Proceedings of the 19th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 689–694
DOI: 10.15439/2024F7060
ISSN 2300-5963 ACSIS, Vol. 39

An Improved Genetic Algorithm for Set Cover using Rosenthal Potential

Dena Tayebi
University College Dublin
Ireland
Email: dena.tayebi@ucdconnect.ie
ORCID: 0000-0001-6447-7930

Saurabh Ray
New York University
Abu Dhabi
Email:saurabh.ray@nyu.edu
ORCID: 0009-0005-6708-125X

Deepak Ajwani
University College Dublin
Ireland
Email:deepak.ajwani@ucd.ie
ORCID: 0000-0001-7269-4150

***Abstract*—A major issue with heuristics for set-cover problem is that they tend to get stuck in a local optimum typically because a large local move is necessary to find a better solution. A recent theoretical result shows that replacing the objective function by a proxy (which happens to be Rosenthal potential function) allows escaping such local optima even with small local moves albeit at the cost of an approximation factor. The Rosenthal potential function thus has the effect of *smoothing* the optimization landscape appropriately so that local search works. In this paper, we use this theoretical insight to design a simple but robust genetic algorithm for weighted set cover. We modify the fitness function as well as the crossover operator of the genetic algorithm to leverage the Rosenthal potential function. We show empirically this greatly improves the quality of the solutions obtained especially in examples where large local moves are required.**

Our results are better than existing state of the art genetic algorithms and also comparable in performance with the recent local search algorithm NuSC (carefully engineered for set cover) on benchmark instances. Our algorithm, however, performs better than NuSC on simple synthetic instances where starting from an initial solution, large local moves are necessary to find a solution that is close to optimal. For such instances, our algorithm is able to find near optimal solutions whereas NuSC either takes a very long time or returns a much worse solution.

I. Introduction

THE SET cover problem (SCP) is one of the most fundamental and well studied problem in theoretical computer science. An instance of the set cover problem consists of a ground set X and a set $\mathcal{S} = \{S_1, \cdots, S_m\}$ of subsets of X. The goal is to pick the smallest subset $Y \subseteq S$ so that the union of the sets in Y is equal to X. In the weighted version of the problem, each set has an associated non-negative weight and the goal is to minimize the total weight of the sets in Y.

A simple greedy algorithm [1] is known to give an $O(\log n)$-approximation (even for the weighted variant) and under standard complexity theoretic assumptions this is asymptotically the best achievable in polynomial time [2]. However, instances of the set cover problems that arise from practical applications are often not worst case instances and one can hope to do better. For instance, significantly better algorithms are known for geometric instances of the set cover problem [3].

A major issue with heuristics for the set-cover problem is that they tend to get stuck in a local optimum. Even for the unweighted setting, it is not difficult to construct examples in which there are local optima for which an arbitrarily large change is necessary to improve the solution. A recent paper [4] finds a way to make local search as good as the greedy algorithm for the general set cover problem (which, as we mentioned before, is the best possible in polynomial time). The main idea is to replace the objective function by a proxy called the Rosenthal potential [5]. With just this change (modulo a few technical details), a local search algorithm which only adds or removes one set from the current solution in any step yields an $O(\log n)$ approximation! While this result in itself is only of theoretical interest, we believe that the idea of changing the objective function is powerful and is likely to have a practical impact since a large number of heuristics for optimization problems are based on local search. In this paper, we present a simple approach to incorporate this idea into the Genetic Algorithm metaheuristic, which yields promising empirical results. In addition to changing the objective function, we need one more crucial ingredient: a *crossover* operator (called the *minimalization operator*) that also utilizes Rosenthal potential function. We also use an idea similar to *simulated annealing* and slowly fade away the effect of the Rosenthal potential so that in the end we are left with the original objective function for the (weighted) set cover problem.

