The Empirical Cost of Generalization: Subgraph Isomorphism and Clique Search

Parenton, Alexis; Čibej, Uroš; Mihelič, Jurij; and Fürst, Luka

Abstract: ***Subgraph isomorphism and clique search are two well-known NP-complete problems, clique search being an instance of the subgraph isomorphism problem where we fix the subgraph to a specific complete graph. Various solvers have been developed for both of these problems independently. The goal of this article is to empirically compare the efficiency of both types of solvers on the clique problem, i.e., we are solving the clique problem with specialized clique solvers and with solvers for a more general problem. In this way we can gain an insight into the differences between different types of solvers and possibly identify some approaches that could be migrated from clique search solvers to subgraph isomorphism solvers and vice versa.***

Index Terms: ***graph, subgraph isomorphism, clique search, solvers, empirical comparison***

1. Introduction

By definition, NP-complete problems can be reduced to one another and by developing a solver for one, we also get a solver for the other. But in practice, this equivalence is not so simple since solvers that exploit some specifics of the problem can be much more efficient than more general solvers. In this paper we will focus on two well-known problems, one being a natural generalization of the other. The general problem is called subgraph isomorphism, where the task is to find a pattern graph G in a larger graph H. The more specific problem is the clique search, where, given a graph H, we search for a complete graph with k vertices as a subgraph. It is easy to see that this is a special case of the subgraph isomorphism problem. Even though there is such an obvious connection, the research community has been dealing with these problems rather independently. There are plenty of solvers [11] for either the clique problem or the subgraph isomorphism problem. But no comparison has been made to establish how well the clique solvers exploit the specifics of the subgraph structure and/or how large the time loss is if we use subgraph isomorphism solvers to solve a more restricted problem. In this paper, we are exploring this connection guided by the research principles as described in [4]. The article is structured as follows. Section 2 gives the basic definitions. Section 3 describes various solvers that we are going to use during our empirical evaluation and Section 4 gives the benchmark graph instances used and Section 5 shows the obtained results. Finally, Section 6 concludes the paper and gives some goals for future research.

Manuscript received December 2019.

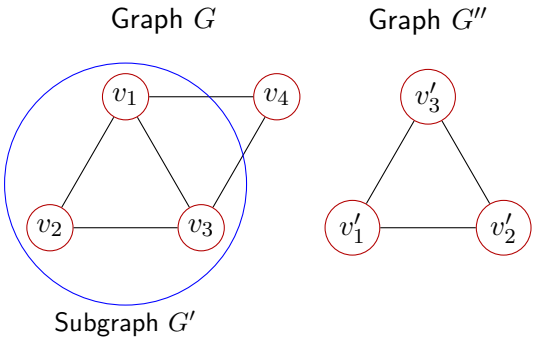
The first author is a student at Polytech Clermont-Ferrand and this work was done when visiting University of Ljubljana. Other authors are affiliated with the Faculty of Computer and Information Science, University of Ljubljana, Slovenia (e-mail: luka.fuerst@fri.uni-lj.si).

2. Definitions

In a graph, we consider the group of vertices as a set V, and the group of edges as E. A graph G is defined by those two sets V and E.

A subgraph G' is a graph where you select some of vertices in V to create a set V' . It forms the E' set with the edges in E that connect vertices in V'. The example G' shown in Figure 1. is a subgraph of G.

The maximum clique search problem is defined as follows. Given a graph $G = \langle V, E \rangle$, find the largest subset of vertices $U \subseteq V$ such that they form a clique, i.e., $\forall u, v \in U, (u, v) \in E$.



$$f_{\text{sub}}\colon (v_1, v'_3), (v_2, v'_1), (v_3, v'_2)$$

Figure 1: An instance of the subgraph isomorphism problem

3. Solvers

Both the subgraph isomorphism and the maximum clique search problem have been studied extensively in the literature. As a result of these studies, many solvers have been developed. In this section we give a brief description of the all the solvers that we used in our experiments.

