Introduction:

The set covering problem is an optimization problem that aims to find minimum cost subset of sets that covers all elements in a given universe. It is a NP-hard problem, and hence heuristic techniques are applied to find good-quality solutions efficiently.

This report presents a python program that attempts to solve the set covering problem using a combination of constructive and improvement heuristics. Greedy algorithm is used for constructing the initial solution, while local search and tabu search algorithms are used for exploring the solution space to obtain a better solution.

Datasets

The performance of a Python program was evaluated on 5 data files hosted of Set covering problem instances hosted on OR Library. The format of data in these problem instances format is as follows:

- number of rows/elements (m), number of columns/sets (n)
- the cost of each column/set c(j) such that j=0,...,n
- for each row/elements i (i=0,...,m): the number of columns which cover row I, followed by a list of the columns/sets which cover row/element i

The count of columns/sets in the dataset starts from 0. For maintaining the format, the solution provided by the program also follow the same format for referencing the column/sets i.e. Set 0 implies first set in the data.

Summary of Work

The python program is designed to accept name of the problems instance from user and build the associated cost array and problem matrix. As a safety measure to highlight discrepancy in data, the user is notified if there is a difference between the number of columns covering the row specified in the instance and the length of the list of columns that covers the row in instance.

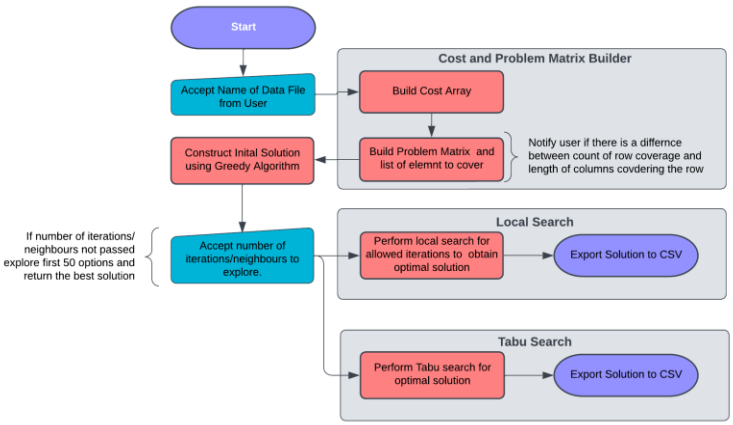

Fig. 1: Flowchart of the Set covering solution program

In certain cases, it may happen that an element is not covered by any set at all. In such scenarios, program notifies the user about the same and finds the best solution that covers all the remaining element, allowing to find solution where a element or set of elements is not covered by any set. Lastly, the program returns all the neighbouring solution of the solution obtained using constructive heuristics that yield the same lowest cost in the local search.

1. Constructive Heuristic

The implemented constructive algorithm starts with an empty solution set and repeatedly adds sets with smallest cost ratio to solution set until all the elements to be covered are covered by the solution subset. In each iteration, the algorithm selects the set with the smallest cost ratio (ratio of the cost of set and the number of uncovered elements covered by the set). Essentially, the algorithm chooses the set that covers the most uncovered elements at the least cost. The elements in the chosen set are removed from the list of elements to be covered, and the algorithm continues until no elements are left to be covered by the solution subset. This algorithm is an efficient way to find an initial solution to the set covering problem and can be improved further using local search and other techniques.

2. Local Search

The implemented local search algorithm uses the k-opt operator in conjunction with the greedy algorithm as operator to generate the neighbourhood of the solution set obtained from the constructive heuristics. Each of these neighbourhood solutions are evaluated for their cost, and the current solution is replaced with a neighbouring set if its cost is lower. If a better solution is found in the neighbourhood, the algorithm updates the neighbourhood of the solution to that of the best solution found so far and explores this neighbourhood to search for a better solution. By iteratively applying this process, the algorithm explores the neighbourhood and obtains the optimal solution that covers all elements with the minimum cost for the subset of sets. One of the main drawbacks of this algorithms is that it can get stuck in local optima.

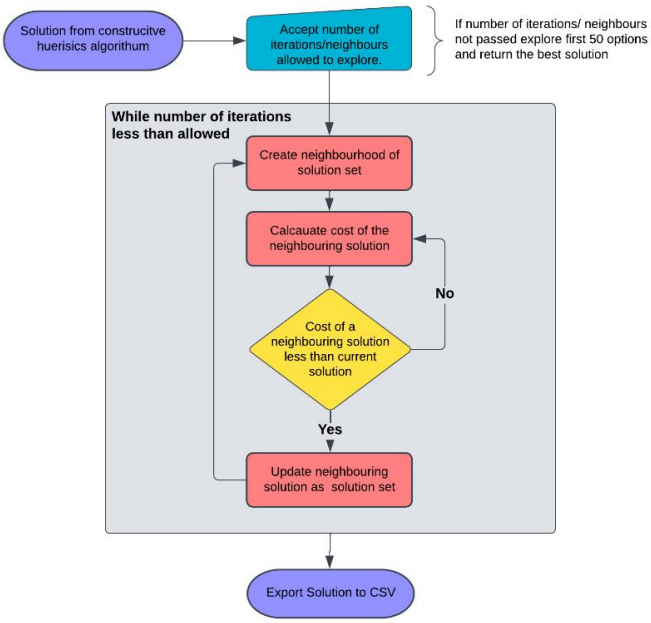

Fig. 2: Flowchart of the Local Search Algorithm

3. Tabu Search

Tabu search is an extension of local search algorithm which overcomes the limitations of local search algorithms by introducing additional restrictions or tabus that guide the search towards better solutions while avoiding getting stuck in local optima. The tabu list in programme enforces the algorithm to explore diverse sets other than the original partial solution set used to create the neighbourhood. This forces algorithm to consider initially less favourable options and evaluate new solution sets for better

