A Genetic Algorithm for Group Steiner Tree Problem

Rebeka Čorić, Mateja Đumić, Slobodan Jelić
Department of Mathematics, J. J. Strossmayer University of Osijek
Osijek, Croatia
Email: rcoric/mdjumic/sjelic@mathos.hr

***Abstract*—In Group Steiner Tree Problem (GST) we are given a weighted undirected graph and family of subsets of vertices which are called groups. Our objective is to find a minimum-weight subgraph which contains at least one vertex from each group (groups do not have to be disjoint). GST is NP-hard combinatorial optimization problem that arises from many complex real-life problems such as finding substrate-reaction pathways in protein networks, progressive keyword search in relational databases, team formation in social networks, etc. Heuristic methods are extremely important for finding the good-enough solutions in short time. In this paper we present genetic algorithm for solving GST. We also give results of computational experiments with comparisons to optimal solutions.**

***Keywords* - genetic algorithm, group Steiner tree problem, minimum spanning tree, integer linear programming**

I. Introduction

The group Steiner tree problem (GST) is NP-hard combinatorial optimization problem that comes from wire routing problem in physical VLSI design [1]. During the last fifteen years this problem is applied in many areas such as keyword search in relational databases [2] , team formation in social networks [3], government formation problem [4], subnetwork extraction methods in bioinformatics [5], etc.

In GST, we are given an undirected graph $G = (V, E)$, $|V| = n$, $|E| = m$, with weight function $w : E \to \mathbb{R}_+$, and a family of subsets of V, $\mathcal{G} = \{G_1, \ldots, G_k\}$, $k \in \mathbb{N}$, $G_i \neq \emptyset$ which are called *groups*. The problem is to find a subtree T such that

$$\sum_{e \in E(T)} w(e)$$

is minimized and $V(T) \cap G_i \neq \emptyset$ for each $i \in [k]$, where $V(T)$ stands for set of vertices of T, and $E(T)$ for set of edges of T. Thus, GST generalizes two important combinatorial optimization problems: *Steiner tree problem* and *set cover problem*. A Steiner tree problem is one of the most important NP-hard problems in network design that admits an approximation algorithm with constant approximation ratio [6], [7]. Actually, it is NP-hard to approximate within ratio less then $96/95$ [8]. In the Steiner tree problem, we are given an undirected graph $G = (V, E)$, with edge-weight function $w : E \to \mathbb{R}_+$ and subset of vertices $\emptyset \neq R \subseteq V$ that are called *terminals*. Vertices in $V \setminus R$ are called *Steiner vertices*. The task is to find a minimum-weight subtree T that spans all terminals. It is easy to observe that the Steiner tree problem is reducible to a special case of GST problem where size of each group is at most one. A set cover problem is the second one, but not less important since it generalizes a dozen of other combinatorial problems. We are given a set of elements U and a family $\mathcal{U}$ of subsets of U such that $\bigcup_{S \in \mathcal{U}} = U$. We say that subfamily $\mathcal{R} \subseteq \mathcal{U}$ is a set cover with respect to the instance $(U,\mathcal{U})$ if every $u \in U$ is covered by at least one set from $\mathcal{R}$. The *set cover problem* introduced by Chvátal [9] is to find a subfamily $\mathcal{R}$ of minimum size. The more general version of the problem is typically called the *weighted set cover problem* where each set from family $\mathcal{U}$ has a nonnegative weight associated with it. It is known that the set cover problem cannot be approximated by approximation ratio better than $(1 - o(1)) \ln n$, unless NP contains slightly superpolynomial time algorithms [10]. For a given set cover instance $(U,\mathcal{U})$ we construct the star graph whose leaves are associated with sets in $\mathcal{U}$. Each element in U defines a group.

This paper is organized as follows: in section 2 a short overview of previous work regarding solving GST by various methods and solving variations of Steiner Tree Problem by using genetic algorithm is given. Also, in section 2 is given brief overview of our contribution. Definition of genetic algorithm and its adaptation for GST can be found in section 3. In section 4 integer linear programming (ILP) model is briefly explained. Experimental results together with implementation details of used methods are given in section 7. Finally, in section 8 a short conclusion is given.