3.1 Cliquer

Cliquer is a set of C routines for finding cliques in an arbitrary weighted graph. It uses an exact branch-and-bound algorithm developed by Patric Östergård [9]. It was made with the aim of being easy to use for everyone and still be flexible.

The solver includes a search for maximum cliques (weighted and unweighted). It can also search for cliques with a size or weight within a given range. Besides that, it supports weighted and unweighted graphs, but unweighted graphs have faster routines.

The first version was copyrighted in 2002 and the last update was released in early 2010. Cliquer calculates the size of the maximum clique in a subgraph, then it increases the subgraph by adding vertices one by one and recalculating the size of the maximum clique. When it finishes determining the maximum clique size, it starts the search for the suitable position of the clique.

For the second part, the order of the vertices has a major impact on the speed of the search. Cliquer can also order the vertices with different functions.

At the end, the solver returns the result with the time at each round of execution, the clique size and weight, and finally the vertices that composed the clique.

3.2 MaxCliquePara

MaxCliquePara [6] is an exact parallel maximum clique algorithm. It is based on a branch and bound algorithm. It uses the sequential MaxCliqueSeq algorithm, which was parallelized by splitting the branch and bound search tree to multiple cores, resulting in the MaxCliquePara algorithm. By splitting that way, it makes MaxCliquePara superior to older algorithms like MaxCliqueSeq, especially in our empirical testing.

It was created basically for protein product graphs and for protein structural comparisons, but it is interesting to test this one on larger DIMACS graphs. The solvers return the searching time, the number of steps, the size of the clique and the vertices of the clique in ascending order.

3.3 MaxCliqueDyn

MaxCliqueDyn [6] is also an exact algorithm for finding maximum clique in undirected graphs. The solver was developed by Janez Konc. It is based on a branch and bound algorithm and mixed with subgraph colouring.

The solver runs twice with different options. In the second execution, it performs dynamic sorting of vertices and improved colouring. For each run it return steps with the maximum clique size at each step, the final maximum clique size with the vertices, the number of steps and the execution time. For small graphs, the second run isn't different or faster, but on larger graphs it can be two or three times faster.

3.4 BBMC

The BBMC algorithm [10] is a little different. It has many variations, which are BBMCI, BBMCR, BBMCL and BBMCS, but we will focus only on the basic variation. This algorithm and its variations are all using bitsets to represent whether a vertex from the graph is in a

set or not. Algorithms using bitsets have two main advantages: the use of a bitset instead of storage implementations reduces the memory needed by a factor equal to the size of a word on the system, and common operations can be done in parallel using bit-wise operation.

The BBMC algorithms is also implemented in C++. To find solutions, they are using graph coloring.

The algorithm also returns the graph information, the searching time, the number of steps, the clique size of course and the clique vertices.

3.5 MoMC

MoMC is a combination of two different algorithms [7]. They are all implemented in C and they are based on a branch and bound algorithm. The first algorithm is SoMC which means Static ordering MaxClique Solver. It is called Static because the vertex ordering between the returned subgraph and the rest of the vertices is always consistent with the initial ordering.

The second one is DoMC which means Dynamic ordering MaxClique Solver. The main differences are in the "GetBranches" functions that will return vertices. It is called Dynamic because those vertices are not always smaller than the rest of the graph so the ordering between them should be dynamically re-defined.

Those two are exact branch and bound based solvers. MoMC is different because the solver switches cleverly between static and dynamic, depending on the graph.

MoMC returns the state of execution at each clique increment. It also returns the MaxSAT instance that can be used in SAT solvers. It finally gives the vertices of the clique.

3.6 Subgraph Isomorphism Constraint Satisfaction

SICS [3] is developed in C++ and must be compiled in C++17 or more. It is a subgraph isomorphism solver. In order to compare it with the other solvers, we need to create the clique and try to find it in the graph.

It can also read graphs from files by using a "read" family of functions. It supports the VF2, gal, galv, galve, gf, ldgraphs unlabelled and ldgraphs labelled formats.

