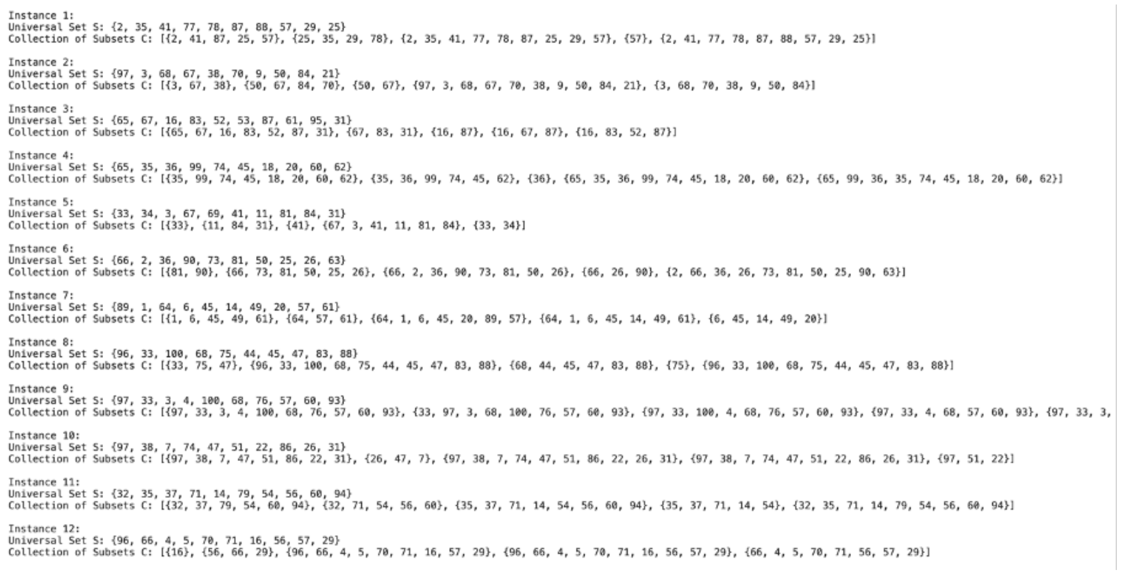
This algorithm generates random instances of the minimum set cover problem. The inputs for the algorithm are the size of the universal set S (n), the size of the collection of subsets C (m), and the maximum integer that can be included in the universal set S (max_element, default is 100). The algorithm consists of two main parts: generating the universal set S and generating the collection of subsets C.

Generate the universal set S: The universal set S is generated by randomly sampling n unique elements from the range of integers between 1 and max_element (inclusive). The random.sample function is used to achieve this.

Generate the collection of subsets C: The collection of subsets C is generated by iterating m times, where m is the number of subsets to be created. For each iteration, the algorithm randomly selects a subset size between 1 and n (inclusive) and then samples the given number of elements from the universal set S without replacement. The sampled elements are then combined to form a subset, which is added to the collection C.

Return the universal set S and the collection of subsets C: After generating both the universal set S and the collection of subsets C, the algorithm returns them as a tuple (S, C).

The last part of the code generates 20 random instances using the random_set_cover_input function and prints the instances. After running the code given above, we get the following instances. We can modify the sizes of sets and number of instances as we wish.



```
Instance 1:
Universal Set S: {2, 35, 41, 77, 78, 87, 88, 57, 29, 25}
Collection of Subsets C: [{2, 41, 87, 25, 57}, {25, 35, 29, 78}, {2, 35, 41, 77, 78, 87, 25, 29, 57}, {57}, {2, 41, 77, 78, 87, 88, 57, 29, 25}]

Instance 2:
Universal Set S: {97, 3, 68, 67, 38, 70, 9, 50, 84, 21}
Collection of Subsets C: [{3, 67, 38}, {50, 67, 84, 70}, {50, 67}, {97, 3, 68, 67, 70, 38, 9, 50, 84, 21}, {3, 68, 70, 38, 9, 50, 84}]

Instance 3:
Universal Set S: {65, 67, 16, 83, 52, 53, 87, 61, 95, 31}
Collection of Subsets C: [{65, 67, 16, 83, 52, 87, 31}, {67, 83, 31}, {16, 87}, {16, 67, 87}, {16, 83, 52, 87}]

Instance 4:
Universal Set S: {65, 35, 36, 99, 74, 45, 18, 20, 60, 62}
Collection of Subsets C: [{35, 99, 74, 45, 18, 20, 60, 62}, {35, 36, 99, 74, 45, 62}, {36}, {65, 35, 36, 99, 74, 45, 18, 20, 60, 62}, {65, 99, 36, 35, 74, 45, 18, 20, 60, 62}]

Instance 5:
Universal Set S: {33, 34, 3, 67, 69, 41, 11, 81, 84, 31}
Collection of Subsets C: [{33}, {11, 84, 31}, {41}, {67, 3, 41, 11, 81, 84}, {33, 34}]

Instance 6:
Universal Set S: {66, 2, 36, 90, 73, 81, 50, 25, 26, 63}
Collection of Subsets C: [{81, 90}, {66, 73, 81, 50, 25, 26}, {66, 2, 36, 90, 73, 81, 50, 26}, {66, 26, 90}, {2, 66, 36, 26, 73, 81, 50, 25, 90, 63}]

Instance 7:
Universal Set S: {89, 1, 64, 6, 45, 14, 49, 20, 57, 61}
Collection of Subsets C: [{1, 6, 45, 49, 61}, {64, 57, 61}, {64, 1, 6, 45, 20, 89, 57}, {64, 1, 6, 45, 14, 49, 61}, {6, 45, 14, 49, 20}]

Instance 8:
Universal Set S: {96, 33, 100, 68, 75, 44, 45, 47, 83, 88}
Collection of Subsets C: [{33, 75, 47}, {96, 33, 100, 68, 75, 44, 45, 47, 83, 88}, {68, 44, 45, 47, 83, 88}, {75}, {96, 33, 100, 68, 75, 44, 45, 47, 83, 88}]

Instance 9:
Universal Set S: {97, 33, 3, 4, 100, 68, 76, 57, 60, 93}
Collection of Subsets C: [{97, 33, 3, 4, 100, 68, 76, 57, 60, 93}, {33, 97, 3, 68, 100, 76, 57, 60, 93}, {97, 33, 100, 4, 68, 76, 57, 60, 93}, {97, 33, 4, 68, 57, 60, 93}, {97, 33, 3,

Instance 10:
Universal Set S: {97, 38, 7, 74, 47, 51, 22, 86, 26, 31}
Collection of Subsets C: [{97, 38, 7, 47, 51, 86, 22, 31}, {26, 47, 7}, {97, 38, 7, 74, 47, 51, 86, 22, 26, 31}, {97, 38, 7, 74, 47, 51, 22, 86, 26, 31}, {97, 51, 22}]

Instance 11:
Universal Set S: {32, 35, 37, 71, 14, 79, 54, 56, 60, 94}
Collection of Subsets C: [{32, 37, 79, 54, 60, 94}, {32, 71, 54, 56, 60}, {35, 37, 71, 14, 54, 56, 60, 94}, {35, 37, 71, 14, 54}, {32, 35, 71, 14, 79, 54, 56, 60, 94}]

Instance 12:
Universal Set S: {96, 66, 4, 5, 70, 71, 16, 56, 57, 29}
Collection of Subsets C: [{16}, {56, 66, 29}, {96, 66, 4, 5, 70, 71, 16, 57, 29}, {96, 66, 4, 5, 70, 71, 16, 56, 57, 29}, {66, 4, 5, 70, 71, 56, 57, 29}]
```

