Project 5: A class of greedy algorithms to solve the set cover problem

Contents

1 Overview

1.1 The set cover problem

The set cover problem (SCP) is a famous optimization problem wherein there are many elements contained in several sets (elements may be contained in more than one set), and the goal is to select the smallest number of sets so that every element is represented in the selected sets. The complete collection of elements is called a "universe". For example, in 1, there are 32 elements (dots) in the universe, and four sets (colored regions) containing the elements. Note that the sets do not have to cover contiguous elements; after all, Figure 1 is just one of infinitely many ways these same elements and sets could have been drawn. (We could have drawn all dots in a single row but that would not fit on the page.)

Frequently, we are not looking for the *smallest* number of sets, but the *cheapest* number of sets. That is, each set can have a certain cost (weight), and we want to the find the cheapest way to cover all the elements. This variation of SCP is called the *weighted SCP*, or wSCP. SCP and wSCP, as well as a multi-cover variation where each element has to be covered multiple times, have countless applications, some of which are shown in Table 1.

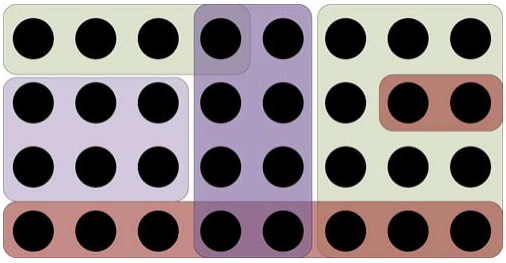

Figure 1: SCP illustration. Elements (dots) are covered by one or more sets (colored regions).

Table 1: Some applications of the set cover problem

Application	Elements	Sets
Airline crew scheduling [1]	Flights	Crews
Energy efficient sensor network design [4]	Coverage points	Sensor nodes
Computer virus detection [6]	Virus code	Strings defining each possible virus
Beam placement in radiation therapy [5]	Cancerous cells	Beams
Species differentiation [2]	Yeast pairs	Enzymes

1.2 SCP optimization model

We can easily describe wSCP as an optimization problem. Say we have n elements and m sets, and each set has cost c_i, $i = 1, \ldots, m$. We also have values a_{ij} that tell us whether or not set i covers element j:

$$a_{ij} = \begin{cases} 1 & \text{if set } i \text{ covers element } j \\ 0 & \text{otherwise} \end{cases} \qquad \forall i = 1, \ldots, m,\ j = 1, \ldots, n$$

Note that we can think of all the a_{ij} values as being a {0,1} matrix, where rows correspond to sets and columns correspond to elements. Now let's define decision variable x_i to indicate whether or not we select set i:

$$x_i = \begin{cases} 1 & \text{if set } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \qquad \forall i = 1, \ldots, m$$

Putting it all together, we get the following integer programming problem, which will give us the optimal solution to the weighted set cover problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} c_i x_i & & \text{(wSCP)} \\ \text{subject to} \quad & \sum_{i=1}^{m} a_{ij} x_i \geq 1 & \forall j = 1, \ldots, n & \\ & x_i \in \{0, 1\} & \forall i = 1, \ldots, m & \end{aligned}$$

If the model was for a regular SCP instead of wSCP, we would just replace all c_i values with 1. If the model was for a multi-cover problem, we would just replace the righthand side of the constraint with a user-provided value k to indicate that each element must be covered at least k times. For this project though, we will stick with the single-cover wSCP as it shown above.

2 A class of greedy algorithms to solve SCP

It is common for greedy optimization algorithms to all generally have the same steps, with the exception of the criteria used to select the next solution (in the case of SCP, the next set to add to the solution sets). You will use inheritance to build a generic greedy set cover solver class, and each algorithm will inherit from that generic parent class.

2.1 A greedy coverage heuristic

The simple greedy coverage algorithm for SCP is very straightforward. You start with an empty solution (meaning you haven't yet selected any sets to be in the final solution), and then add the set that will cover the most uncovered elements. The premise is that even though we ignore the costs of the sets, we will pick a very small number of sets in total, which should keep the total cost (the objective function value) low. We terminate when all elements are covered, or adding a further set cannot increase coverage.

