Comparison of solution approaches for the Set Cover Problem

Dominik Ramsauer, Matrikelnummer: 1737690

Lehrstuhl für Informationswissenschaft, Universität Regensburg

September 25, 2018

Abstract

The Set Cover Problem is a research issue in computer science with a lot of applications e.g. in machine learning, data mining and information retrieval. It is about selecting as few sets as possible from a collection that cover some universe. The aim of this project is to compare two approaches - a deterministic one and a statistical - and to rate them by concerns of coverage rate, solution size and computation time.

1 Introduction

The **Set Cover Problem** is a research issue in computer science, combinatorics and computational complexity theory which finds applications in machine learning, data mining/information retrieval[1], wireless sensor networks[2] and web host analysis[3] as also in learning theory[4] and operation research[5]. In simple terms, the issue of the problem is about selecting as few sets as possible from a collection that cover some universe[6, 7]. The Set Cover problem is known to be NP-hard and finding an optimal solution is NP-complete [8]. Richard M. Karp stated in 1972 21 NP-complete problems[9], where the Set Cover Problem is also mentioned. This means that there is no algorithm existing, which finds an optimal solution in polynomial time.

Therefore, the problem needs to be approximated properly. Even though algorithms based on a greedy heuristic are the most used to solve it, there are a bunch of other approaches discussing the Set Cover problem. For this project we want to consider two kinds of algorithms in particular: deterministic algorithms on one hand and statistical algorithms on the other hand. One for each of those two attempts is to be presented in this work - a kind of tuned greedy algorithm for the deterministic attempt as presented in [6] and the so called simulated-annealing algorithm [10, 11, 12, 13], which was inspired by a technique in metallurgy. The aim of this project is two compare these approaches and to rate them by concerns of coverage rate, solution size and computation time.

2 Technical Definitions and Algorithms

2.1 The Set Cover Problem

For this project we concentrate on the standard Set Cover problem, meaning that all sets unweighted [14]. We have given a collection $\mathscr{S} = \{S_1, S_2, ..., S_m\}$ of size $|\mathscr{S}| = m$, our sets-universe, containing a bunch of m text sets and a universe $X = \{x_1, x_2, ..., x_n\}$ of n elements, our words-universe. The aim is to get a sub-collection of $\mathscr{S}$ containing as few sets as possible that covers our words-universe X.

The most used algorithm for approximating the Set Cover problem is based on a greedy heuristic[6, 15, 11, 16]. This algorithm adds sets from the collection to the solution-collection measuring the size of the sets. The intention behind this procedure is, that large sets lead potentially to a high coverage. Therefore, we define a set Σ containing the indices of the sets that are added to our solution, and a set C containing the different elements that have been covered so far. A snippet of pseudo-code is shown on the next page.

Procedure:[6]
```
Initialization. $\Sigma = \emptyset$, $C = \emptyset$
Loop. while $C$ != $X$:
    1. Choose a set with $max(|S_i \backslash C|)$; let its index be $i^*$
    2. Add $i^*$ to $\Sigma$
    3. Let $C = C \cup S_i^*$
Return. $\Sigma$
```

So at each step of the greedy algorithm there has to be computed, which set in the collection has the most elements that have not already been covered. In more mathematical terms: the set $|Si \backslash C|$ after subtracting all already covered elements, that has the maximum length. This procedure requires many iterations through the data set, so that it has to be read very often, or some support technique has to be implemented such as an inverted index[6]. Another point that has to be mentioned is, that a kind of threshold has to be implemented when the greedy procedure may stop, especially when the words-universe X is not known in advance. If it is known, the greedy heuristic can be implemented in that way, that it checks at every step if all words are covered and stops at this point[11, 10].

The standard greedy heuristic does not accomplish to find the optimal solution in most cases, meaning that there are often shorter solutions. The approaches presented in the next sections are extensions of the greedy heuristic and had been chosen to improve it by concerns of speed and accuracy.

2.2 Disk-Friendly Greedy

As the name indicates, this algorithm published by Cormode, Karloff, and Wirth is based on the greedy algorithm presented before. As the greedy algorithm does need many random accesses on disk, the greedy algorithm performs badly on disk resident data, especially if data is very large[6]. To scale the greedy procedure to large data sets they introduce a new algorithm suited for these modern claims.