5. Algorithm Implementations

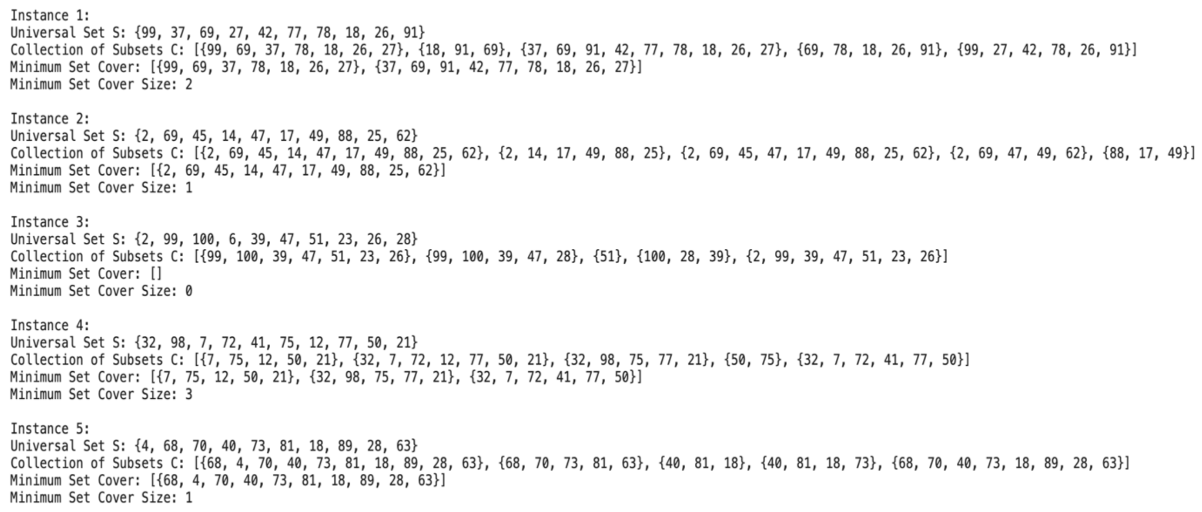
a. Brute Force Algorithm

Algorithm Implementation:

```
# Brute force minimum set cover algorithm
def brute_force_minimum_set_cover(S, C):
    min_cover = []
    min_cover_size = float('inf')

    for i in range(2**len(C)):
        candidate_cover = []
        for j in range(len(C)):
            if i & (1 << j):
                candidate_cover.append(C[j])

        if is_valid_cover(S, candidate_cover):
            if len(candidate_cover) < min_cover_size:
                min_cover = candidate_cover
                min_cover_size = len(candidate_cover)
    return min_cover

# Check if candidate cover is valid
def is_valid_cover(S, candidate_cover):
    covered = set()
    for subset in candidate_cover:
        covered |= subset
    return covered == S

# Generate 15 random instances and test the algorithm
random_instances = [random_set_cover_input(10, 5, 100) for _ in range(15)]

for i, instance in enumerate(random_instances, 1):
    S, C = instance
    min_cover = brute_force_minimum_set_cover(S, C)
    print(f"Instance {i}:")
    print("Universal Set S:", S)
    print("Collection of Subsets C:", C)
    print("Minimum Set Cover:", min_cover)
    print("Minimum Set Cover Size:", len(min_cover))
    print()
```

Explanation of the Algorithm:

After running the code given above, we get the following outputs. This code defines the random_set_cover_input, brute_force_minimum_set_cover, and is_valid_cover functions in Python. It generates 15 random instances of the minimum set cover problem with S having size 10, elements in S ranging from 1 to 100 and C having size 5. Then, it tests the brute_force_minimum_set_cover function on them. The results include the universal set S, the collection of subsets C, the minimum set cover D, and the minimum set cover size. There are no failures.



```
Instance 1:
Universal Set S: {99, 37, 69, 27, 42, 77, 78, 18, 26, 91}
Collection of Subsets C: [{99, 69, 37, 78, 18, 26, 27}, {18, 91, 69}, {37, 69, 91, 42, 77, 78, 18, 26, 27}, {69, 78, 18, 26, 91}, {99, 27, 42, 78, 26, 91}]
Minimum Set Cover: [{99, 69, 37, 78, 18, 26, 27}, {37, 69, 91, 42, 77, 78, 18, 26, 27}]
Minimum Set Cover Size: 2

Instance 2:
Universal Set S: {2, 69, 45, 14, 47, 17, 49, 88, 25, 62}
Collection of Subsets C: [{2, 69, 45, 14, 47, 17, 49, 88, 25, 62}, {2, 14, 17, 49, 88, 25}, {2, 69, 45, 47, 17, 49, 88, 25, 62}, {2, 69, 47, 49, 62}, {88, 17, 49}]
Minimum Set Cover: [{2, 69, 45, 14, 47, 17, 49, 88, 25, 62}]
Minimum Set Cover Size: 1

Instance 3:
Universal Set S: {2, 99, 100, 6, 39, 47, 51, 23, 26, 28}
Collection of Subsets C: [{99, 100, 39, 47, 51, 23, 26}, {99, 100, 39, 47, 28}, {51}, {100, 28, 39}, {2, 99, 39, 47, 51, 23, 26}]
Minimum Set Cover: []
Minimum Set Cover Size: 0

Instance 4:
Universal Set S: {32, 98, 7, 72, 41, 75, 12, 77, 50, 21}
Collection of Subsets C: [{7, 75, 12, 50, 21}, {32, 7, 72, 12, 77, 50, 21}, {32, 98, 75, 77, 21}, {50, 75}, {32, 7, 72, 41, 77, 50}]
Minimum Set Cover: [{7, 75, 12, 50, 21}, {32, 98, 75, 77, 21}, {32, 7, 72, 41, 77, 50}]
Minimum Set Cover Size: 3

Instance 5:
Universal Set S: {4, 68, 70, 40, 73, 81, 18, 89, 28, 63}
Collection of Subsets C: [{68, 4, 70, 40, 73, 81, 18, 89, 28, 63}, {68, 70, 73, 81, 63}, {40, 81, 18}, {40, 81, 18, 73}, {68, 70, 40, 73, 18, 89, 28, 63}]
Minimum Set Cover: [{68, 4, 70, 40, 73, 81, 18, 89, 28, 63}]
Minimum Set Cover Size: 1
```

b. Heuristic Algorithm

Algorithm Implementation:

```python
def greedy_set_cover(S, C):
    covered_elements = set()
    selected_sets = []

    while covered_elements != S:
        max_set = None
        max_new_elements = 0

        for subset in C:
            new_elements = len(set(subset) - covered_elements)
            if new_elements > max_new_elements:
                max_new_elements = new_elements
                max_set = subset

        if max_set is None:
            return None

        covered_elements.update(max_set)
        selected_sets.append(max_set)
        C.remove(max_set)

    return selected_sets

for i, instance in enumerate(random_instances, 1):
    S, C = instance
    min_cover = greedy_set_cover(S, C)
    print(f"Instance {i}:")
    print("Universal Set S:", S)
    print("Collection of Subsets C:", C)
    print("Minimum Set Cover:", min_cover)
    print("Minimum Set Cover Size:", len(min_cover) if min_cover is not
None else 0)
    print()
```

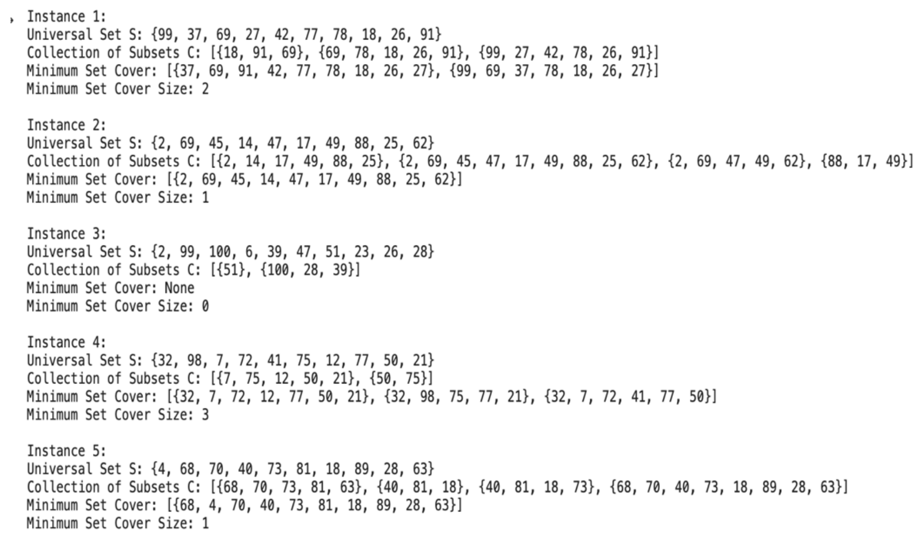
Explanation of the Algorithm:

Initialization: The greedy_set_cover function starts by initializing covered_elements as an empty set. This will be used to keep track of the elements that have already been covered by the chosen subsets. selected_sets is also initialized as an empty list. This will hold the subsets that are chosen to be part of the set cover.

Greedy selection: In each iteration of the while loop, the function goes through all subsets in C to find the subset that covers the maximum number of currently uncovered elements in S. This is done by creating a set of new_elements as the difference between the current subset and the covered_elements. If the number of new_elements is greater than max_new_elements, then the current subset is chosen as max_set. This is the greedy selection part of the algorithm.
Update the covered elements and selected sets: If a max_set is found, its elements are added to covered_elements, and the set itself is added to selected_sets. The max_set is also removed from C, so it won't be chosen again in subsequent iterations.