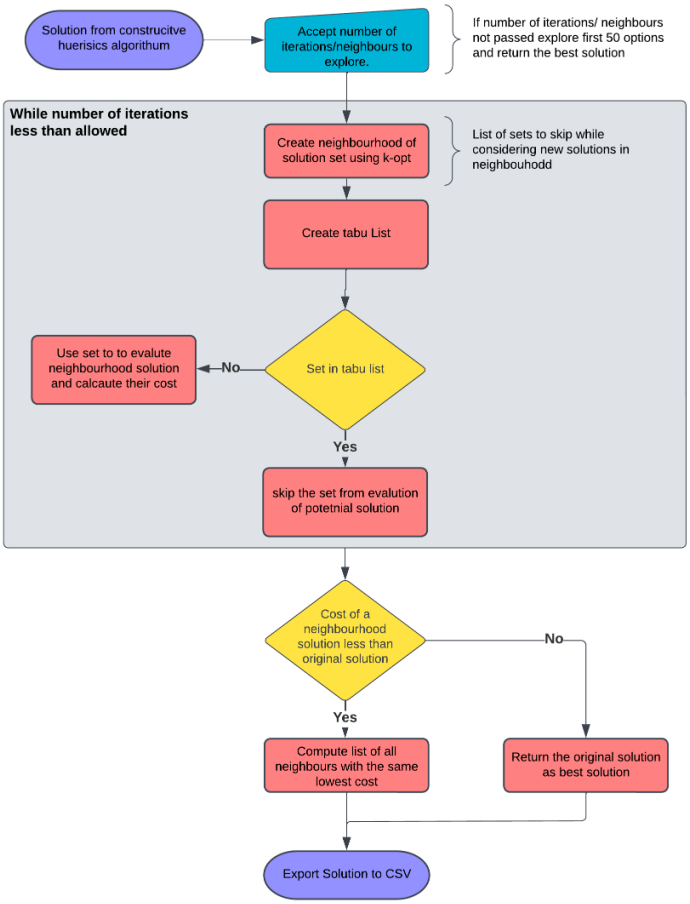

Fig. 3: Flowchart of the Tabu search Algorithm

Result and Analysis

Performance of a Python program was evaluated using 5 data files with varying size to test the performance on large datasets. As expected, due to the nature of the problem, the execution speed of the program increased exponentially with increase in size of the dataset. The execution time of the program can be reduced using multiprocessing/parallelization of code, but this was not explored in this project. The following table summarizes the cost of the subset of set from each of the algorithm for 5 data instances.

Data File	Cost of solution from Greedy Algo	Cost of solution from Local Search Algo.			Cost of solution from Tabu Search Algo.		
		(N=10)	(N=30)	(N=50)	(N=10)	(N=30)	(N=50)
SCP_01	540.0	536.0	532.0	532.0	532.0	532.0	486.0
SCP_02	263.0	263.0	263.0	263.0	263.0	263.0	263.0
SCP_03	320.0	305.0	305.0	305.0	310.0	307.0	267.0
SCP_04	468.0	467.0	360.0	308.0	415.0	415.0	411.0
SCP_05	693.0	692.0	687.0	687.0	690.0	690.0	686.0

Table 1: Cost of best subsets proposed from the various algorithm

It was also observed that the minimum cost from the local search and tabu search can dependent on the number of iterations/neighbouring sets explored. The algorithm produced better selection of sets with increasing number of iteration/ neighbouring sets explored, but this in turn also increases the execution time for program.

Conclusion

In conclusion, this project presents a program to estimate the optimal solution for a set covering problem and its performance on large data files with varying sizes. As expected, due to the nature of the problem, the execution time for the program increased exponentially with the increase in dataset size.

It is observed that the minimum cost for subset of sets from the local search and tabu search is dependent on the number of iterations or neighbouring sets explored. The algorithm produced better selection of sets with an increasing number of iteration or neighbouring sets explored, but this also increases the execution time of the program.

Analysis of the results of the local search methods on various smaller databases showed that algorithm achieved optimal solutions, leaving little to no room for improvement using tabu search. It is important to note that the small size of datasets considered may be a contributing factor to this, and tabu search may produce better results on larger datasets. However, achieving these improved results with tabu search may require a greater number of iterations to explore potential solutions, which can be computationally taxing and time-consuming. Finally, the parallelization of the code using multiprocessing libraries can reduce execution time but was not explored in this project.

Reference:

- Greedy approximate algorithm for set cover problem (2023) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/greedy-approximate-algorithm-for-set-cover-problem/
- Local search (optimization) (2022) Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Local_search_(optimization)
- Musliu, N. (1970) Local search algorithm for UNICOST set covering problem, SpringerLink. Springer Berlin Heidelberg. Available at: https://link.springer.com/chapter/10.1007/11779568_34
- Tabu search (2023) Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Tabu_search