The idea behind their disk-friendly greedy (DFG) is to build sub-collections of the given sets-universe by the size of the sets. They introduce a parameter $p > 1$, which will control the set-size in the sub-collections and therefore acts as a running time and approximation factor.

Each set S_i will be attached to sub-collection $\mathcal{S}^{(k)}$ if:

$$p^k \leq |S_i| < p^{k+1} \tag{1}$$

The sub-collection $\mathcal{S}^{(K)}$, with K being the highest value of all k where the $\mathcal{S}^{(k)}$ is not empty, is therefore the sub-collection with the biggest-sized sets. Table 1 shows an example of sets from Cormode et al. with Table 2 showing how these sets are categorized at initialization by the size rule (1) presented before with $p = 2$.

Beginning with the sub-collection $\mathcal{S}^{(K)}$ containing the largest sets, the algorithm selects a set S_i in the sub-collection, and removes all elements from the set which are already covered in C. If the set-size is still $\geq p^k$, the algorithm adds the set index i to our solution Σ and adds the (remaining) elements to our already-covered-list C; If set size is now below p^k, the updated set will be appended to the sub-collection $\mathcal{S}^{(k')}$, where the set-size fulfills rule (1). This procedure is repeated for each set in $\mathcal{S}^{(k)}$ for each k in descending order until there are only sets containing only one element. Then the algorithm simply checks if the element in the set is not already covered and updates Σ and C if so.

Sets and sub-collection example in Cormode et al.[6]

1	ABCDE	6	EH
2	ABDFG	7	CI
3	AFG	8	A
4	BCG	9	E
5	GH	10	I

Table 1: Sets with words universe = {A,B,C,D,E,F,G,H,I}

$\mathcal{S}^{(k)}$	Set-Size	Set (#)
$\mathcal{S}^{(2)}$	4-7	ABCDE(1), ABDFG(2)
$\mathcal{S}^{(1)}$	2-3	AFG(3), BCG(4),
		GH(5), EH(6), CI(7)
$\mathcal{S}^{(0)}$	1	A(8), E(9), I(10)

Table 2: Sub-collections at initialization of DFG

Disk-Friendly Greedy (DFG):[6]

```
*Initialization.* $\Sigma = \emptyset$, $C = \emptyset$, choose a value for $p$, build sub-collections $\mathcal{S}$ with rule (1)
*Loop.* while $k = K \to 1$:
    *Loop.* foreach $S_i$ in $\mathcal{S}^{(k)}$:
        * *If* $|S_i \backslash C| \geq p^k$:
            1. Add $i$ to $\Sigma$
            2. Update $C$
        * *Else*:
            1. Let $S_i \leftarrow S_i \backslash C$
            2. Add updated $S_i$ to sub-collection $\mathcal{S}^{(k')}$ with
            set-size $|S_i|$ fulfilling rule (1)
*Loop.* foreach $S_i$ in $\mathcal{S}^{(0)}$:
    *If* $|S_i \backslash C| = 1$:
        1. Add $i$ to $\Sigma$
        2. Update $C$
*Return.* $\Sigma$
```

Example. The algorithm starts greedy-like with one of the largest sets, so it selects one of the sets in $\mathcal{S}^{(2)}$ with set-size 4-7, e.g. set $S_1 = \{A, B, C, D, E\}$. Since there are no elements covered so far ($C = \emptyset$), $i = 1$ is added to $\Sigma \leftarrow \{1\}$, and all elements of S_i are added to $C \leftarrow \{A, B, C, D, E\}$. The algorithm continues with the next set in $\mathcal{S}^{(2)}$: $S_2 = \{A, B, D, F, G\}$. Since C is not empty anymore, it checks whether set S_2 contains covered elements and deletes them if so. Since S_2 contains three covered elements (A, B, D), S_2 is updated to $S_2 \leftarrow \{F, G\}$ and is appended to a different sub-collection to fulfill the size-rule (1), in this case to $\mathcal{S}^{(1)}$.