II. Previous work

Garg et al. [11] give a polylogaritmic approximation for the GST problem based on novel randomized rounding scheme where the input graph is a tree. They also extended their results to general graphs by tree-metric embedding technique. Halperin and Krauthgamer [12] proved that it is NP-hard to approximate GST with the ratio better than $\Omega(\log^2 k)$, where k is the number of groups, even when the input graph is a tree. Approximation algorithms for GST that use (randomized) rounding of the solution of LP relaxation, motivate a research on approximation algorithms for fractional GST where integer variables are relaxed [13].

There are also some approaches that use heuristics for solving GST. Duin et al. [14] gave a set of heuristics for GST that use transformation to the undirected Steiner

problem in graphs. Nguyen et al. [15] used ant colony optimization to solve GST. Kapsalis et al. in [16] use genetic algorithm to solve graphical Steiner tree problem. Esbensen [17] uses genetic algorithm to solve the Steiner problem in a graph and compares obtained solutions with two deterministic heuristics. Haouari and Siala [18] use hybrid Lagrangian genetic algorithm to solve the prize collecting Steiner tree problem where the genetic algorithm fully exploits both primal and dual information produced by Lagrangian decomposition of a minimum spanning tree formulation of the problem. Haghighat et al. [19] proposed genetic algorithm in which genotype representation is based on Prüfer number for solving Steiner tree optimization with multiple constraints. Julstrom [20] uses genetic algorithm in order to solve rectilinear Steiner problem.

To the best of our knowledge, there are no papers that solve group Steiner tree problem by using genetic algorithm so this paper presents first heuristic approach for GST that is based on genetic algorithm. The flow-based integer linear programming formulation of GST is used to find an exact solution on relatively small instances. Heuristic approach, presented in this paper, is able to find an optimal solution in almost all standard instances. The most important contribution of this paper is a significant speed-up comparing to exact solver on large problem instances.

III. GA FOR GST

In this section, genetic algorithms (GAs) and how they usually work will be briefly explained. Also, implementation of GA and its application for solving GST problem will be explained. The general idea for GAs stems from the principles of natural selection and genetics. That is the reason why certain terms in GAs are equal to the terms that can be seen in genetics. Basic component of GA is an individual. Individuals are coded as finite dimensional vectors which come from an alphabet of some cardinality. Based on the selection of alphabet, individuals can be represented in various ways. Some of them are [21]: array of bits, permutation array, matrix representation, floating point vector, integer vector etc.

The strength of genetic algorithms is that they use a population of solutions, rather than only one solution, and that way one can explore broader set of possible solutions to a given problem. In order to obtain the best solution, one should determine some fitness function which can compare solutions and determine which solution is better. In GA, it is important to determine size of the population, choose individual representation and determine fitness function. After that, GA can be started and evolve solutions for a given problem by using the following steps [22]:

1) **initialization:** first, desired number of individuals for the initial population is generated randomly, or if some prior information about the problem is known, it is used for generating the individuals;
2) **evaluation:** every individual from current population is evaluated by given fitness function
3) **selection:** two or more individuals are selected from current population by some selection algorithm in order to reproduce; usually, better individuals have higher probability of being selected for reproduction in order to maintain better solutions in a population;
4) **crossover:** selected individuals are combined by some crossover operator in order to produce one or more children;
5) **mutation:** produced children are mutated with a given probability in order to maintain population diversity;
6) **replacement:** individuals obtained by selection, crossover and mutation replace (with some probability) worst or randomly selected individual from current population.

Aforementioned steps (except initialization) are repeated until some stopping criterion is met. Stopping criteria are usually: maximum number of generations, maximum number of evaluations, number of generations without improvement of solution fitness, achieving known optimal solution etc. GAs can be sequential and parallel [21]. Furthermore, sequential GAs can be steady-state GA or generational GA. Steady-state GA chooses two parents from population, makes one child, mutates it and evaluates it. With some probability, child can then be inserted in population (in that case some other solution candidate gets thrown out) or rejected. In generational GA in every step whole new population of children is created which replaces old parent generation. If in that process one does not want to lose best found solution, elitism can be introduced, i.e. the best current known solution is always put in child population.