II. Related Work

Given the fundamental importance of set cover problem, a large number of algorithmic techniques have been developed for this problem. These include exact algortihms (e.g., [6]) and approximation algorithms (e.g. [7]). In addition, a number of heuristics and metaheuristics have been developed for this problem which is also the focus of this paper. A significant research focus in this area has been on unweighted set cover

algorithms that deal with instances where all sets have a uniform weight (also called unicost SCP). These include a greedy randomized adaptive search procedure (GRASP) [8], element-state configuration checking to cut down search spaces [9], a local search algorithm based on "electromagnetism" theory [10] and an adaptive row weighting algorithm [11]. In contrast, the weighted set cover heuristics often rely on MIP or MaxSAT formulations (e.g.,[12]), metaheuristics (such as simulated annealing [13], genetic algorithm [14], bee colony [15]), local search (e.g., [12], [16]) and greedy heuristics (e.g. [17]).

Very recently, a local search heuristic NuSC [16] was proposed that outperforms the other heuristics on benchmark instances. This heuristic combines the strengths of algorithmic preprocessing (to reduce the search space), a greedy algorithm (for generating an initial solution), Tabu Search (to remove some local moves for consideration for a limited time) and local search moves based on a carefully designed scoring functions to add and remove subsets. We use NuSC as the state-of-the-art baseline to compare our genetic algorithm and show that while it performs very well on the benchmark instances because of its careful engineering, it is easy to generate instances where it fails to provide good solutions in a reasonable time.

Genetic algorithms is an extremely popular metaheuristic framework that is widely used in the design of optimization heuristics. We refer the reader to a recent survey [18] on the past, present and future of genetic algorithms. For set cover problem, Beasley and Chu [14] gave a genetic algorithm that has been widely used. In their genetic algorithm, they used a fitness-based fusion crossover opreator, a variable mutation rate and a heuristic feasibility operator tailored specifically for the set covering problem. We use this work as one of the baselines for comparison.

III. Rosenthal Potential Function and Set Cover

The primary inspiration for this work is a recent paper of Gupta, Lee and Li [4] who found a way to “redeem" local search work for the weighted set cover for which it apriori seems doomed. A simple example which shows that local search does not work is the following. We have a set system in which the ground set has n elements. We have one set of weight 1 which covers all ground set elements, and we have n sets of weight $\epsilon \ll 1/n$ each of which covers a distinct element of the ground set. In this case, the optimal solution consists of the sets with weight ϵ. However, if we start with the set of weight 1 as our initial feasible solution, we cannot reduce the objective function (the sum of the weights of sets in the solution) by making a *small* change. The only way to reduce the objective function is to remove the set of weight 1 from the solution and add all n sets of weight ϵ to the solution. Depending on how small ϵ is, the locally optimal solution consisting of the set of weight 1 can be arbitrarily bad compared to the optimal solution. This apparent obstacle is bypassed in [4] via *non-oblivious local search* (NOLS) (introduced in [19], see also [20], [21]). The idea is to minimize a *potential function* different from the objective function. Let X denote the ground set and $\mathcal{S}$ denote the set of subsets of X in the given instance. At any point in time, the algorithm of [4] maintains a feasible solution $\mathcal{F} \subseteq \mathcal{S}$ along with a mapping $c : X \mapsto \mathcal{F}$ that maps each element $x \in X$ to a set $F \in \mathcal{F}$ that covers x. The potential function the algorithm seeks to minimize is the Rosenthal potential defined as:

$$\phi(\mathcal{F}, c) = \sum_{F \in \mathcal{F}} w(F) \cdot H(|c^{-1}(F)|),$$

where $w(F)$ is the weight associated with the set F, $H(m)$ is the m^{th} Harmonic number $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{m}$ and $c^{-1}(F) = \{x \in X : c(x) = F\}$. The algorithm starts with any feasible solution $\mathcal{F}$ and any valid mapping c. In any local move, it adds a set $S \in \mathcal{S}$ that is not in the optimal solution and modifies the mapping c as follows: for every $x \in S$, we change $c(x)$ to S. Any set F in the current solution for which $c^{-1}(F)$ becomes empty as result of this, is dropped from the current solution. A relatively simple analysis then shows that the solution obtained by repeatedly applying local moves that reduce the potential until no such move is possible, has weight at most H_k times the weight of the optimal solution. Here, k is the size of the largest set in $\mathcal{S}$. Note that the *size* of the local moves in this algorithm is just 1 - we only add *one* set to the current solution at a time and remove the sets made redundant by it. Local moves of larger size are also considered in the paper for lower order improvements in the approximation factor. From our point of view, the key takeaway is that local search can be “redeemed" by the use of a potential function. Since local search is a powerful heuristic frequently used in practical applications, we believe that incorporating this idea into those applications will lead to practical gains. We also believe that instead of fixing a particular potential function, learning it from data could be more effective. However, we don’t pursue that avenue in this paper.

IV. Our Genetic Algorithm

Our algorithm follows the standard framework of genetic algorithms. The key components that are distinct from standard genetic algorithms are the following.

- *Fitness function.* We use the following slightly modified form of the Rosenthal potential as the fitness function:

$$\Phi(\mathcal{F}, c) = \sum_{F \in \mathcal{F}} w(F) \cdot H^{\alpha}(|c^{-1}(F)|),$$

where $\alpha \in [0, 1]$ is a parameter.

The parameter α used in the modified Rosenthal potential function is initially non-zero and is slowly decreased to 0. Note that when $\alpha = 1$, the objective function is the Rosenthal potential and when $\alpha = 0$, it is simply the sum of the weights of the sets in $\mathcal{F}$, which is the original objective of the weighted set cover problem. In this way, our algorithm initially avoids getting stuck in a local minima, but later focuses only on moves that improve upon the original objective. We note that this is somewhat similar to the idea of simulated annealing. For some instances, we even start the algorithm with an $\alpha > 1$.

One difficulty with computing Φ is that in addition to a feasible soution $\mathcal{F}$, it requires the map c. One way to avoid this is to define c implicitly as the mapping that minimizes the potential. However, finding such a mapping is a non-trivial optimization problem. Instead, we choose a mapping c greedily as follows. We first map each ground set element to one of the sets covering it uniformly at random. We then do one or more rounds of the following: we go over the elements one by one and find the optimal set it should be mapped to while considering the mapping of all other ground set elements fixed. Our experiments show that typically just one round suffices to obtain a good mapping. In future references to the potential function, we will not explicitly define c and assume that it is chosen by the greedy algorithm.

- *Crossover operator.* We combine two feasible solutions as follows. We start with the union U of two solutions which itself is a feasible solution and apply the following *minimalization* operation. This operation considers the sets in a feasible solution in some order and removes the set currently being considered from the solution iff the remaining sets still form a feasible solution. For the crossover operation, the order in which the sets in U are considered is determined by the contributions of the sets to the Rosenthal potential (with c computed greedily as described earlier).
- *Initial population.* We considered two approaches for creating an initial population of feasible solutions.

  Random Minimalization: This approach starts with feasible solution consisting of all sets and then applies the minimalization operation while processing the sets in a random order.

  Probabilistic Greedy: Recall that the greedy algorithm for set cover starts with an empty (infeasible) solution and add sets one by one until a feasible solution is obtained. The next set S added is the one that is the most cost effective i.e., it minimizes $w(S)/\nu(S)$ where $w(S)$ is the weight of the set S and $\nu(S)$ is the number of elements that are covered by S but not by previously added sets. We make one slight modification to this in order to obtain a large number of solutions. Instead of always adding the most cost effective set, we add any set S with probability proportional to its cost effectiveness $w(S)/\nu(S)$.