SICS implements a large set of algorithms. There are more than 40 algorithms, but some of them are similar, so we choose the most different ones. The differences are the way to count, to store in memory, to jump quicker or lazier to other vertices, etc.

To run the solver, we needed two files. First, the graph in entry is an obligation, because this is what we want to study. After that, we need the subgraph we would like to search. In the experiment, the subgraph is a clique, corresponding to the maximum clique of the graph.

We had to create each clique graph, in the DIMACS format. When done, the algorithm can be run. The solver needs the clique first and try to find it in the graph.

In the main file, we choose to stop the execution when the first solution is found, because subgraph isomorphism solvers can find every solution, when maximum clique solver are made for one.

The solver returns the clique nodes and list the corresponding nodes in the graph. The vertex 0 is random and has no impact on the clique. The clique size and the executing time are returned

3.7 Glasgow Solver

SICS is based on the Glasgow solver [8], which is another solver based on many approaches developed for the constraint satisfaction problems. It can solve both problems that we address here, i.e., the subgraph isomorphism solver and the maximum clique problem.

Subgraph Isomorphism Solver. This is a solver for subgraph isomorphism (induced and non-induced) problems, based upon a series of papers by subsets of Blair Archibald, Ciaran McCreesh, Patrick Prosser and James Trimble at the University of Glasgow, and Fraser Dunlop and Ruth Hoffmann at the University of St Andrews. A clique decision / maximum clique solver is also included.

We can build it with a C++17 compiler. Fortunately, they had an auto-detection for the

format, but it is better to specify it.

It was logical to test this solver in the experiment, because this is a second subgraph isomorphism solver, and because SICS was based on the Glasgow solver, on an older version.

The subgraph solver is a constraint programming style backtracker, which recursively builds up a mapping from pattern vertices to target vertices. It includes inference based upon paths (not just adjacency) and neighbourhood degree sequences, has a fast all-different propagator, and uses sophisticated variable- and value-ordering heuristics to direct a slightly random restarting search.

The solver returns the number of vertices, the executing time and the vertices corresponding to the clique.

Clique Solver. In the latest version of the solver, they added a maximum clique solver. We compared it with the other maximum clique solvers, and with the subgraph isomorphism solver. It works the same way as the subgraph solver, but it estimates a clique size which will exist in the graph, and it increases the size at each executing step. The solver does not return the result the same way. It returns the number of vertices, the running time and the clique vertices.

4. Benchmark Graphs

We had a set of instances from the Second DIMACS Implementation Challenge [2]. It was better to use this instead of creating our own because those ones have special properties, and it makes the experiment more accurate. There are many graph formats that we can use. On advice from the tutors, we choose the DIMACS format, which is one of the simplest.

These are the graph classes in the benchmark set [1].

- The "C" graph family: they are random graphs generated with the number of vertices and the edge probability. Michael Trick generated the "C" family using ggen, a generating program by Craig Morgenstern.
- The "Dsjc": random graphs generated by David Johnson [5].
- The "Mann" family: clique formulation of the Steiner Triple Problem, instances generated by Carlo Mannino.
- The "Brock" family: random graphs with cliques hidden among vertices with low degree. Generated by Mark Brockington and Joe Culberson.
- The "Gen" family: generated graphs with huge clique. Instances generated by Laura Sanchis.
- The "Hamming" family: graphs generated using the hamming distance, which is distance between two different words. Generated by Panos Pardalos.
- The "Keller" family: graphs generated using the Keller conjecture on tilings using hypercubes. Generated by Peter Shor.
- The "P_hat" family: random graphs with p_hat generator which is a generalization of the classical uniform random graph generator. Those graphs have wider node degree spread and larger cliques than standard ones.

.

5. Results

The main part was the testing part. We had to test every graph with every solver. To get homogeneous results, We had to use the same virtual machine on the same computer, so we had to do it one by one. For some of the solvers and graphs, the execution took days. Having results like this was not useful, so we decided to stop the solver execution after 600 seconds.