Continuing with the next sub-collection $\mathcal{S}^{(1)}$, the algorithm selects the first set $S_3 = \{A, F, G\}$. In considering of C, the only covered element in this set is A. We let $S_3 \leftarrow \{F, G\}$; since this set still fulfills rule (1), we add $i = 3$ to $\Sigma \leftarrow \{1, 3\}$, take its remaining elements and add them to $C \leftarrow C \cup \{F, G\}$; so until now we have covered seven of nine elements: $C = \{A, B, C, D, E, F, G\}$. Checking the other sets in $\mathcal{S}^{(1)}$, the algorithm shortens every set: $S_4 \leftarrow \emptyset$, $S_5 \leftarrow \{H\}$, $S_6 \leftarrow \{H\}$, $S_7 \leftarrow \{I\}$, $S_2 \leftarrow \emptyset$. Since none of them fulfills the size rule (1) they are all appended to $\mathcal{S}^{(0)}$.

Now the algorithm takes on the second main loop: it checks every remaining set if it still contains an uncovered element after putting out all the already covered ones. In this case these are S_5 and S_7 - their indices are added to $\Sigma \leftarrow \{1, 3, 5, 7\}$ and their elements H and I are added to $C = \{A, B, C, D, E, F, G, H, I\}$. The algorithm now finishes and returns Σ: S_1, S_3, S_5 and S_7 are the sets that the algorithm has chosen. In this case it has done its job perfectly: all different elements in our example sets have been covered.

2.3 Simulated-Annealing

The second algorithm that is investigated in this work is based on the simulated annealing algorithm, a statistical technique to find the global optimum of a function which has become famous for solving optimization problems[10, 11, 12, 13]. In metallurgy, annealing is a procedure to harden metallic structures by cooling and heating materials in a controlled way so that they build larger crystals. This makes the materials free from defects an more reliable. The simulated annealing was inspired by this technique and can be applied as a metaheuristic to approximate optima.

Jacobs and Brusco have published a note, in which they introduced a local-search heuristic (LSH) for the set cover problem based on simulated annealing. As the simulated annealing is a metaheuristic, an initial, feasible solution Σ is required to start with. Jacobs and Brusco used a greedy heuristic based on a fast approach by Balas and Ho. Having a feasible solution, the simulated annealing tries to find better solutions. As metric for the goodness of a solution, the solution size is chosen in this case. At initialization, three values have to be set: the temperature T to start with, the temperature length TL, which is the number of iterations, that are done at a certain value of T, and the cooling factor CF, which reduces the temperature after each step. The values of these three parameters are not very sensitive to the quality of the solution - therefore almost the same values as in [10] have been used, despite of the temperature length, which was reduced to only 5 iterations for saving computation time.

Parameter	T	TL	CF
Value	1.3	5	0.9

Table 3: Chosen parameters for **Simulated Annealing**

At each step of the algorithm a new solution Σ' is searched with a search heuristic (I), which will be explained later on. After finding a new solution, we compare its length with the last solution's length. If the new solution is shorter, than it is our new best solution by now. Otherwise, we only accept the solution with a probability of $e^{-\delta/T}$; if the probability is too low, we discard the new solution. These steps are done TL-times, until the temperature is cooled down by CF $T = T \times CF$ and the loop starts again. After a chosen running time t, the algorithm stops optimizing.

Simulated Annealing:[10]

```
Initialization. $\Sigma$ = Feasible Solution, $|\Sigma|$ = Length of $\Sigma$
                $T$ = Initial Temperature, $TL$ = Temperature Length, $CF$ = Cooling Factor
                $t$ = maximum running time .
Loop. while True:
    Loop. while $i = 1 \to TL$:
        Find a new solution $\Sigma'$ and compute its length $|\Sigma'|$ with a Search Heuristic (I)
        Let $\delta = |\Sigma'| - |\Sigma|$
        * If $\delta <= 0$ :
            New solution is current solution: $\Sigma = \Sigma'$
            New solution is also the best solution: $\Sigma_{Best} = \Sigma'$
        * Else:
            Choose a random number between $r = [0, 1]$
            * If $r < e^{-\delta/T}$: New solution is current solution: $\Sigma = \Sigma'$
    Stop. If running is higher than $t$.
    Let $T = T \times CF$
Return. $\Sigma_{Best}$
```

For finding a new solution at every iteration step Jacobs and Brusco introduced a local search heuristic. They remove randomly an amount of sets in the solution Σ and bring then sets with a certain size to solution Σ' again. In this work - for saving running time - the size of the sets was disregarded. The amount of removed sets N is scaled by a neighbourhood-scale p: $N = |\Sigma| \times p$. In this approach, random sets were added to solution if they re-cover any element, that got uncovered before.