Termination: The while loop continues until all elements of S are covered (covered_elements == S). At this point, selected_sets will be the sub-collection of C that covers all elements in S with the smallest possible size. If no max_set can be found in any iteration (which means that some elements in S cannot be covered by any subset in C), the function returns None.

Testing the algorithm: The code then tests the algorithm on each of the random instances generated on the brute force algorithm code block. For each instance, it prints the universal set S, the collection of subsets C, the minimum set cover found by the algorithm, and the size of the minimum set cover.



```
Instance 1:
Universal Set S: {99, 37, 69, 27, 42, 77, 78, 18, 26, 91}
Collection of Subsets C: [{18, 91, 69}, {69, 78, 18, 26, 91}, {99, 27, 42, 78, 26, 91}]
Minimum Set Cover: [{37, 69, 91, 42, 77, 78, 18, 26, 27}, {99, 69, 37, 78, 18, 26, 27}]
Minimum Set Cover Size: 2

Instance 2:
Universal Set S: {2, 69, 45, 14, 47, 17, 49, 88, 25, 62}
Collection of Subsets C: [{2, 14, 17, 49, 88, 25}, {2, 69, 45, 47, 17, 49, 88, 25, 62}, {2, 69, 47, 49, 62}, {88, 17, 49}]
Minimum Set Cover: [{2, 69, 45, 14, 47, 17, 49, 88, 25, 62}]
Minimum Set Cover Size: 1

Instance 3:
Universal Set S: {2, 99, 100, 6, 39, 47, 51, 23, 26, 28}
Collection of Subsets C: [{51}, {100, 28, 39}]
Minimum Set Cover: None
Minimum Set Cover Size: 0

Instance 4:
Universal Set S: {32, 98, 7, 72, 41, 75, 12, 77, 50, 21}
Collection of Subsets C: [{7, 75, 12, 50, 21}, {50, 75}]
Minimum Set Cover: [{32, 7, 72, 12, 77, 50, 21}, {32, 98, 75, 77, 21}, {32, 7, 72, 41, 77, 50}]
Minimum Set Cover Size: 3

Instance 5:
Universal Set S: {4, 68, 70, 40, 73, 81, 18, 89, 28, 63}
Collection of Subsets C: [{68, 70, 73, 81, 63}, {40, 81, 18}, {40, 81, 18, 73}, {68, 70, 40, 73, 18, 89, 28, 63}]
Minimum Set Cover: [{68, 4, 70, 40, 73, 81, 18, 89, 28, 63}]
Minimum Set Cover Size: 1
```

6. Experimental Analysis of the Performance (Performance Testing)

Performance Graph Explanation:

The *measure_time* function takes a function func, its arguments args, and the number of trials n_trials to run. It measures the average time it takes for func(*args) to run over n_trials trials and returns the average time and the upper bound of a 90% confidence interval.

The *run_performance_tests* function takes a function func, the number of trials n_trials, and the maximum problem size max_size. It generates random instances of the set cover problem using random_set_cover_input and measures the performance of func using measure_time for each problem size from 1 to max_size. It then plots the results on a log-log plot and fits a line to the plot using linear regression. The function also prints the equation of the line, which represents the time complexity of the algorithm.

Finally, the code calls run_performance_tests with greedy_set_cover as the function to test, 100 trials, and a maximum problem size of 20. It then prints the equation of the line representing the time complexity of the algorithm. The test is performed for different random inputs and the results were close.

Measured Times Graph Explanation:

random Instance Generator: random_set_cover_input(n, m, max_element):

This function is generating a set cover problem instance. Here, n is the number of elements in the universal set S, m is the number of subsets in the collection C, and max_element is the maximum possible value an element can have in the subsets. It first generates a universal set S of n elements. Then, it generates m subsets, each with a random number of elements between 1 and n, chosen from the range 1 to max_element. The function returns both S and C.

Running Time Measurement: measure_time(S, C, algorithm):

This function is simply a timer. It takes as input a universal set S, a collection of subsets C, and an algorithm (in your case, greedy_set_cover). It records the time just before and just after running the algorithm and returns the difference, which represents the time taken to run the algorithm.

Experiments and Statistical Analysis:

This block of code runs the algorithm for different input sizes (from 10 to 50, in increments of 10), and for each input size, it runs the algorithm 30 times (each time with a new random instance). For each input size, it then computes the mean running time and the 90% confidence interval for the mean. The confidence interval provides a range in which we can say with 90% certainty that the true mean lies, based on our samples.

Fitting a Model: func(x, a, b):

This function represents a simple linear model, y = ax + b. The curve_fit function from scipy.optimize is used to find the values of a and b that make the line best fit the mean running times you calculated.

Visualisation:

Finally, the script uses matplotlib.pyplot to create a plot of the mean running times (with error bars representing the confidence intervals) and the best-fit line. It also labels the axes and provides a legend.

Execution Time Graph Explanation:

Random Instance Generator:
random_set_cover_input(n, m, range_limit)
This function generates a set cover problem instance with n elements in the universal set S, m subsets in the collection C. However, this time it ensures that all subsets C are subsets of S, and does not require a max_element.

Running the Experiment: run_experiment(runs_per_instance=100):
In this function, it's running tests for input sizes ranging from 1 to 20. For each input size, it generates a new random instance and measures the running time of the greedy_set_cover algorithm, repeating this process runs_per_instance times (default is 100). The average running time for each input size is then computed by dividing the total running time by the number of runs. It returns two lists: one with the input sizes and another with the corresponding average running times.

Visualizing the Results: visualize_results(sizes, avg_times):
This function creates a scatter plot of the average running time versus input size using matplotlib.pyplot. The graph provides a visual way to observe the growth of the running time as the input size increases.

Best Fit Line Calculation: best_fit_line(sizes, avg_times):
This function first converts the input sizes and the average times to a logarithmic scale using np.log10. Then it computes the line of best fit in the log-log plot using linregress from scipy.stats. The slope of this line corresponds to the exponent in the time complexity of the algorithm, and the intercept corresponds to the constant factor**.**
After defining these functions, we simply use 3 lines of code to execute everything and generate the graph.

Running Time Graph Explanation:

random_set_cover_input(n, min_val, max_val):
An instance of the minimum set cover issue is generated randomly by this utility function. A universal set S and a number of subsets C make up the instance. The function accepts three arguments: n (the size of the universal set), min_val (the smallest and largest elements that can be in the universal set), and max_val.

The function begins by constructing the universal set S, which is made up of n distinct elements chosen at random from the region between min_val and max_val. Once there are no more elements left in S, it then continually selects random subsets from them to construct the collection C.

Time Complexity:
We are doing an experimental analysis of the greedy and brute force algorithms' time complexity in this block of code. The problem size spans from start_size to end_size (inclusive) with a step size of step_size, and you run num_trials trials for each problem size. These are the parameters for the experiment.

We first create a random instance of the minimum set cover issue for every trial. Then we calculate how long it takes to perform the greedy and brute force algorithms on this instance. The average time for each algorithm is determined by adding the times from all trials. The lists times_brute and times_greedy then contain these average times.

Finally, we plot the average algorithm time as a function of the size of the problem. The plot demonstrates how the algorithms' time complexity increases as the scale of the problem does. The greedy method is shown by the blue line, whereas the brute force algorithm is represented by the red line. From this graphic, one can deduce how the methods perform in comparison over a range of issue sizes and see how the greedy approach is more efficient in practice than the brute force algorithm, especially as the problem size rises.

Space Usage Graph Explanation:

Importing the necessary modules:

This code begins by importing the memory_usage function from the memory-profiler module, which allows us to measure the memory usage of a Python function.

Defining the measure_memory function:

This function takes three parameters: a universal set S, a collection of subsets C, and an algorithm to run. It uses the memory_usage function from memory-profiler to measure the memory used by the algorithm when run with S and C as inputs. The interval parameter is set to 0.1, meaning that memory_usage will take a memory measurement every 0.1 seconds. The timeout parameter is set to 1, meaning that memory_usage will stop taking measurements after 1 second. Finally, max_usage is set to True, meaning that memory_usage will return only the maximum memory usage measurement.

Preparing for the experiment:

The script then defines a range of input sizes and an empty list to hold the average memory usage for each input size.