6. Conclusions

All results are shown in Figure 2. In the middle in colour, there is the executing time for the maximum clique solvers. The maximum time is 600 seconds, because we decided to stop the execution after that time. There are 7 solvers:

Benchmark	Nodes	Edges	Clique Size	Cliquer	maxClique	MCQD	MCQD with DSV	bbMC	MoMC	Glasgow MCS	Sics	Glasgow Sub Solver
p_hat300-1	300	10933	8	**0,00 s**	0,0036 s	0,002185 s	0,002098 s	0,0016536 s	0,01 s	0,001 s	0,105 s	0,030 s
p_hat300-2	300	21928	25	0,24 s	0,0065 s	0,018401 s	0,017491 s	0,00864548 s	**0,00 s**	0,007 s	> 600 s	0,003 s
p_hat300-3	300	33390	36	343,58 s	0,8786 s	4,4042 s	2,00353 s	1,3646 s	**0,26 s**	0,762 s	> 600 s	0,426 s
p_hat500-1	500	31569	9	0,05 s	0,0130 s	0,019075 s	0,018714 s	**0,0124556 s**	0,04 s	0,015 s	0,052 s	0,006 s
p_hat500-2	500	62946	36	103,30 s	**0,1846 s**	1,10059 s	0,886646 s	0,367492 s	0,22 s	0,211 s	> 600 s	0,117 s
p_hat500-3	500	93800	50	> 600 s	46,8133 s	517,483 s	184,764 s	90,1734 s	**10,90 s**	78,80 s	> 600 s	35,214 s
p_hat700-1	700	60999	11	0,08 s	0,0436 s	0,071252 s	0,065877 s	0,0423942 s	0,10 s	**0,037 s**	186,68 s	0,036 s
p_hat700-2	700	121728	44	> 600 s	2,0233 s	10,9285 s	7,59705 s	2,98859 s	**0,84 s**	2,079 s	> 600 s	0,0352 s
p_hat700-3	700	183010	62	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	**117,97 s**	> 600 s	> 600 s	> 600 s
p_hat1000-1	1000	122253	10	0,94 s	0,2186 s	0,366106 s	0,367115 s	0,244469 s	0,42 s	**0,165 s**	14,636 s	0,037 s
p_hat1000-2	1000	244799	46	> 600 s	101,412 s	569,181 s	221,993 s	159,782 s	**20,93 s**	95,952 s	> 600 s	68,121 s
p_hat1000-3	1000	371746	???	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s
p_hat1500-1	1500	284923	12	6,79 s	**1,9988 s**	3,20349 s	3,15755 s	2,36875 s	4,11 s	2,044 s	> 600 s	0,952 s
p_hat1500-2	1500	568960	65	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s
p_hat1500-3	1500	847244	94	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s
keller4	171	9435	11	0,08 s	0,0066 s	0,02045 s	0,012205 s	0,00743196 s	0,04 s	**0,006 s**	0,088 s	0,001 s
keller5	776	225990	27	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	**147,46 s**	> 600 s	> 600 s	49,269 s
keller6	3361	4619898	59	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s
gen200_p0.9_44	200	17910	44	> 600 s	0,1838 s	1,03331 s	0,850446 s	0,471233 s	**0,03 s**	1,751 s	> 600 s	0,157 s
gen200_p0.9_55	200	17910	55	> 600 s	0,3980 s	1,1603 s	0,43667 s	0,41097 s	**0,04 s**	0,167 s	> 600 s	0,132 s
gen400_p0.9_55	400	71820	55	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	**1,27 s**	> 600 s	> 600 s	> 600 s