Even with an emphasis on keeping the number of sets small, sometimes SCP solutions have too many sets to be practical. For example, in the application of SCP to select beams in radiation therapy treatments

[5], it was found that greedy algorithms would require up 50 beams to use in a treatment, even though only about 25 beams are reasonable from a clinical perspective. The fix was an adjustment to the algorithm to stop when 95% of all elements were covered. Although the resulting solution is technically infeasible to the original mathematical model wSCP (since not all elements were covered), it provided much more realistic and clinically acceptable solutions.

Therefore, in your algorithm, you should also allow the user to specify a percentage α of elements that must be covered, after which the algorithm can stop running. If the user wants a strict 100% covering, then the user can simply specify $\alpha = 100\%$ coverage.

2.2 A greedy cost heuristic

The simple greedy cost algorithm for SCP is just like the greedy coverage algorithm, with the exception that the next set selected is the one with the lowest cost that covers at least one uncovered element (i.e., we do not care how many uncovered elements it covers so long as it covers at least one). The premise is that even though we ignore the number of elements covered by the sets, we will pick only the cheapest sets that improve the solution, which should keep the total cost (the objective function value) low, even if we have to select many sets.

2.3 Chvátal's algorithm

Chvátal's algorithm [3] is possibly the most well-known and best-performing SCP algorithm, even though it is just another greedy algorithm and was invented over 30 years ago. Chvátal's algorithm is a greedy algorithm specifically designed for wSCP, and the only difference from the simple greedy algorithm is in the selection of the next set to add to the solution. Rather than add the set that covers the most elements, Chvátal's algorithm adds the set that has the smallest cost per uncovered element, called the cost-coverage ratio. For example, if set i has cost c_i and covers e_i elements *that have not already been covered by other selected sets*, then it has a cost-coverage ratio of c_i/e_i.

As with the other greedy algorithms, Chvátal's algorithm starts with an empty solution and stops when α percentage of elements have been covered or further improvement is not possible.

3 Program description

In this project, you will implement the greedy algorithms in Section 2 to solve the wSCP integer programming problem. You will compare the performance of the algorithms using the following three metrics:

- Final objective function value
- Feasibility of the final solution (element coverage)
- Computation time needed for the algorithm to run

As in previous projects, we provide you with a substantial amount of starter code including all input and output and all error messages. You are expected to read through the code and comments to understand what has been given, the function of each class, member, and method, and what is left to do. Places marked **TODO** in the code are places you should fill in your own code. You should not need to modify `Pro5_utorid` other than renaming the class with your actual utorid.

Before you dive into the code, it is critical that you run the solution jar with different sample inputs (three are provided with the project handout) and that you fully understand what is being printed out, how each algorithm is executing, and why each algorithm produces the result it does.

Menu options and error messages are included below for completeness, followed by helpful hints.

4 Menu options

- `M - Enter SCP model data`
  The user enters the number of elements, number of sets, costs of sets, and elements covered by each set.
- `P - Print SCP instance`
  The model instance is printed to the console.
- `A - Set minimum coverage percentage`
  The user enters a value for α, the minimum percentage of elements that must be covered.
- `C - Solve SCP with the greedy coverage heuristic`
  Run the greedy coverage heuristic on the problem.
- `S - Solve SCP with the greedy cost heuristic`
  Run the greedy cost heuristic on the problem.
- `V - Solve SCP with Chvatal's algorithm`
  Run Chvátal's algorithm on the problem.
- `X - Compare algorithm performance`
  Provide comparison on algorithm performance metrics and indicate which algorithms performed best in each metric.
- `Q - Quit`
  Quit the program.