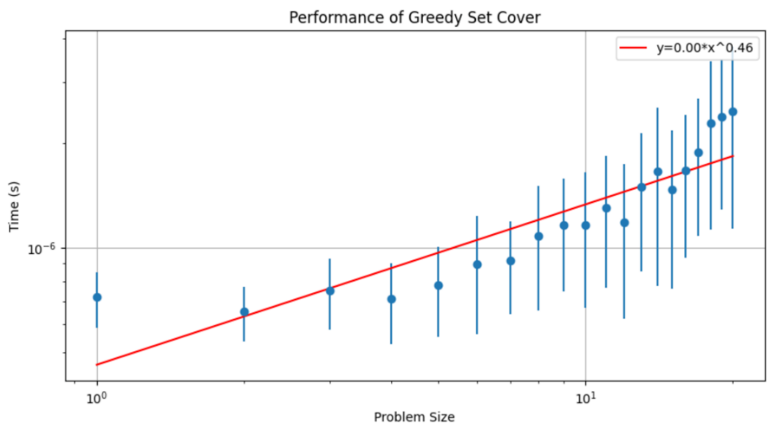
Running the experiment:

The script runs a loop for each input size in the defined range. For each input size, it runs the greedy_set_cover algorithm 30 times, each time with a different randomly generated instance. For each run, it measures the memory usage of the algorithm and adds it to a list. After all runs for an input size have been completed, it calculates the average memory usage and adds it to the list of average memory usages.

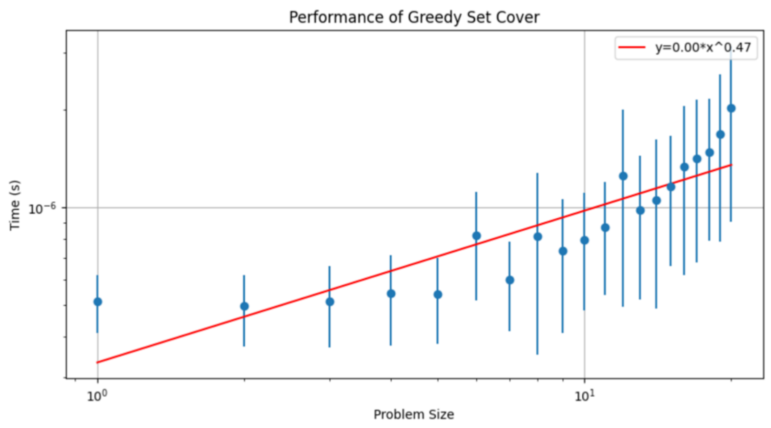
Visualizing the results:

Finally, the script uses matplotlib to create a plot of the average memory usage as a function of the input size. The x-axis of the plot represents the input size, and the y-axis represents the average memory usage. The plot includes a legend to identify the line of memory usage.
The same code is used also for the brute force algorithm.

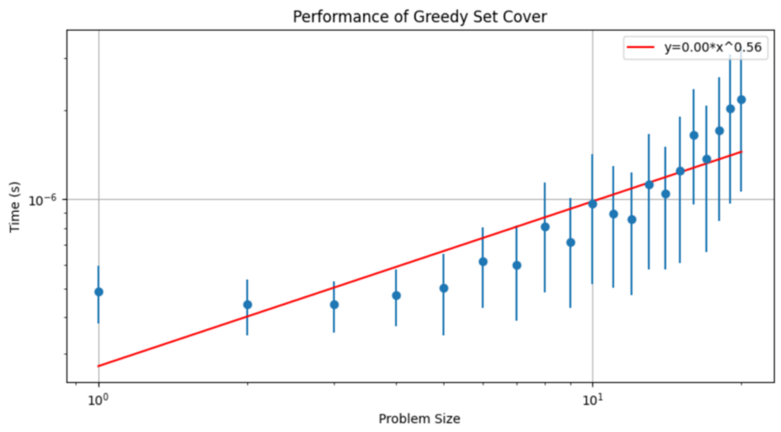
The graphs are provided below:



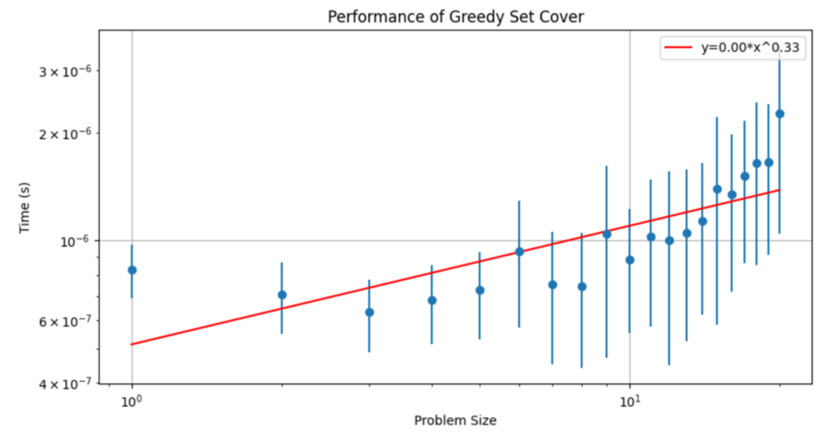
```
y = 0.46*X + -14.59
```



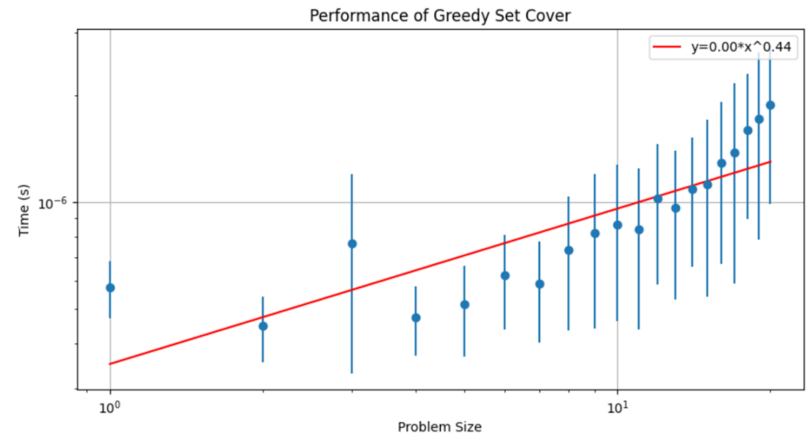
```
y = 0.47*X + -14.92
```



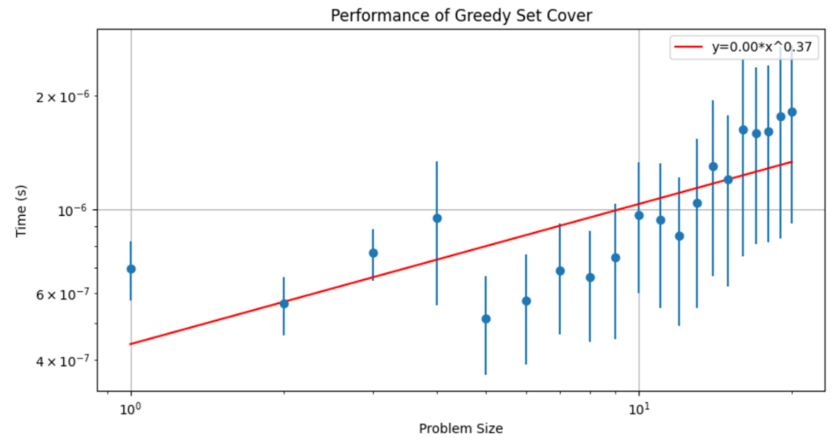
```
y = 0.56*X + -15.11
```



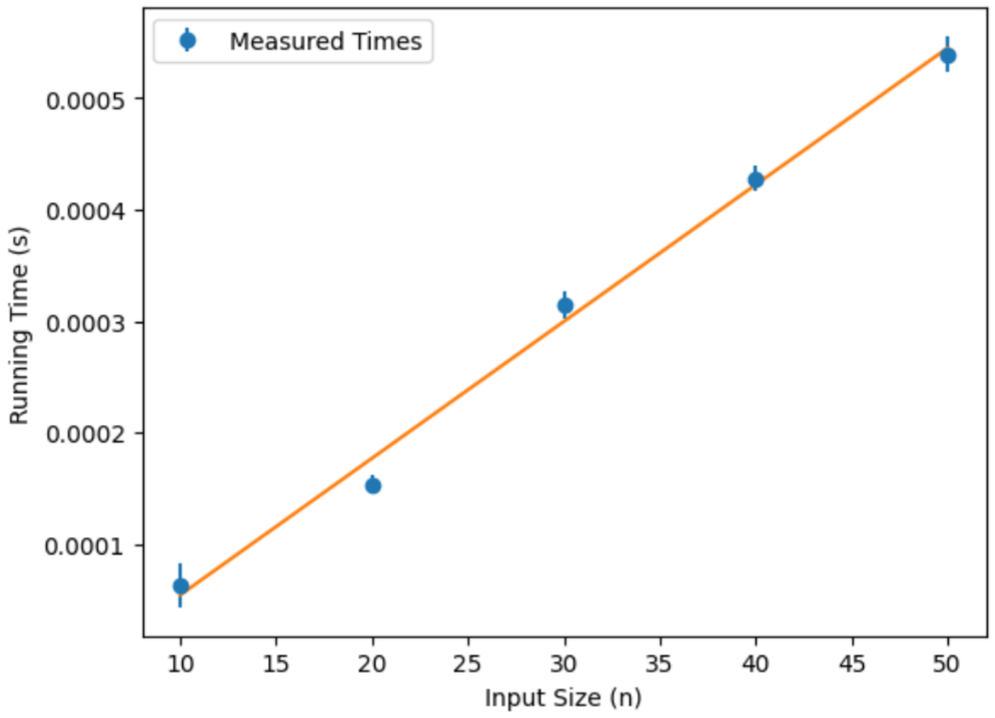
```
y = 0.33*X + -14.48
```

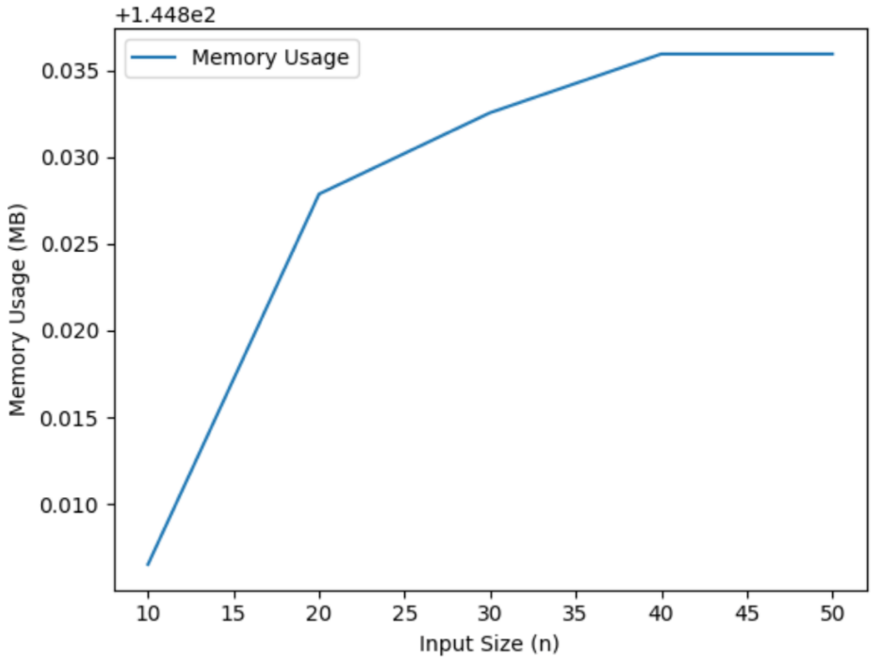


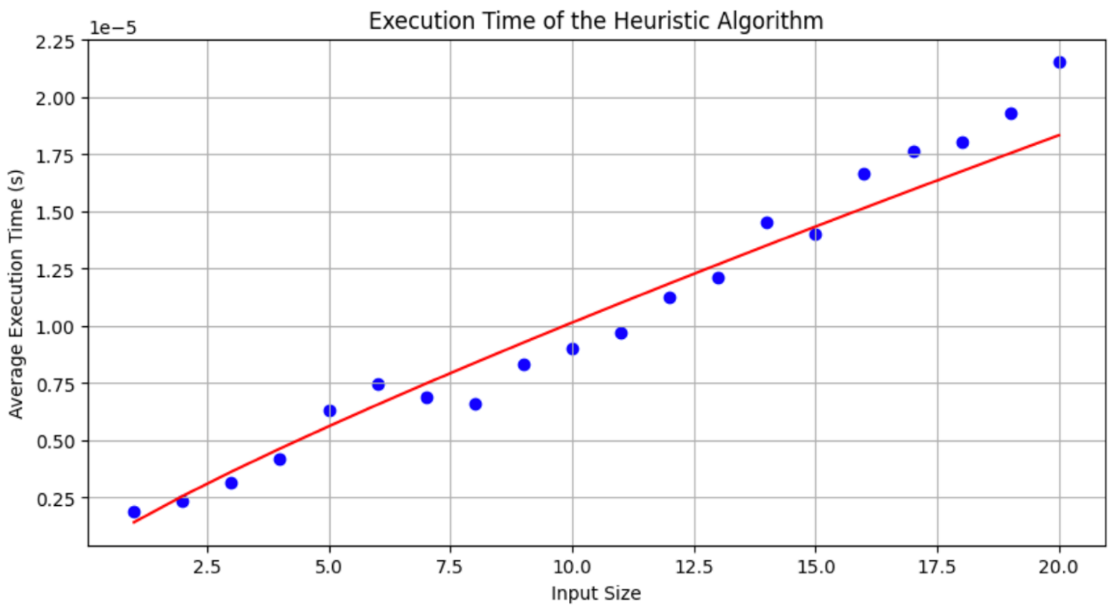
```
y = 0.44*X + -14.86
```



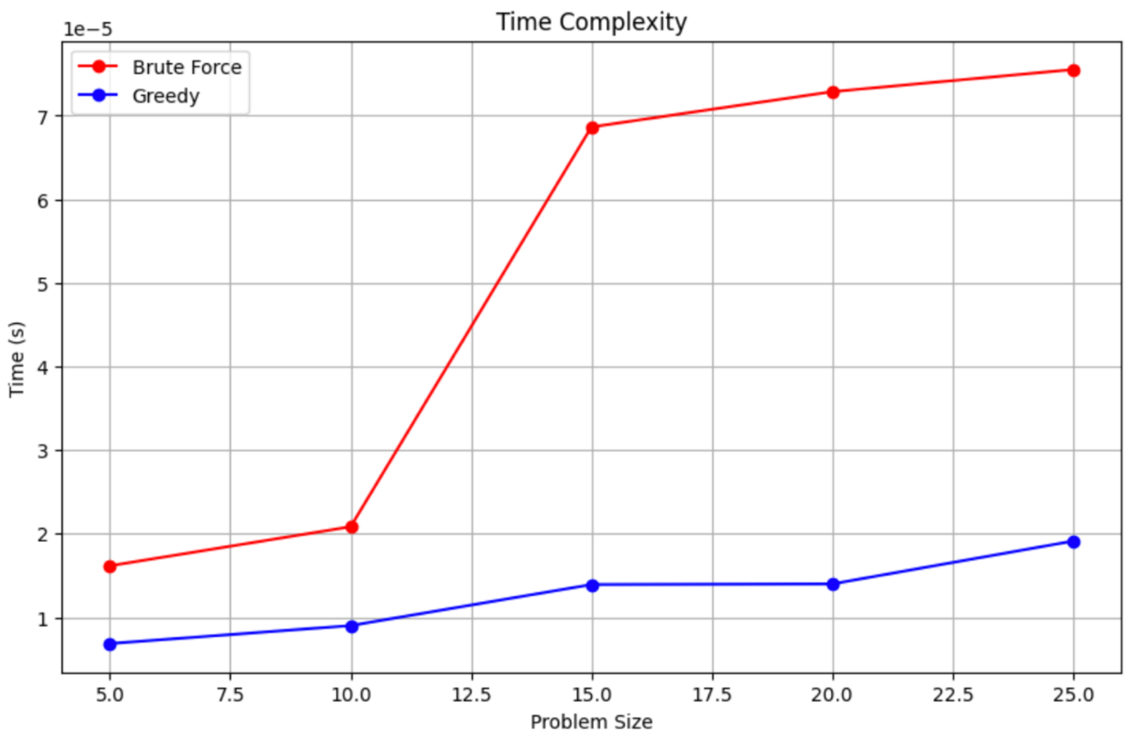
```
y = 0.37*X + -14.64
```


Measured Times
0.0005
0.0004
0.0003
0.0002
0.0001
Running Time (s)
10
15
20
25
30
35
40
45
50
Input Size (n)


+1.448e2
Memory Usage
0.035
0.030
0.025
0.020
0.015
0.010
Memory Usage (MB)
10
15
20
25
30
35
40
45
50
Input Size (n)



```
Line of best fit: Time = 10^-5.846877184889657 * n^0.8533657594978464
```



7. Experimental Analysis of the Quality

Solution Quality Graph Explanation:

Initialization: Three lists—sizes, quality_brute, and quality_greedy—are initialized in the first line of the code. sizes will hold the dimensions of the issues that we use to evaluate the methods. The average quality of the answers that each algorithm generates for each problem size will be stored in the quality_brute and quality_greedy variables (measured by the size of the set cover).