For implementation of GA for solving GST, a sequential steady-state GA with elitism is used. There are not many papers which connect Steiner Tree Problem and genetic algorithms, but in almost all of them, bitstring representation is used for representing an individual. That is the reason why in this paper it is decided to use bitstring representation (vectors containing only ones and zeroes) for individuals as well. Lengths of individuals are defined by number of nodes in problem graph. Every bit in bitstring represents a node in a graph. If bit is equal to one, that means that corresponding node is selected for spanning tree and if the bit is equal to zero, corresponding node in not selected for spanning tree. In the process of evaluating each individual, first, algorithm checks if every group is visited by selected nodes. If that is not the case, simple procedure is used in order to complete individual to be feasible in that regard. For every selected node, its neighbours are put in a list and if some of those neighbours are from a group that is not yet visited, one of them is randomly chosen and included in graph. The procedure is repeated until all the groups are visited or if another node that will be put in a tree cannot be found. If the procedure finishes and not all groups are visited, an individual is assigned fitness value large enough for used set of problems in the sense that individuals with that value of fitness will probably not be selected to reproduce. After

that, each individual whose nodes make sure that every group is visited, is evaluated using Prim's algorithm. If the selected nodes are not connected, Prim's algorithm gives value of -1 and individual at hand is given large fitness value same as before.

IV. ILP MODEL

Since GA gives us a solution without guarantee on its optimality, we need an exact solution that serves us as a ground truth for our computational experiments. The usual approach tries to use general-purpose ILP solvers in order to find optimal solution. We present two straight-forward ILP formulations. In the natural cut-based ILP formulation we introduce binary decision variable $x_e \in \{0,1\}$, for each $e \in E$, that is 1 when e is a part of the solution tree, and 0 otherwise:

$$\begin{aligned} \min \quad & \textstyle\sum_{e\in E} w(e)x_e \\ s.t. \quad & \textstyle\sum_{e\in\delta(S)} x_e \;\geq\; 1, \quad S \in \mathcal{S}_r, \\ & x_e \;\in\; \{0,1\}, \quad e \in E. \end{aligned} \tag{1}$$

In order to give an interpretation of (1), we introduce a new dummy node g_i for each group G_i and directed edges from g_i to each vertex $v \in G_i$. A value of variable x_e is interpreted as a capacity of edge e (i.e. the maximum amount of flow that can pass in any direction trough edge e). Constraints ensure that the capacity of each (G_i, r)-cut is at least 1. Since this constraint is satisfied for all such cuts and all groups G_i, $i \in [k]$, it is also satisfied for cuts with minimum capacity. By max-flow-min-cut theorem [23] it follows that capacities x_e ensure that we are able to send at least one unit of flow from each group G_i to r. On the other hand, integer program minimizes a total weight of reserved capacities $\sum_{e\in E} w(e)x_e$ subject to previously mentioned constraints. In the case of integrality constraints, a total weight of selected edges is minimized subject to constraint that there is at least one path from each group G_i to r.

Unfortunately, cut-based formulation has exponentially many constraints[1] and it is intractable even in the case of moderately large instances. Fortunately, GST problem has a compact integer linear programming formulation with polynomially many variables and constraints.

Now, we describe a transformation of undirected graph G to digraph $\overline{G}$ with some artificial vertices and edges as it is described on Figure 1. For each group G_i we introduce one dummy vertex g_i and directed edge from g_i to each vertex in G_i. On the other hand, each undirected edge is replaced by two directed edges with opposite directions. A new set of dummy vertices is denoted by $V_{\mathcal{G}}$, while a new set of directed edges is denoted by $\mathcal{A}$. Here, we introduce binary decision variables $f_i^{st}, x_e \in \{0,1\}$ for each $i \in [k]$, $(s,t) \in \mathcal{A}$ and $e \in E$. Variable f_i^{st} represents an amount of flow that vertex g_i sends to r trough directed edge (s,t). As it is previously described, x_e can be interpreted as a capacity of two directed edges that are introduced for undirected edge e.