The remaining details of our genetic algorithm are as follows:

- *Selection operator.* We use a 2-way tournament selection operator. For this, we choose two random pairs from the current population perform a crossover on the pair obtained by choosing the fitter individual from each random pair. This process is repeated until the required number of individuals are created for the next generation. We refer the readers to an extensive survey by Goldberg and Deb [22] for a comparative analysis of different selection operators. In addition, a fraction of the top (elite) individuals are passed from one generation to the next without any crossover operation.
- *Mutation operator.* In the mutation operator, we simply add to the current solution a few randomly selected sets that are not already part of it.

V. Experimental Analysis

A. Dataset description

- We use the OR-library weighted SCP instances [6], [23], [17]. This dataset is a collection of test data sets for a variety of Operations Research (OR) problems and is divided into two sets OR-small and OR-large according to their size. We consider the OR-large instances in this paper.
- The second benchmark (Rail)[1] contains real-world weighted SCP instances that arise from an application in Italian railways.
- The third benchmark (STS) [24] is obtained from Steiner triple systems and consists of unweighted (or equivalently, uniformly weighted) SCP instances.

B. Experimental Set up

Next, we describe our experimental set up. Our code is implemented in C++11 and compiled using g++ with optimization flags -O3. All running times are measured on a server with an AMD EPYC 7281 16-Core Processor with 32 threads. Each core has a boost speed of 2.7 GHz.

[1] http://people.brunel.ac.uk/∼mastjjb/jeb/orlib/files/

The server has a total shared memory of 96 GB and a total L3 cache of 32 MB.

For the baseline NuSC, we used the publicly available implementation[2]. For the genetic algorithm for set cover from Beasley and Chu [14], we couldn't find any public implementation, so we implemented it ourselves based on the description in their paper.

In our genetic algorithm, the population size is set to 100 for all instances, and the number of genetic iterations is 500. The initial value of α is 1 and in each iteration, the α decays by a multiplicative factor of 0.99. We transfer 20% of elite individuals directly to the next generation without crossover operation. The mutation probability is set to 0.2 and the crossover probability is 0.8. For NuSC, we took the default values from their publicly available implementation that were tuned for the benchmark instances.

All algorithms have been run for 10 independent runs and the best and average objective function is recorded in the tables.

C. Performance on Benchmark Instances

Tables I, II and III show the comparison between the different algorithms on the weighted, unweighted and the Rail instances, respectively. Here, ILP refers to solving the integer linear programming formulation of weighted set cover using the Gurobi solver. The "obj" column of the ILP gives the optimal objective function value for the instance. As expected, the ILP takes a very long time on larger and more complex instances. We refer to our genetic algorithm with Rosenthal potential function (with α decaying from 1 by a factor of 0.99 in each iteration) and probabilistic greedy initial population as "GA (with pot. fn.)", our genetic algorithm without the Rosenthal potential function (i.e., $\alpha = 0$) as "GA (w/o pot. fn.)" the genetic algorithm from Beasley and Chu [14] as "GA-BC" and the NuSC heuristic [16] as "NuSC".

From Table I, we observe that our genetic algorithm gives near-optimal solutions on weighted instances. In fact, we get optimal solutions on these instances except for instances scpnrg1 where we get 177 instead of 176 and scpnrg3 where we get 168 instead of 166. Furthermore, we find that there is very little variance in the objective function and the time between the ten independent runs of our genetic algorithm, as indicated by the best and the average set cover weight obtained. We note that GA-BC also obtains near optimal solutions. However, this comes at the cost of a very high running time. This is because GA-BC requires a large number of iterations (generations) to obtain these solutions. The heuristic NuSC is able to achieve near-optimal solutions with significantly less running time. The results on unweighted instances and Rail instances (Tables II and III) are similar, though on these larger instances, GA-BC can't get anywhere close to optimal in reasonable running time (less than 5 minutes).