Search Heuristic based on [10]:

```
Initialization. $\Sigma' = \Sigma$ = Feasible Solution, $U = \emptyset$ = Uncovered Elements,
                $N = |\Sigma| \times p$
Loop. while $n = 0 \to N$ :
    1. Randomly remove a set $S_i$ from $\Sigma'$
    2. Update $U$
If $U = \emptyset$: Return. $\Sigma'$
Loop. while $U! = \emptyset$:
    Randomly choose a set $S_j$ from sets, that are not in solution
    Loop. for each element E in $S_j$:
        *If $E \in U$:
            1. Update $U = U \setminus E$
            2. Add $S_j$ to solution $\Sigma'$
        *Else:
            Continue.
Return. $\Sigma'$
```

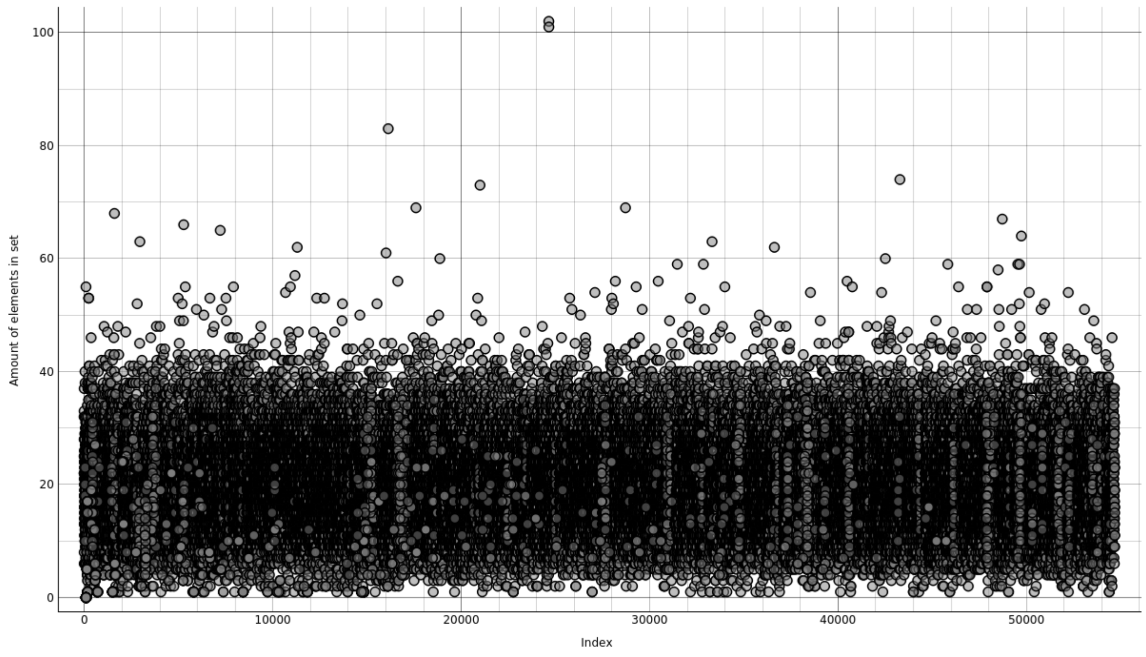


Figure 1: Sizes of the sets in Reuters collection.

3 Experiments and Results

3.1 Data-Set and Test Environment

Data-Set. The data set, that the algorithms were tested on, was the Reuters collection from the Natural Language Toolkit (NLTK) library, that is written in Python programming language for natural language processing in English. This text corpus gives us a total amount of 29181 words to be covered from a sets-universe of 54716 sets. Since the focus of this work lies on the computation and coverage performance of the algorithms, the computation time, amount of covered elements and coverage rate were considered as measuring values.
System. For testing the algorithms a system with a 2.5GHz Intel Core i7 system running Ubuntu 18.04 LTS was used. The CPU has 2 cores, 4 threads, 512KB L2 cache per core, 4MB L3 cache and integrated Intel HD Graphics 520 (Skylake GT2). The system works with 16GB of RAM on a 256GB SSD.