5 Error messages

- `ERROR: Invalid menu choice!`
  when the user enters an invalid menu choice.
- `ERROR: No problem information has been loaded!`
  when the user attempts to solve the problem without loading in the problem information.
- `ERROR: No minimum coverage percentage has been specified!`
  when the user attempts to solve the problem without providing α.
- `ERROR: Cannot perform comparison because <method> has not been run for this problem instance!`
  when the user attempts to compare algorithm performances without having run one of the SCP solver methods.
- `ERROR: Input must be an integer in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of a `double`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of a `double`, then "`-infinity`" is displayed instead of the actual LB value.
- `ERROR: Input must be a real number in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of an `int`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of an `int`, then "`-infinity`" is displayed instead of the actual LB value.
- `WARNING: All sets added! Impossible to achieve <alpha>% coverage level.`
  when the algorithm terminates without reaching α percentage because all sets have been added to the solution.

6 Helpful Hints

- Take time to understand the provided starter code, sample runs of the solution jar on different inputs, and the control flow of the code required to produce this output (i.e., **note where polymorphism is occurring**). Considering writing `main` methods to test the functionality of classes. Make sure you understand the basic program loop interaction (i.e., `Pro5_utorid` which has been completely written for you) and then the structure of `SCPModel` and `ElementSet`. You will need to properly represent and print out an `SCPModel` instance before you can begin optimizing it with `GreedySolver`. If you find writing the heuristic solver subclasses hard, write a new subclass of `GreedySolver` called `ChooseFirstUnselectedSet` that does exactly what its name says. `GreedySolver` should find a solution (if it exists) with any selection method so this allows you to focus initial algorithm debugging on `GreedySolver`. Once `GreedySolver` is working then you can focus on the implementations of `GreedyCoverageSolver`, `GreedyCostSolver`, and `GreedySolver` as outlined in Section 2.
- Refresh yourself on the Collections class hierarchy (subclasses of Collection, SortedSet, Iterable, etc.), the methods available to each class, how to specify the sort ordering for sorted Collections, etc. Much of this material was covered in lecture, but Google will also provide excellent resources for augmenting the lecture content.
- As you are selecting sets and covering elements, make sure you remove the covered elements (i.e., by removing them `_elementsNotCovered`). Make sure you are not corrupting the original `SCPModel` when performing this update — copy Collections when necessary. Recall that you can copy a Collection by passing it as an argument to the constructor of the same class type.
- Remember to end the algorithm if all sets have been selected or no set can improve the coverage (i.e., `nextBestSet()` returns `null` in this case), even if you miss the minimum coverage percentage.
- To ensure that your solution matches the solution jar, you need to break ties in exactly the same way. This means that (a) you should always iterate in increasing order of set ID (this is why `ElementSet` has a comparator and `SCPModel` stores `ElementSet`'s in a `SortedSet` – `ElementSet`'s should be naturally sorted based on ID) and (b) if two sets have the same score for the greedy scoring function being evaluated, the set with lower ID should be chosen.
- **Don't worry if your computation times are not exactly the same as the solution times.** The computation time is not only dependent on the algorithm, but also on the coding style: efficiency of loops, efficiency of data structures, pre-processing, etc. Your code should not run substantially slower than the solution jar. A zip file of random SCP instances that can be copy/pasted into an input file is provided for testing.
- Your code will need more methods than the subset provided in the starter code.
- **Copying warning: there are innumerable ways to provide a correct solution to this project. We will carefully monitor code similarity of provided solutions and report all cases of unusual similarity to the department for disciplinary action.**

References

[1] E. Baker and M. Fisher. Computational results for very large air crew scheduling problems. *Omega*, 9(6):613–618, 1981.

[2] D. Buezas. Constraint-based modeling of minimum set covering: Application to species differentiation, 2010.

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

[4] S. Jenjaturong and C. Intanagonwiwat. A set cover-based density control algorithm for sensing coverage problems in wireless sensor networks. In *Cognitive Radio Oriented Wireless Networks and Communications, 2008. CrownCom 2008. 3rd International Conference on*, pages 1–6, May 2008.

[5] C. J. Lee, D. M. Aleman, and M. B. Sharpe. A set cover approach to fast beam orientation optimization in intensity modulated radiation therapy for total marrow irradiation. *Physics in Medicine and Biology*, 56(17):5679–95, 2011.

[6] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.