[2]https://github.com/chuanluocs/NuSC-Algorithm

D. Comparison with Genetic Algorithm without Potential Function

We observe that the high quality of the solutions from our genetic algorithm is due in large parts to the usage of Rosenthal potential function. Tables I, II and III show that if we were to optimize the weighted set cover objective directly, the objective function value of the resultant solutions would have been significantly worse. For instance, on scpnrh1 instance, our genetic algorithm with Rosenthal potential function yields an optimal solution with the objective of 63, while the genetic algorithm without it returns a solution with objective of 80 (27% away from the optimal solution).

E. Comparison with NuSC

We compare our genetic algorithm with the state of the art local search heuristic NuSC. As described in Section II, NuSC is a very recent approach (published in 2024) and is carefully engineered for weighted set cover problem benchmark instances. It has been shown to outperform a large number of existing heuristics on these benchmark instances [16].

Since our main motivation for using the Rosenthal potential was to escape local optima, we created simple synthetic instances which would require moves of large size with the standard local search as follows. We consider two collections of sets called A and B, which contain n and m sets respectively where $m \approx n/2$. There are $n+m$ sets in total. For each triple of sets, consisting of two sets from A and one set from B, we create a ground set element present in exactly those three sets. Thus, the number of ground set elements is $m \cdot \binom{n}{2}$. It can be checked both A and B are feasible solutions and B is an optimal solution to the unweighted set cover problem. Despite the large gap in sizes, if we start from A, there is no local move that improves the solution other than the move that changes A to B directly - involving a large local move (of size proportional to n). We also extend these instances by replacing each set in A by k copies of the same set for some $k > 1$. We now have $kn + m$ sets and the number of ground set elements remains $m \cdot \binom{n}{2}$.

Note that on these instances, the linear programming (LP) relaxation of the set cover integer linear program will result in all sets from the collection A getting the value $1/2$ while all sets from the collection B will get the value 0. Thus, heuristics based on LP rounding techniques or those that just return sets with non-zero LP values will fail on these instances.

Table I: Comparison of different algorithms on weighted SCP instances from the OR Library

	ILP		Our GA (with pot. fn.)		GA (w/o pot. fn.)		GA-BC		NuSc	
Instance	obj	time	min-obj (avg)	time	min-obj (avg)	time	min-obj (avg)	time	min-obj (avg)	time
scpnrg1	176	791.9	177(177.6)	418.54	195(195.7)	387.5	178(180.1)	2124.2	176(176)	0.08
scpnrg2	154	318.8	154(155.3)	417.67	165(167.5)	402.4	158(159)	2613.4	154(154)	0.14
scpnrg3	166	3510.5	168(168.8)	419.99	183(184.5)	378.8	168(169.2)	2921.7	166(166)	2.16
scpnrg4	168	2339.2	168(169.8)	420.10	189(192.1)	398.2	170(171.6)	2431.3	168(168)	99.28
scpnrg5	168	8099.3	168(169)	420.10	188(188.8)	365.7	170(171)	2812.2	168(168)	2.55
scpnrh1	63	300259.5	63(63.7)	498.95	80(80.3)	335.9	64(64.8)	3701.3	63(63)	3.49
scpnrh2	63	121093.7	63(63.9)	499.42	78(78.9)	387.3	64(64.8)	3503.3	63(63)	0.44
scpnrh3	59	12897.6	59(60)	494.22	65(67)	412.4	60(61)	3723.9	59(59)	1.43
scpnrh4	58	26613.8	58(58.8)	491.69	65(66.5)	387.2	59(59.9)	3402.3	58(58)	0.91
scpnrh5	55	33843.5	55(55.8)	477.54	62(63)	404.2	55(58.2)	3801.1	55(55)	0.5

Table II: Comparison of different algorithms on unweighted SCP instances from the OR library and the Steiner triple system (sts) instances