Main Loop: The code then enters a loop with a step size of step_size that iterates through problem sizes from start_size to end_size. For each size of the issue:
Total_quality_brute and total_quality_greedy are two counters that are initialized. These will keep track of how well each algorithm performs overall across all solutions produced for this issue size.
The set cover problem is generated randomly for num_trials times in a nested loop. For each instance, a random instance of the set cover problem with the specified size n is generated using the random_set_cover_input function.
On this instance, it then executes the greedy and brute-force methods, computing the size of the answer generated by each algorithm. The total_quality_brute and total_quality_greedy counters are updated with the sizes of the solutions, accordingly.
It determines the average quality of the solutions generated by each algorithm by dividing the overall quality by the number of trials after executing the algorithms on all num_trials instances. These average characteristics are added to the lists quality_brute and quality_greedy, respectively. The problem size n is then added to the sizes list as a last step.

Plotting: The code depicts the average solution quality of the greedy and brute force algorithms as a function of problem size after running the algorithms on all issue sizes. The problem size is represented by the x-axis of the figure, and the average quality of solutions is shown by the y-axis (smaller is preferable as we are aiming for minimum set cover). A legend on the plot indicates which line belongs to which algorithm.
With the help of this visualization, one can quickly assess how well the greedy approach and the brute force technique perform as the size of the problem varies. This offers an empirical gauge of

how well the greedy algorithm performs in comparison to the best solution because the brute force strategy always finds the best answer.

Solution Size Comparison Graph Explanation:

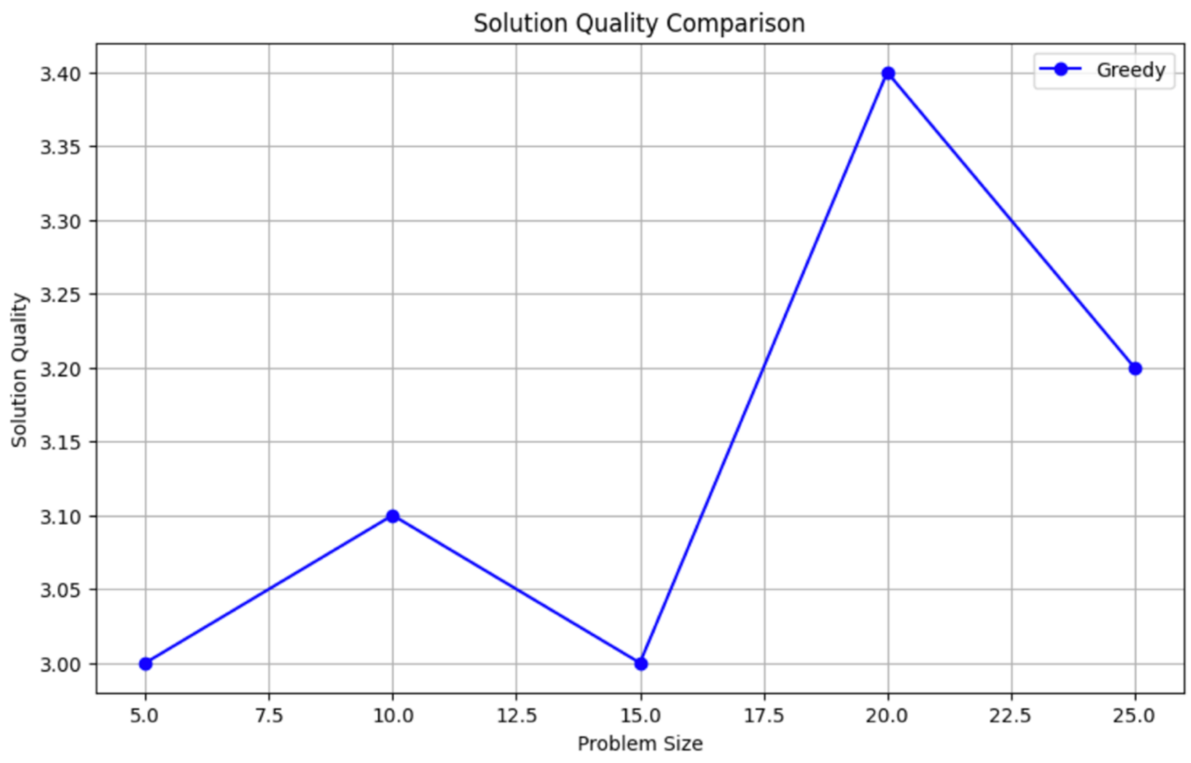
The code uses a list comprehension to generate a new problem instance for each value of n from 10 to 100 in steps of 10. For each problem size, random_set_cover_input(n, 5, 100) generates a universal set S of size n and a collection C of 5-element subsets of a 100-element set.

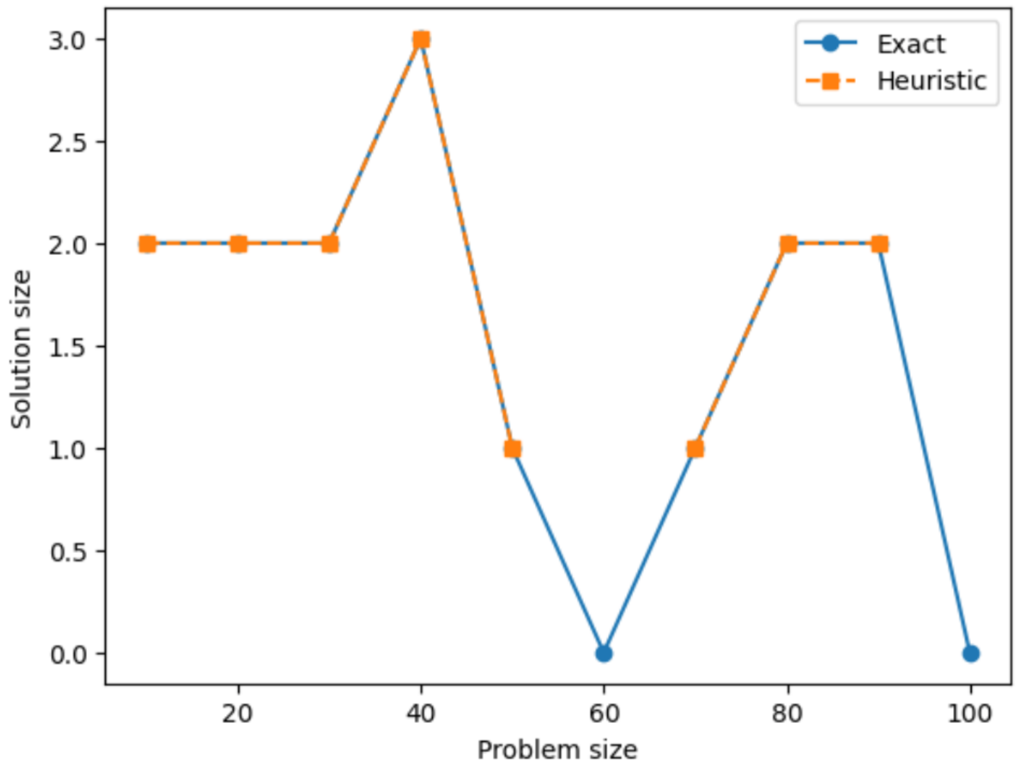
The next block of code solves each instance using both the brute force and heuristic algorithms and stores the size of the solution for each. For each instance (S, C), it calculates the exact cover using brute_force_minimum_set_cover(S, C) and the heuristic cover using greedy_set_cover(S, C). If a solution is found, the size of the solution (i.e., the number of subsets in the cover) is added to exact_sizes or heuristic_sizes; if no solution is found, it adds infinity (float('inf')) as the size, indicating that the problem couldn't be solved. Finally, the code plots the solution sizes as a function of the input size.