[1] a number of cuts separating groups form r is exponential function of input size

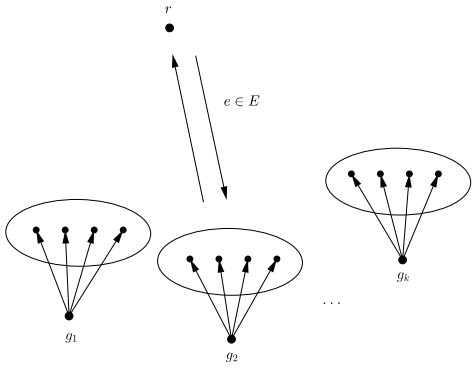

Fig. 1. A construction of the graph $\overline{G}$.

$$\begin{aligned} \min \quad & \textstyle\sum_{e\in E} w(e)x_e \\ s.t. \quad & \textstyle\sum_{vt\in\delta^+(v)} f_i^{vt} - \sum_{sv\in\delta^-(v)} f_i^{sv} = \\ & \begin{cases} 0 & , \quad v \neq r, g_1, \ldots, g_k \\ 1 & , \quad v = g_i \\ -1 & , \quad v = r \end{cases} \quad , \quad i \in [k], \\ & f_i^{st}, f_i^{ts} \leq x_e, \quad \{s,t\} \in E, i \in [k], \\ & f_i^{st}, x_e \in \{0,1\}, \quad (s,t) \in \mathcal{A}, e \in E, i \in [k]. \end{aligned} \tag{2}$$

Like in the case of cut-based formulation, we want to reserve capacities x_e such that from each vertex g_i the one unit of flow can be sent to r. These unit flows are not sent simultaneously. First constraint in (2) represents a law of flow conservation. In other words, a total amount of flow into a vertex $v \neq r, g_1, \ldots, g_k$ must be equal to the total flow out of that vertex. One unit flow enters the network at vertex g_i and exits from the network at r. Second constraint ensures that at most x_e units of flow can pass trough directed edges (s,t) and (t,s) that are introduced due to the undirected edge e.

Since flow-based formulation has polynomially many variables and constraints, we use (2) for finding optimal solutions (Section V).

V. COMPUTATIONAL EXPERIMENTS

The flow-based ILP model, described in section IV, is implemented in C++ using MILP solver Gurobi [24] and graph library Lemon [25]. GA is implemented by using ECF and Boost library [26]. Selection strategy used in this implementation is steady state tournament of size 3. For crossover, uniform crossover [27] operator was used and as for mutation, simple (single bit) mutation was used with probability of mutating a bit of 0.5%. After brief tuning phase, population size of 500 individuals and mutation rate of 0.7 (i.e. on average 7 out of 10 new individuals

will get mutated) is selected. As a stopping criterion, maximum number of 50 generations is determined. For every problem instance, the described GA is run 10 times. All tests are conducted on Linux machine with AMD Ryzen 7 2.2 GHz Eight-Core Processor, 31 GB of RAM, GCC 5.4.0. All experiments are run on only one core. We tested our algorithms on set of instances that is generated according to the procedure described in [14], [15]. Our set of instances contains three types of instances: RANDOM, EUCLIDEAN and GRID, with $n \in \{10, 40, 70, 100, 400, 700\}$. Random instances contain weighted graphs where weights are sampled from uniform distribution on segment $[0, 10]$. The weighted graphs in Euclidean and grid instances are derived from set of n points in plane where the weight of the edge, if such exists, is an Euclidean/Manhattan distance between its endpoints.

Although flow-based ILP formulation has polynomially many variables and constraints, the size of ILP grows fast enough such that branch and cut process cannot find optimal solution in reasonable time. Computational times for ILP solver in Figure 2 suggest that the number of groups k significantly affects running times since the number of flow variables grows linearly with k. It is the reason why some instances with larger number of k cannot be solved within time limit of 600 or 7200 seconds. The largest instance that we solved to optimality, with ILP solver in 2573.45 seconds, is the grid instance with $n = 100$, $m = 200$ and $k = 25$.