gen400_p0.9_65	400	71820	65	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	**0,38 s**	> 600 s	> 600 s	> 600 s
gen400_p0.9_75	400	71820	75	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	**0,24 s**	> 600 s	> 600 s	> 600 s
brock200_1	200	14834	21	5,31 s	**0,2426 s**	0,744184 s	0,446836 s	0,293971 s	0,64 s	0,316 s	> 600 s	0,189 s
brock200_2	200	9876	12	0,01 s	0,0032 s	0,006697 s	0,006288 s	0,00258277 s	0,02 s	**0,002 s**	152,589 s	0,001 s
brock200_3	200	12048	15	0,09 s	0,0119 s	0,031467 s	0,025849 s	0,010977 s	0,04 s	**0,009 s**	> 600 s	0,001 s
brock200_4	200	13089	17	0,41 s	0,0484 s	0,104879 s	0,08238 s	0,0451459 s	0,15 s	**0,039 s**	> 600 s	0,027 s
brock400_1	400	59723	27	> 600 s	251,614 s	> 600 s	> 600 s	355,327 s	**144,50 s**	209,192 s	> 600 s	62,614 s
brock400_2	400	59786	29	> 600 s	**108,139 s**	385,384 s	166,05 s	173,348 s	117,35 s	144,846 s	> 600 s	18,997 s
brock400_3	400	59681	31	> 600 s	172,536 s	> 600 s	> 600s	271,15 s	**86,67 s**	115,868 s	> 600 s	7,009 s
brock400_4	400	59765	33	> 600 s	87,1936 s	379,275 s	183,455 s	128,631 s	**23,58 s**	54,825 s	> 600 s	2,285 s
brock800_1	800	207505	23	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	453,479 s
brock800_2	800	208166	24	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	503,344 s
brock800_3	800	207333	25	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	553,375 s
brock800_4	800	207643	26	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	35,222 s
C125.9	125	6963	34	6,41 s	0,0307 s	0,104076 s	0,049653 s	0,0308461 s	**0,01 s**	0,035 s	> 600 s	0,012 s
C250.9	250	27984	44	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	**145,01 s**	> 600 s	> 600 s	232,684 s
C500.9	500	112332	57	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s
hamming8-4	256	20864	16	**0,00 s**	0,0435 s	0,071269 s	0,031809 s	0,0418765 s	0,12 s	0,038 s	0,001 s	0,007 s
hamming10-4	1024	434176	40	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s	> 600 s
MANN_a9	45	918	16	**0,00 s**	0,001 s	0,000101 s	0,000089 s	0,000065 s	**0,00 s**	**0,000 s**	0,006 s	0,000 s
MANN_a27	378	70551	126	> 600 s	0,2487 s	1,24308 s	1,38893 s	0,251938 s	**0,13 s**	0,201 s	> 600 s	0,174 s
MANN_a45	1035	533115	345	> 600 s	118,203 s	> 600 s	> 600 s	120,388 s	10,10 s	> 600 s	> 600 s	> 600 s
DSJC500_5	500	125248	13	6,44 s	1,1078 s	2,85274 s	2,14542 s	1,05236 s	2,14 s	**0,789 s**	> 600 s	0,094 s
DSJC1000_5	1000	499652	15	> 600 s	154,578 s	267,625 s	176,029 s	146,646 s	140,61 s	**80,955 s**	> 600 s	0,469 s