3.2 Greedy Heuristic by Balas and Ho

To compare the two extensions, the greedy heuristic was implemented first. As a fast approach to gain a feasible greedy solution to the set cover problem, a procedure by Balas and Ho was used. The running time was quite constant at a value of 202-204 seconds and 19443 sets got into solution. A summary of the results can be found in Table 4.

	Coverage Rate	Amount of sets in solution	Best running time in seconds
Value	1.0	19443	202.413

Table 4: Results of **Greedy Heuristic** by Balas and Ho

3.3 Disk-Friendly Greedy

As the DFG only needs the sets universe and a p-value to create a solution, a interval from 1.005 to 2.000 was tested on the p-value.

Cormode et al. proposed to limit the size of sub-collections created to a value of 250 sets for each sub-collection $\mathcal{S}^{(k)}$. That actually makes the DFG algorithm extremely fast, but as shown in Figure 1, most set sizes are in the same low range of 40 elements per set. With this limitation, the running time did not pass 60 seconds, but the coverage rate also was lower than .40, which is not satisfying.

Taking higher sizes, e.g. $|\mathcal{S}^{(k)}| = 3000$ raises both the coverage rate and the solution size. The solution with the highest coverage rate in this case made it to a value of .86 at $p = 1.075$, but with a large amount of sets in solution: 33793 sets. That corresponds to nearly 62% of the whole data set. Whereas the time elapsed for this run was with 139.4 seconds much faster than the greedy algorithm. Figure 2 and 3 present a summary of these tests.

Without any sub-collection limit, the DFG needed almost the same time as the standard greedy heuristic with about 200-250 seconds. The coverage rate then went up to .90-.99. On closer inspection, it became apparent, that these high coverage rates were only reached by the algorithm adding nearly all sets to solution. For reaching a 0.9 rate, there were added over 40,000 sets; for passing the 0.95 mark it needed 45,000 sets and for beating 0.99 over 52,000.

3.4 Simulated-Annealing

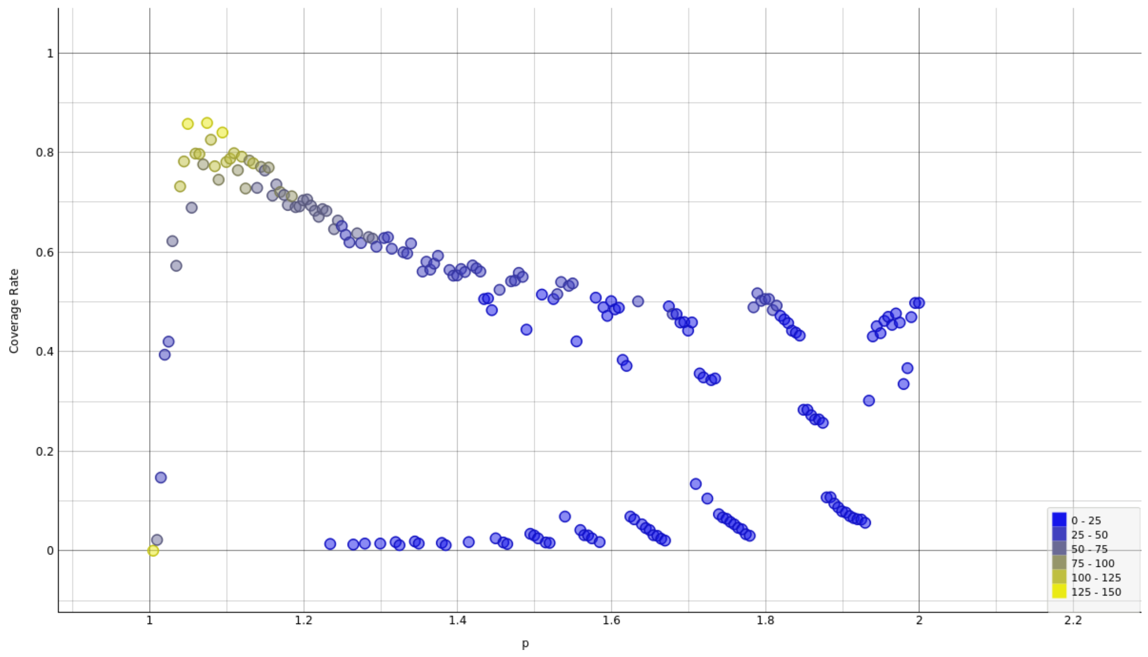
At implementation of the simulated annealing it turned out to be a challenge to find a elegant way of checking at each creation of a new solution, if this solution re-covers all un-covered elements that got uncovered by removing sets randomly. It was no expedient solution to check all sets in the whole solution and withdraw all covered elements. In the end it was decided to save all mentions of a element in a hash-array, which was carried on and updated at each solution-finding step. As this updating procedure has to be done twice (at removing sets from solution and bringing other sets into again), this lets the running time shoot upwards.