	Size	ILP		Our GA (with pot. fn.)		GA (w/o pot. fn.)		GA-BC		NuSc	
Instance	row×column	min-obj	time	min-obj (avg)	time	bj	time	min-obj (avg)	time	min-obj (avg)	time
scpclr10	511×210	25	3.2	25(26.7)	15.2	47(48.3)	20.9	25(26.7)	28.9	25(25)	0.0
scpclr11	1023×330	23	88.6	25(25.2)	230.8	54(57)	219.1	25(26.2)	35.6	23(23)	0.04
scpclr12	2047×495	23	1304.0	26(26.2)	610.3	44(45.6)	596.8	26(26.5)	39.3	23(23)	0.35
scpclr13	4095×715	23	19772.7	25(25.2)	1211.3	43(44.5)	1102.9	25(26.5)	46.7	23(23)	0.61
scpcyc06	240×192	60	1002.4	62(62.1)	21.0	70(74.2)	21.1	66(68.0)	61.4	60(60)	0.0
scpcyc07	672×448	144	1002.1	148(148.8)	87.1	161(165.5)	83.2	160(160.8)	66.3	144	0.03
scpcyc08	1792×1024	342	1003.3	360(362.1)	283.4	392(398.1)	291.3	402(408.8)	79.5	344(344)	30.52
scpcyc09	4068×2304	772	1001.5	850(885)	1413.1	892(900.8)	1398.3	911(917.3)	115.7	780(780)	796.36
scpcyc10	11520×5120	1798	1000.9	1992(1996.8)	3098.3	2087(2092.6)	2871.3	2108	217.6	1794	340.15
scpcyc11	28160×11264	3968	1004.3	4104(4108.4)	4873.4	4687(4693.3)	4242.4	4398(4404.5)	783.4	3968(3968)	288.08
sts135	3015×135	103	–	106(106.5)	117.1	118(119.6)	129.6	106(106.5)	126.8	103(103)	158.67
sts243	9801×243	198	–	204(204.5)	508.1	227(227.8)	563.1	206(206.5)	618.3	198(198)	0.0
sts405	27270×405	335	–	344(345.7)	832.3	379(282.4)	862.4	350(250.5)	922.4	336(336.5)	10.33
sts729	88452×729	617	9142.1	628(630.1)	4974.9	687(690.3)	4871.3	652(653.1)	2118.2	617(617)	52.19

Table III: Comparison of different instances on Rail instances

	Size	ILP		Our GA (with pot. fn.)		GA (w/o pot. fn.)		NuSc	
Instance	row×column	min-obj	time	min-obj (avg)	time	min-obj (avg)	time	min-obj (avg)	time
rail507	507×63009	174	104.8	182(182.5)	873.1	297(299.4)	814.6	174(174)	528.79
rail516	516×47311	182	56.7	189(189.3)	1841.4	286(288.2)	1791.5	182(182)	2.95
rail582	582×55515	211	89.2	216(216.8)	2487.3	298(299.1)	2413.3	211(211)	50.46
rail2536	2536×1081841	689	6277.6	726(727)	4481.5	954(958.5)	4173.9	699(699.4)	185.51
rail2586	2586×920683	951	11092.8	1108(1109.2)	6723.2	1985(1985.6)	6034.3	960(961.5)	866.2

Table IV: Comparison of our Genetic algorithm and NuSC on synthetic instances. Ins(m,n,k) is an instance with m sets in collections B, n sets in collection A and k copies of each set in A.

	Size	ILP		Our GA (with pot. fn.)		GA (w/o pot. fn.)		NuSc	
Instance	row×column	obj	time	min-obj (avg)	time	min-obj (avg)	time	min-obj (avg)	time
Ins(7,13,1)	546×20	7	0.03	7(7)	0.2	7(7)	0.2	7(7)	205.9
Ins(11,20,1)	2090×31	11	0.21	11(11)	1.3	19(19)	45.5	11(11)	403.5
Ins(16,30,1)	6960×46	16	1.06	16(16)	2.2	29(29)	169.9	16(16)	2540.3
Ins(51,100,1)	252450×151	51	789.9	51(51)	180.6	99(99)	3000.0	99(99)	3000
Ins(7,13,2)	546×33	7	0.05	7(7)	1.2	12(12)	13.0	7(7)	981.0
Ins(11,20,2)	2090×51	11	0.34	11(11)	3.1	19(19)	49.2	19(19)	3000
Ins(16,30,2)	6960×76	16	1.60	16(16)	17.8	29(29)	187.2	29(29)	3000
Ins(51,100,2)	252450×251	51	615.3	51(51)	214.2	99(99)	3000.0	99(99)	3000