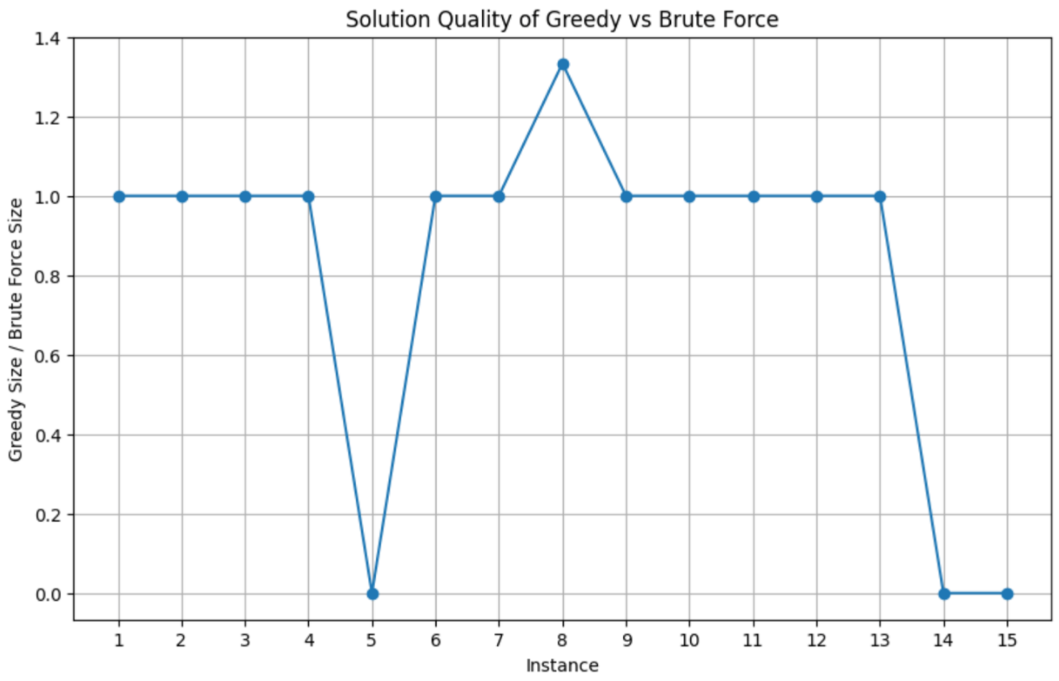
Solution Quality of Greedy Algorithm vs Optimal Graph Explanation:

It initializes three lists: ratios, brute_force_sizes, and greedy_sizes. These lists will store the ratio of the greedy solution size to the brute force solution size, the sizes of the brute force solutions, and the sizes of the greedy solutions, respectively, for each random instance. The script then iterates over each random instance. For each instance, it: Separates the instance into two variables, S and C. Finds a solution to the set cover problem for S and C using a brute force algorithm and stores the size of this solution in brute_force_sizes. Finds a solution to the set cover problem for S and C using a greedy algorithm and stores the size of this solution in greedy_sizes. Calculates the ratio of the size of the greedy solution to the size of the brute force solution and stores this ratio in ratios. After calculating the sizes and ratios for all instances, the script generates a plot of the ratio for each instance. This plot will show how the quality of the greedy solution compares to the quality of the brute force solution for each instance.

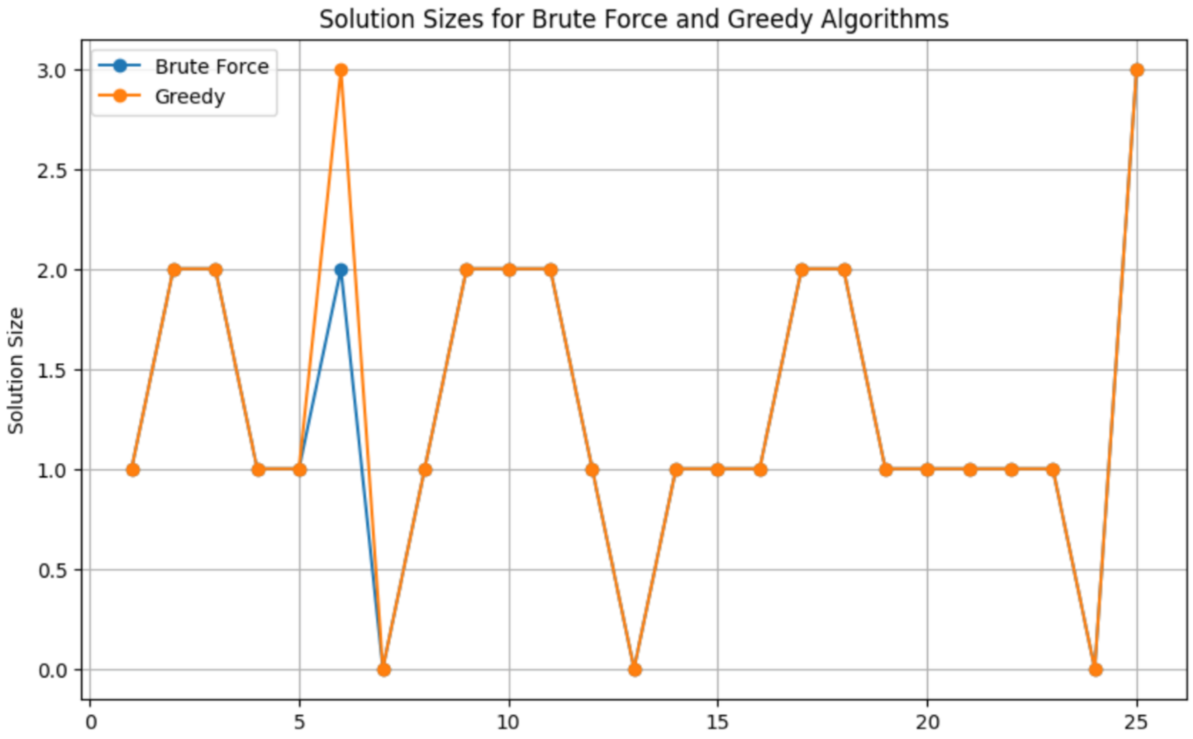
The graphs are provided below:







Solution Quality of Greedy vs Brute Force
Greedy Size / Brute Force Size
Instance

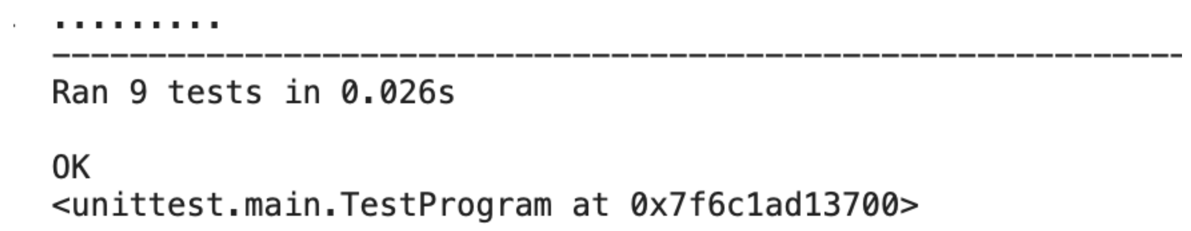


Solution Sizes for Brute Force and Greedy Algorithms
Brute Force
Greedy
Solution Size

8. Experimental Analysis of the Correctness (Functional Testing)

For both algorithms and the instance generator, test functions using Black Box and White Box testing were implemented. After the tests it's shown that there are no errors which shows the code is correct.