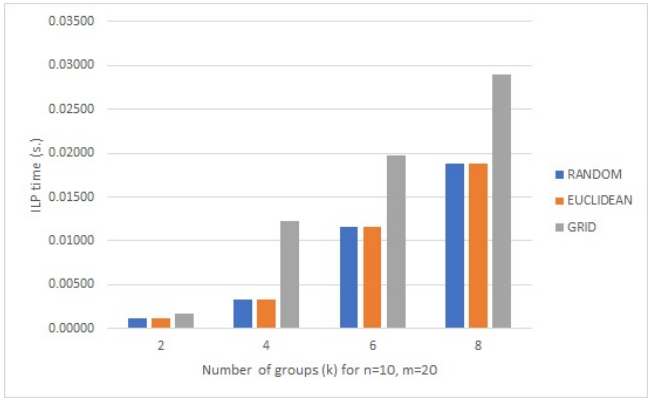

Fig. 2. Computational time for ILP solver with regard to k.

On the other hand, computational time of GA is slow increasing function with respect to the instance size. In table I we observe that GA is slower on smaller GRID instances, but the speed-up factor (calculated as ILP time over GA time) explodes when instance dimensions grow. Table I shows only those instances where ILP solver found a solution in under 600s and for all that instances GA found optimal solution too. Similar pattern can be observed on RANDOM and EUCLIDEAN instances as can be seen in tables II and III.

Interesting observations can be found on instances that are not solved to optimality, but the ILP solver returned the best found solution within the given time limit. GA returned better or equally good solution in about 82.85% instances of all tree types. There are several instances that are not solved by GA or it returned slightly worse solution than ILP solver. In Figure 3 improvements of objective function value are given for GA. Figure shows objective value obtained by GA (in blue) and improvement (difference between objective value obtained by ILP in 600 seconds and objective value obtained by GA) in orange. Also, the percentage of improvement is given for each of depicted instances which is calculated as value of improvement over objective value obtained by ILP in 600 seconds. Figure 4 gives improvements on large instances where GA found much better solutions. Similar behaviour can be observed in RANDOM and EUCLID instances, so figures for those cases are omitted.

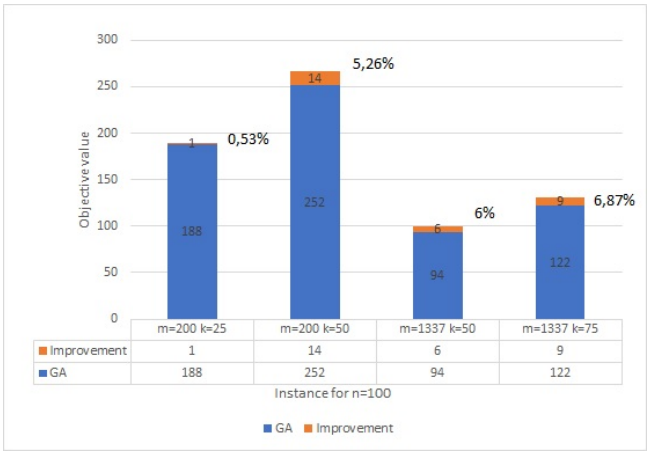

Fig. 3. Objective value improvement for GRID instances (n=100).

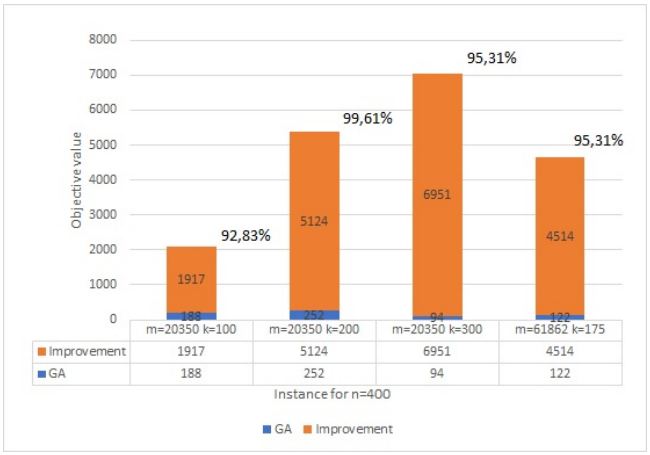

Fig. 4. Objective value improvement for GRID instances (n=400).