The procedure as in Jacobs and Brusco, to put set sizes and amount of re-covered elements in relation by introducing a additional parameter for search depth, was not tested out. The reason for disregarding this parameter was, that the computation took to long with the solution not being noticeable shorter. As a result it should be recorded, that the important thing in this approach is finding a accurate and fast heuristic to get another feasible solution at every iteration of the simulated annealing.

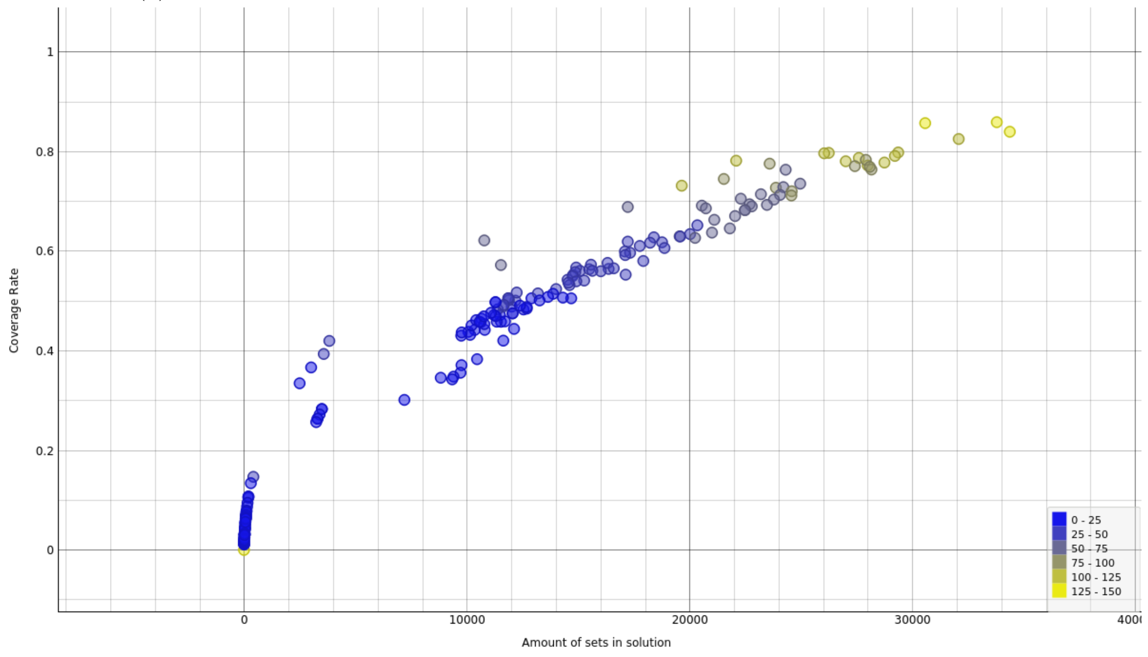
Apart from the running time, the outstanding power of the simulated annealing lies in finding the optimal solution. The algorithm was run three times for 1500 seconds at each test, with the neighbourhood-scale increasing from .1 to .5 from test to test. As starting solution, the solution of the greedy heuristic by [11] with 19443 sets was used. The best solution found was with only 10419 sets almost half as long as the greedy solution it began with.

Amount of sets	Neighbourhood-Scale	Time elapsed in sec
15715	0.1	1530.449
11572	0.2	1798.165
10727	0.3	1973.523
10640	0.4	2109.728
10419	0.5	2000.038
19443	initial value by Greedy Heuristic	

Table 5: Results of the **Simulated Annealing**



(a) p-Value plotted against coverage rate, colored by the time elapsed in seconds.



(b) Amount of sets in solution plotted against coverage rate, colored by the time elapsed in seconds.