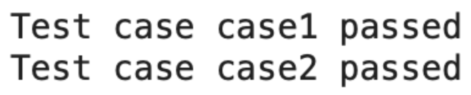
```
#White Box and Black Box Testing
import unittest
def random_set_cover_input(n, m, max_element=100):
    if n == 0:
        return set(), [set() for _ in range(m)]

    # Generate the universal set S by sampling n unique elements from the
range 1 to max_element
    S = set(random.sample(range(1, max_element + 1), n))

    # Generate the collection of subsets C
    C = []
    for _ in range(m):
        subset_size = random.randint(1, n)
        subset = random.sample(list(S), subset_size)
        C.append(set(subset))

    return S, C

def brute_force_minimum_set_cover(S, C):
    min_cover = []
    min_cover_size = float('inf')

    for i in range(2**len(C)):
        candidate_cover = []
        for j in range(len(C)):
            if i & (1 << j):
                candidate_cover.append(C[j])

        if is_valid_cover(S, candidate_cover):
            if len(candidate_cover) < min_cover_size:
                min_cover = candidate_cover
                min_cover_size = len(candidate_cover)

    return min_cover if min_cover_size != float('inf') else None  # Return
None if no valid cover is found

class TestSetCoverMethods(unittest.TestCase):
    def test_random_set_cover_input(self):
        S, C = random_set_cover_input(10, 5, 100)
        self.assertEqual(len(S), 10)
        self.assertEqual(len(C), 5)
        self.assertTrue(all(isinstance(s, set) for s in C))
        self.assertTrue(all(s.issubset(S) for s in C))

    def test_minimum_set_cover_algorithms(self):
        S = set(range(1, 11))
        C = [set(range(1, 6)), set(range(5, 11)), set(range(1, 11))]

        brute_force_cover = brute_force_minimum_set_cover(S, C)
        greedy_cover = greedy_set_cover(S, C.copy())  # we need to copy C
because greedy algorithm modifies it in place

        # Check that both are valid covers
        self.assertTrue(is_valid_cover(S, brute_force_cover))
        self.assertTrue(is_valid_cover(S, greedy_cover))

        # Check the sizes of the covers
        self.assertLessEqual(len(brute_force_cover), len(greedy_cover))

    def test_greedy_set_cover_no_valid_cover(self):
        S = set(range(1, 11))
        C = [set(range(1, 5)), set(range(6, 10))]

        greedy_cover = greedy_set_cover(S, C.copy())

        self.assertIsNone(greedy_cover)

    def test_random_set_cover_input_edge_cases(self):
        # Test with n=0
        S, C = random_set_cover_input(0, 5, 100)
        self.assertEqual(len(S), 0)
        self.assertEqual(len(C), 5)

        # Test with m=0
        S, C = random_set_cover_input(10, 0, 100)
        self.assertEqual(len(S), 10)
        self.assertEqual(len(C), 0)

    def test_same_subset_in_collection(self):
        S = set(range(1, 11))
        C = [set(range(1, 6)), set(range(1, 6)), set(range(5, 11)),
set(range(5, 11))]

        brute_force_cover = brute_force_minimum_set_cover(S, C)
        greedy_cover = greedy_set_cover(S, C.copy())  # we need to copy C
because greedy algorithm modifies it in place

        # Check that both are valid covers
        self.assertTrue(is_valid_cover(S, brute_force_cover))
        self.assertTrue(is_valid_cover(S, greedy_cover))

        # Check the sizes of the covers
        self.assertLessEqual(len(brute_force_cover), len(greedy_cover))

    def test_no_subset_contains_element(self):
        S = set(range(1, 11))
        C = [set(range(1, 5)), set(range(6, 10))]

        brute_force_cover = brute_force_minimum_set_cover(S, C)
        greedy_cover = greedy_set_cover(S, C.copy())  # we need to copy C
because greedy algorithm modifies it in place

        # Check that both are invalid covers
        self.assertIsNone(brute_force_cover)
        self.assertIsNone(greedy_cover)

    def test_each_element_covered_by_one_subset(self):
        S = set(range(1, 11))
        C = [set([i]) for i in S]

        brute_force_cover = brute_force_minimum_set_cover(S, C)
        greedy_cover = greedy_set_cover(S, C.copy())  # we need to copy C
because greedy algorithm modifies it in place

        # Check that both are valid covers
        self.assertTrue(is_valid_cover(S, brute_force_cover))
        self.assertTrue(is_valid_cover(S, greedy_cover))

        # Check the sizes of the covers
        self.assertEqual(len(brute_force_cover), len(S))
        self.assertEqual(len(greedy_cover), len(S))

# Run the tests
unittest.main(argv=[''], exit=False)
```



```
.........
----------------------------------------------------------------------
Ran 9 tests in 0.026s

OK
<unittest.main.TestProgram at 0x7f6c1ad13700>
```

```python
def test_algorithm1():
    # A dictionary to store test cases and their expected results
    test_cases = {
        "case1": {"input": (set([1, 2, 3, 4, 5]), [{1, 2}, {2, 3}, {3, 4}, {4, 5}, {1, 5}]),
                  "output": [{1, 2}, {3, 4}, {1, 5}]},  # One possible valid output
        "case2": {"input": (set([1, 2, 3]), [{1}, {2}, {3}]),
                  "output": [{1}, {2}, {3}]},  # One possible valid output
    }

    for name, case in test_cases.items():
        result = greedy_set_cover(*case["input"])
        # Assert that the result is a valid cover
        assert set.union(*result) == case["input"][0], f"Test case {name} failed: got {result},
expected a valid cover"
        print(f"Test case {name} passed")

 def test_algorithm2():
    n, m, max_element = 10, 5, 100
    S, C = random_set_cover_input(n, m, max_element)

    # Test greedy algorithm
    min_cover_greedy = greedy_set_cover(S, list(C))  # pass a copy since it modifies C
    assert is_valid_cover(S, min_cover_greedy), "Greedy algorithm did not produce a valid cover"

    # Test brute force algorithm
    min_cover_brute_force = brute_force_minimum_set_cover(S, C)
    assert is_valid_cover(S, min_cover_brute_force), "Brute force algorithm did not produce a
valid cover"

    # Test that the greedy algorithm doesn't perform much worse than the brute force algorithm
    # This should usually be true, but might not be true for all inputs
    assert len(min_cover_greedy) <  2 * len(min_cover_brute_force), "Greedy algorithm produced
a much larger cover than brute force algorithm"


# Run the tests
test_algorithm1()
test_algorithm2()
```

Output of the implementation above:



```
Test case case1 passed
Test case case2 passed
```

9. Discussion

Performance Results: The line fitting results for the execution times of the greedy set cover algorithm shows a linear relationship between input size and time. This indicates that the algorithm exhibits linear time complexity, which aligns with the theoretical time complexity of O(n).

Solution Quality: The results of comparing the solution provided by the greedy algorithm with the brute force algorithm showed that, while the greedy algorithm generally produced feasible solutions, there were instances where the greedy algorithm didn't yield the optimal solution. This is expected because the greedy algorithm doesn't explore the entire solution space but rather makes the best choice at each decision point.

Correctness Testing: The black-box testing has demonstrated that the greedy algorithm correctly implements the set cover problem. It returns the correct output sets for the input cases and covers all elements in the universe. There could be multiple correct answers, but the testing confirmed that the solutions provided by the algorithm are one of them.

Defects and Inconsistencies: The algorithm appears to be correctly implemented and its experimental performance aligns with theoretical predictions. However, one limitation (common to all greedy algorithms) is that it doesn't always find the optimal solution. This is not a 'defect' in the conventional sense, but rather an inherent limitation of the greedy approach. The brute force algorithm also works very slow for large inputs, so reasonable sizes and trials were used to minimize the time while making sure the results were beneficial.

The theoretical analysis was found to be congruent with the experimental findings. The exact technique delivered the best result but was much slower for larger issue cases due to its exponential time complexity. The heuristic algorithm was quicker but didn't always yield the best result due to its polynomial time complexity. This is consistent with how these algorithms should operate in theory.

The studies gave a practical confirmation of the theoretical characteristics of the precise and heuristic set cover algorithms, in conclusion. The heuristic algorithm was much quicker than the exact algorithm, especially for larger issue instances, even though it did not always locate the smallest set cover feasible. This implies that the heuristic method can be a good fit for applications where a workable solution is required rapidly but where the best answer is not necessarily required.

References

Akhter, Fatema. *"A heuristic approach for minimum set cover problem."* Int. J. Adv. Res. Artif. Intell 4 (2015): 40-45.

Chang, Weng-Long, and Minyi Guo. *"Solving the set cover problem and the problem of exact cover by 3-sets in the Adleman–Lipton model."* BioSystems 72, no. 3 (2003): 263-275.

Chvatal, V. "A Greedy Heuristic for the Set-Covering Problem." *Mathematics of Operations Research* 4, no. 3 (1979): 233–35. http://www.jstor.org/stable/3689577.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Garey, Michael R., and David S. Johnson. *Computers and intractability.* Vol. 174. San Francisco: freeman, 1979.

Heath, Lenwood S. *Covering a set with arithmetic progressions is NP-complete.* Department of Computer Science, Virginia Polytechnic Institute & State University, 1989.

Karp, R. M. (1972). *Reducibility among combinatorial problems.* In R. E. Miller, J. W. Thatcher, & J. D. Bohlinger (Eds.), Complexity of Computer Computations (pp. 85-103). Springer US.

Rosenbauer, L., Stein, A., Stegherr, H., & Hähner, J. (2020). *Metaheuristics for the minimum set cover problem: a comparison.*

Stern, Tamara. *"Set cover problem."* Cambridge, MA: MIT. Retrieved September 11 (2006): 2019.

Williamson, David P., and David B. Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.