As seen in Table IV, our genetic algorithm (with $\alpha = 4$ and decay factor of 0.98) always obtains the optimal solution on these instances. In contrast, NuSC is often quite far from the optimal solution even after the time-out of 3000 seconds. This is particularly true for instance that are large or instances that have multiple copies of A. For instance, on Ins(51,100,1), NuSC obtains a solution of 99 while the optimal solution has the objective value of 51. For these instances, NuSC is reporting the sets from the collection A in the returned solution while the actual optimal solution consists of sets from the collection B. The results with 4, 6 and 8 copies of sets in A are similar to that of 2 copies.

Again, we note that without the Rosenthal potential function, our genetic algorithm would have returned solutions similar to the NuSC heuristic or even worse. Thus, we conclude that the main reason why our genetic algorithm is able to do local moves of larger size on these instances is the usage of Rosenthal potential function.

VI. Conclusion

Our experiments indicate that incorporating the Rosenthal potential has a significant impact on the quality of the solution obtained. While our genetic algorithm is able to match the quality of the solution obtained for benchmark instances, NuSC is faster on those instances. The difference in running times stems primarily from the large populations sizes maintained in a genetic algorithm. On the other hand, the advantage of the our genetic algorithm with modified Rosenthal potential is clear when we have instances requiring large local moves. For instance, on synthetic instances requiring large local moves, our algorithm gets optimal solutions in time that is often two orders of magnitude less than the state-of-the-art heuristic NuSC (which typically fails to obtain the optimal solution even after a long time). It is an interesting challenge to combine the advantages of both the algorithms. Another interesting direction is to find other applications where modifying the objective function improves the practical performance of local search.

References

[1] V. Chvatal, "A greedy heuristic for the set-covering problem." *Math. Oper. Res.*, vol. 4, no. 3, pp. 233–235, 1979. doi: https://doi.org/10.1287/MOOR.4.3.233

[2] I. Dinur and D. Steurer, "Analytical approach to parallel repetition," in *STOC*. ACM, 2014. doi: https://doi.org/10.1145/2591796.2591884 p. 624–633.

[3] K. Clarkson and K. Varadarajan, "Improved approximation algorithms for geometric set cover," *Discrete Computational Geometry*, vol. 37, pp. 43–58, 2007. doi: https://doi.org/10.1007/S00454-006-1273-8

[4] A. Gupta, E. Lee, and J. Li, "A local search-based approach for set covering," in *SOSA*. SIAM, 2023. doi: https://doi.org/10.1137/1.9781611977585.CH1 pp. 1–11.

[5] R. W. Rosenthal, "A class of games possessing pure-strategy nash equilibria," *Int. Jour. of Game Theory*, vol. 2, pp. 65–67, 1973.

[6] J. E. Beasley, "An algorithm for set covering problem," *European Journal of Operational Research*, vol. 31, no. 1, pp. 85–93, 1987. doi: https://doi.org/10.1016/0377-2217(87)90141-X

[7] N. Bansal, A. Caprara, and M. Sviridenko, "A new approximation method for set covering problems, with applications to multidimensional bin packing," *SIAM J. Comput.*, vol. 39, pp. 1256–1278, 2009. doi: https://doi.org/10.1137/080736831

[8] J. Bautista and J. Pereira, "A grasp algorithm to solve the unicost set covering problem," *Computers & Operations Research*, vol. 34, no. 10, pp. 3162–3173, 2007. doi: https://doi.org/10.1016/j.cor.2005.11.026