Figure 2: Results of the **DFG**-Algorithm

4 Conclusion

In this work several ways of solving the Set Cover Problem were tested. The aim was to improve the often used greedy heuristic, which has become kind of a standard procedure for solving this problem. Different aspects were put in the foreground: computation time, coverage rate and size of the solution, that the various approaches lead to. It became apparent, why the strategy of a greedy algorithm is so convincing: with little effort for implementing a greedy heuristic you can get good feasible solutions in quite short time.

Nevertheless, it can make sense to adapt the greedy heuristic for one's specific application. While only testing the Disk-Friendly-Greeedy by Cormode et al. on one data set, which has quite small sets, it can well make sense to use it for very large data sets as mentioned in the paper. Finding a fitting limitation of sub-collection size might give a good trade-off between running time and solution size, which could beat the standard greedy in running time by lengths and save a lot of disk memory. Another point in favour of this algorithm is, that it does not need a universe of elements as reference. Without comparing the solution space with the given element universe, the algorithm manages to find a solution. In a real world example, in which the element universe is not known yet, the DFG can solve this problem extremely fast.

Putting the optimal solution to the problem in the spotlight, the simulated annealing definitively is a well-performing tool, which will find better solutions to end up with. The challenge is to find a valid neighbouring solution at every step within a reasonable time. Good indexing approaches could solve the running time problem in this particular procedure. If you have enough resources and the quality of the solution needs to be best, simulated annealing is a good way to reach this aim.

References

[1] B. Saha and L. Getoor, "On maximum coverage in the streaming model & application to multi-topic blog-watch," in *Proceedings of the 2009 siam international conference on data mining.* SIAM, 2009, pp. 697–708.

[2] D. Zorbas, D. Glynos, P. Kotzanikolaou, and C. Douligeris, "Solving coverage problems in wireless sensor networks using cover sets," *Ad Hoc Networks*, 2010.

[3] F. Chierichetti, R. Kumar, and A. Tomkins, "Max-cover in map-reduce," in *Proceedings of the 19th international conference on World wide web.* ACM, 2010, pp. 231–240.

[4] M. J. Kearns, U. V. Vazirani, and U. Vazirani, *An introduction to computational learning theory.* MIT press, 1994.

[5] T. Grossman, A. Wool *et al.*, "Computational experience with approximation algorithms for the set covering problem," *European Journal of Operational Research*, vol. 101, no. 1, pp. 81–92, 1997.

[6] G. Cormode, H. Karloff, and A. Wirth, "Set cover algorithms for very large datasets," in *Proceedings of the 19th ACM international conference on Information and knowledge management.* ACM, 2010, pp. 479–488.

[7] U. Feige, "A threshold of ln n for approximating set cover," *Journal of the ACM*, 1998.

[8] P. Indyk, S. Mahabadi, R. Rubinfeld, A. Vakilian, and A. Yodpinyanee, "Set Cover in Sub-linear Time *," 2018. [Online]. Available: http://www.mit.edu/{~}vakilian/publication/IMRVY18.pdf

[9] R. M. Karp, "Reducibility Among Combinatorial Problems," in *50 Years of Integer Programming 1958-2008.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 219–241. [Online]. Available: http://link.springer.com/10.1007/978-3-540-68279-0{_}8

[10] L. W. Jacobs and M. J. Brusco, "Note: A local-search heuristic for large set-covering problems," *Naval Research Logistics (NRL)*, 1995.

[11] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study," in *Combinatorial optimization.* Springer, 1980, pp. 37–60.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," Tech. Rep. 4598, 1983. [Online]. Available: https://pdfs.semanticscholar.org/beb2/1ee4a3721484b5d2c7ad04e6babd8d67af1d.pdf

[13] V. Černỳ, "Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm," *Journal of optimization theory and applications*, vol. 45, no. 1, pp. 41–51, 1985.

[14] N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, J. . Seffi, and . Naor, "The Online Set Cover Problem," 2003.

[15] T. A. Feo and M. G. C. Resende, "Greedy Randomized Adaptive Search Procedures," *Journal of Global Optimization*, vol. 6, pp. 109–133, 1995.

[16] D. S. Johnson, "Approximation algorithms for combinatorial problems," *Journal of computer and system sciences*, vol. 9, no. 3, pp. 256–278, 1974.