VI. Conclusion

Comparison of computational experiments confirmed that GA finds better solution than ILP solver in almost all benchmark instances, that are generated according to the procedures in [14], [15]. Improvements of solution quality, for instances where optimal solutions are not known, increases as the size of instances grows. ILP solver that uses flow-based model is computationally inefficient even in the case of moderately large instances. This motivates us to find another ILP formulation(s) for GST. In this paper

TABLE I
SPEEDUP ON GRID INSTANCES

n	m	k	Opt. sol.	GA time(s)	ILP time(s)	Speedup
10	20	2	0	0,416911	0,00175	0,004190343
10	20	4	14	0,445866	0,01232	0,027638349
10	20	6	21	0,465224	0,01978	0,042523602
10	20	8	24	0,477765	0,02898	0,060657436
10	21	2	0	0,416782	0,00049	0,001170876
10	21	4	8	0,436231	0,00655	0,015019565
10	21	6	14	0,451295	0,00941	0,020840027
10	21	8	23	0,470552	0,02936	0,062386304
40	80	10	41	0,542979	0,77733	**1,431600485**
40	80	20	72	0,633147	839,14208	**1325,351105**
40	80	30	93	0,715044	572,09693	**800,0863248**
40	235	10	20	0,555644	6,22750	**11,20771753**
70	140	17	55	0,723776	808,11554	**1116,527131**
100	200	25	94	0,850832	2573,45130	**3024,629182**

TABLE II
SPEEDUP ON RANDOM INSTANCES

n	m	k	Opt. sol.	GA time(s)	ILP time(s)	Speedup
10	20	2	0	0,415962	0,00117	0,002815161
10	20	4	7	0,453622	0,00328	0,00723069
10	20	6	17	0,470323	0,01155	0,024549086
10	20	8	13	0,469291	0,01876	0,039979458
10	21	2	2	0,427177	0,00078	0,001814236
10	21	4	17	0,442047	0,02677	0,060561434
10	21	6	18	0,448869	0,01916	0,042680604
10	21	8	16	0,475875	0,02320	0,048756501
40	80	10	24	0,523203	0,28057	0,536250748
40	80	20	65	0,599342	807,70074	**1347,645815**
40	80	30	99	0,702115	119,55635	**170,2803002**
40	235	10	13	0,608102	10,65462	**17,52110172**
70	140	17	40	0,677868	292,62090	**431,6782928**
70	673	17	16	0,9044	101,78776	**112,5472789**
70	1206	17	12	0,986232	3553,69938	**3603,309745**

TABLE III
SPEEDUP ON EUCLID INSTANCES

n	m	k	Opt. sol.	GA time(s)	ILP time(s)	Speedup
10	20	2	0	0,418271	0,00080	0,001907854
10	20	4	19	0,441973	0,00946	0,02140176
10	20	6	14	0,443973	0,01773	0,039934861
10	20	8	18	0,468536	0,03446	0,073539707
10	21	2	0	0,416872	0,00042	0,0010147
10	21	4	14	0,442232	0,00585	0,013219306
10	21	6	16	0,452036	0,01644	0,036368785
10	21	8	24	0,466493	0,03462	0,074211189
40	80	10	48	0,516337	0,52757	**1,021753235**
40	80	20	91	0,598032	279,75385	**467,7907737**
40	80	30	118	0,655126	591,39184	**902,7146579**
40	235	10	20	0,509099	273,46640	**537,157598**
70	140	17	98	0,704743	2561,66174	**3634,887803**

we tested our GA on generated instances. Since GST has interesting applications, future work should involve further tests on real-world instances.

REFERENCES

[1] G. Reich, P. Widmayer, Beyond Steiner's Problem: A VLSI Oriented Generalization, in: Proceedings of the Fifteenth International Workshop on Graph-theoretic Concepts in Computer Science, WG '89, Springer-Verlag New York, Inc., New York, NY, USA, 1990, pp. 196–210.

[2] J. Coffman, A. C. Weaver, An Empirical Performance Evaluation of Relational Keyword Search Techniques, IEEE Transactions on Knowledge and Data Engineering 26 (1) (2014) 30–42.

[3] C.-T. Li, M.-K. Shan, S.-D. Lin, On team formation with expertise query in collaborative social networks, Knowledge and Information Systems 42 (2) (2015) 441–463.

[4] S. Jelić, D. Ševerdija, Government Formation Problem, Central European Journal of Operations Research (2017) 1–9.