[9] Y. Wang, S. Pan, S. Al-Shihabi, J. Zhou, N. Yang, and M. Yin, "An improved configuration checking-based algorithm for the unicost set covering problem," *EJOR*, vol. 294, no. 2, pp. 476–491, 2021. doi: https://doi.org/10.1016/j.ejor.2021.02.015

[10] Z. Naji-Azimi, P. Toth, and L. Galli, "An electromagnetism metaheuristic for the unicost set covering problem," *EJOR*, vol. 205, no. 2, pp. 290–300, 2010. doi: https://doi.org/10.1016/j.ejor.2010.01.035

[11] C. Gao, X. Yao, T. Weise, and J. Li, "An efficient local search heuristic with row weighting for the unicost set covering problem," *EJOR*, vol. 246, no. 3, pp. 750–761, 2015. doi: https://doi.org/10.1016/j.ejor.2015.05.038

[12] Z. Lei and S. Cai, "Solving set cover and dominating set via maximum satisfiability," in *EAAI*. AAAI Press, 2020. doi: https://doi.org/10.1609/AAAI.V34I02.5517 pp. 1569–1576.

[13] M. J. Brusco, L. W. Jacobs, and G. M. Thompson, "A morphing procedure to supplement a simulated annealing heuristic for cost- andcoverage-correlated set-covering problems," *Ann. Oper. Res.*, vol. 86, pp. 611–627, 1999. doi: https://doi.org/10.1023/A%3A1018900128545

[14] J. Beasley and P. Chu, "A genetic algorithm for the set covering problem," *EJOR*, vol. 94, no. 2, pp. 392–404, 1996. doi: https://doi.org/10.1016/0377-2217(95)00159-X

[15] B. Crawford, R. Soto, R. Cuesta, and F. Paredes, "Application of the artificial bee colony algorithm for solving the set covering problem," *Scientific World Journal*, 2014. doi: https://doi.org/10.1155/2014/189164

[16] C. Luo, W. Xing, S. Cai, and C. Hu, "Nusc: An effective local search algorithm for solving the set covering problem," *IEEE Trans. Cybern.*, vol. 54, no. 3, pp. 1403–1416, 2024. doi: https://doi.org/10.1109/TCYB.2022.3199147

[17] T. Grossman and A. Wool, "Computational experience with approximation algorithms for the set covering problem," *EJOR*, vol. 101, no. 1, pp. 81–92, 1997. doi: https://doi.org/10.1016/S0377-2217(96)00161-0

[18] S. Katoch, S. Chauhan, and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimed Tools Appl*, vol. 80, p. 8091–8126, 2021. doi: https://doi.org/10.1007/s11042-020-10139-6

[19] S. Khanna, R. Motwani, M. Sudan, and U. Vazirani, "On syntactic versus computational views of approximability," *SIAM Journal on Computing*, vol. 28, no. 1, pp. 164–191, 1998.

[20] Y. Filmus and J. Ward, "Monotone submodular maximization over a matroid via non-oblivious local search," *SIAM Journal on Computing*, vol. 43, no. 2, pp. 514–542, 2014.

[21] V. Cohen-Addad, A. Gupta, L. Hu, H. Oh, and D. Saulpic, "An improved local search algorithm for k-median," in *SODA*. SIAM, 2022. doi: 10.1137/1.9781611977073.65 pp. 1556–1612.

[22] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in *First Workshop on Foundations of Genetic Algorithms*, vol. 1. Elsevier, 1991. doi: https://doi.org/10.1016/B978-0-08-050684-5.50008-2 pp. 69–93.

[23] J. E. Beasley, "A lagrangian heuristic for set-covering problems," *Naval Research Logistics*, vol. 37, pp. 151–164, 1990. doi: https://doi.org/10.1002/1520-6750

[24] D. Fulkerson, G. L. Nemhauser, and L. Trotter, "Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of steiner triple systems," in *Approaches to integer programming*, 1974. doi: https://doi.org/10.1007/BFb0120689 pp. 72–81.