[5] K. Faust, P. Dupont, J. Callut, J. van Helden, Pathway discovery in metabolic networks by subgraph extraction, Bioinformatics 26 (9) (2010) 1211–1218.

[6] J. Byrka, F. Grandoni, T. Rothvoß, L. Sanità, An improved LP-based approximation for steiner tree, in: STOC '10 Proceedings of the 42nd ACM symposium on Theory of computing, ACM Press, New York, USA, 2010, pp. 583–592.

[7] J. Byrka, F. Grandoni, T. Rothvoss, L. Sanità, Steiner Tree Approximation via Iterative Randomized Rounding, Journal of the ACM 60 (1) (2013) 1–33.

[8] M. Chlebík, J. Chlebíková, The Steiner tree problem on graphs: Inapproximability results, Theoretical Computer Science 406 (3) (2008) 207–214.

[9] V. Chvátal, A Greedy Heuristic for the Set-Covering Problem, Mathematics of Operations Research 4 (3) (1979) 233–235.

[10] U. Feige, A threshold of ln n for approximating set cover, Journal of the ACM 45 (4) (1998) 634–652.
[11] N. Garg, G. Konjevod, R. Ravi, A Polylogarithmic Approximation Algorithm for the Group Steiner Tree Problem, Journal of Algorithms 37 (1) (2000) 66–84.
[12] E. Halperin, R. Krauthgamer, Polylogarithmic inapproximability, in: Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03, ACM Press, New York, New York, USA, 2003, pp. 585–594.
[13] S. Jelic, An FPTAS for the fractional group Steiner tree problem, Croatian Operational Research Review.
[14] C. Duin, a. Volgenant, S. Voß, Solving group Steiner problems as Steiner problems, European Journal of Operational Research 154 (1) (2004) 323–329.
[15] T.-D. Nguyen, P.-T. Do, An ant colony optimization algorithm for solving Group Steiner Problem, in: The 2013 RIVF International Conference on Computing & Communication Technologies - Research, Innovation, and Vision for Future (RIVF), Vol. 16, IEEE, 2013, pp. 163–168.
[16] A. Kapsalis, V. Raywad-Smith, G. Smith, Solving the graphical steiner tree problem using genetic algorithms, Journal of the Operational Research Society 44 (1993) 397–406.
[17] H. Esbensen, Solving the graphical steiner tree problem using genetic algorithms, Networks 26 (1995) 173–185.
[18] M. Haouari, J. C. Siala, A hybrid lagrangian genetic algorithm for the prize collecting steiner tree problem, Computers Operations Research 33 (5) (2006) 1274 – 1288.
[19] A. T. Haghighat, K. Faez, M. Dehghan, A. Mowlaei, Y. Ghahremani, A genetic algorithm for steiner tree optimization with multiple constraints using prüfer number, in: EurAsia-ICT 2002: Information and Communication Technology. Lecture Notes in Computer Science, vol 2510., Springer, Berlin, Heidelberg, 2002, pp. 272–280.
[20] B. A. Julstrom, A genetic algorithm for the rectilinear steiner problem, in: Proceedings of the 5th International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, pp. 474–480.
[21] M. Čupić, Prirodom inspirirani optimizacijski algoritmi. Metaheuristike., Fakultet elektrotehnike i računarstva, Zagreb, 2013.
[22] K. Sastry, D. Goldberg, G. Kendall, Search methodologies, Springer, 2014, Ch. Genetic algorithms.
[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, S. Clifford, Introduction to algorithms, The MIT Press, 2009.
[24] I. Gurobi Optimization, Gurobi Optimizer Reference Manual., http://www.gurobi.com/documentation/.
[25] B. Dezs, A. Jüttner, P. Kovács, Lemon - an open source c++ graph template library, Electron. Notes Theor. Comput. Sci. 264 (5) (2011) 23–45.
[26] D. Jakobovic, et al., Evolutionary computation framework, http://ecf.zemris.fer.hr/ (Oct. 2015).
[27] G. Syswerda, Uniform crossover in genetic algorithms, in: Proceedings of the 3rd International Conference on Genetic Algorithms, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989, pp. 2–9.