Figure 2: Results of the experiment

Cliquer, maxClique, MCQD, MCQD with dynamic sorting vertices, BBMC, MOMC and Glasgow maximum clique solver. The colours are the following ones: the orange is for the worst, green for the best, white for the middle one. When the orange is faded, it means that the solver is slow, but not the worst. It is the same for the green, faded green is for quick solver, but not the best. The best time is also in bold characters.

There is also a grey part, when every solver spends more than 600 seconds of executing time.

On the right, there is two columns for the subgraph isomorphism solvers. They don't have colours, because it is not useful for the experiment.

As said before, Cliquer seems to be the worst one, even though there are cases where it is the best. This is for the very small graphs. In fact, it seems to be because of the approximated time.

MaxClique does not have a lot of "best times", but it has many good times on average. In conclusion for MaxClique, the parallelism made on MaxCliqueSeq to create MaxCliquePara optimized well the solver.

For the two MCQD solvers, the executing times are mostly light orange and white, so it seems in the median or a little lower. In fact, the dynamic sorting version is slightly better than the standard one, but they share mostly the same place.

BBMC does not have a lot of orange times, so it seems to be quick. The graph colouring and the bitset memory storage obviously play an important role in the execution time.

MOMC was the best in the majority of cases. It has more than three quarters of best times. It was very powerful, especially for the "Gen" family, where its times are less than 2 seconds and the other solvers took more than 10 minutes. It is the worst, when Cliquer is the best. MOMC also allow a quick execution for large graph.

Glasgow MCS has good times. When

MOMC is not the best, it seems that Glagow MCS is the best. It is in the top, but it has the second place.

When we look at the subgraph isomorphism solvers, there is a huge difference. SICS is far behind the Glasgow solver. In fact, it is too slower than the other ones, that he mostly took more than 600 seconds. On the other hand, the Glasgow solver has better times than the Glasgow MCS, which is simply the result of the fact that the MCS estimates the clique size and increase it at each step, but in the subgraph isomorphism solver, we give the maximum clique size, which explains why this is quicker.

From the results we can conclude that the more general solvers pay a huge price (in execution time), which is of course not surprising, but it is evidence that there are still many possible properties of graphs that could be exploited in order to speed up subgraph isomorphism solvers.

References

[1] Graph instances. `http://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`. Accessed: 2019-11-30.

[2] The homepage of the dimacs challenge. `http://www.dimacs.rutgers.edu/programs/challenge/`. Accessed: 2019-11-30.

[3] The homepage of the sics solver. `https://git.sr.ht/~xnevs/sics`. Accessed: 2019-11-30.

[4] V. Blagojević, D. Bojić, M. Bojović, M. Cvetanović, J. Đorđević, Đ. Đurđević, B. Furlan, S. Gajin, Z. Jovanović, D. Milićev, V. Milutinović, B. Nikolić, J. Protić, M. Punt, Z. Radivojević, Ž. Stanisavljević, S. Stojanović, I. Tartalja, M. Tomašević, and P. Vuletić. A systematic approach to generation of new ideas for phd research in computing. In Ali R. Hurson and Veljko Milutinović, editors, *Creativity in Computing and DataFlow SuperComputing*, volume 104 of *Advances in Computers*, pages 1 – 31. Elsevier, 2017.

[5] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, May 1991.

[6] Janez Konc and Dušanka Janezic. An improved branch and bound algorithm for the maximum clique problem. *Proteins*, 4(5), 2007.

[7] Chu-Min Li, Hua Jiang, and Felip Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 2017.

[8] Ciaran McCreesh and Patrick Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In *International conference on principles and practice of constraint programming*, pages 295–312. Springer, 2015.

[9] Sampo Niskanen and Patric RJ Östergård. Cliquer user's guide. *Helsinki University of Technology*, 2003.

[10] Pablo San Segundo, Fernando Matia, Diego Rodriguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 7(3):467–479, Mar 2013.

[11] Uroš Čibej and Jurij Mihelič. Improvements to ullmann's algorithm for the subgraph isomorphism problem. *International Journal of Pattern Recognition and Artificial Intelligence*, 29(07):1550025, 2015.

Alexis Parenton is a student at Polytech Clermont-Ferrand. This work has been made while he was visiting University of Ljubljana, Faculty of Computer and Information Science as an exchange student.

Uroš Čibej received his Ph.D. in computer science from the University of Ljubljana in 2007. He is employed as a teaching assistant at the University of Ljubljana, Faculty of Computer and Information Science. His research areas include graph algorithms, backtracking, and various applications of symetries.

Jurij Mihelič received his doctoral degree in Computer Science from the University of Ljubljana in 2006. Currently, he is with the Laboratory of Algorithmics, Faculty of Computer and Information Science, University of Ljubljana, Slovenia, as an assistant professor. His research interests include algorithm engineering, combinatorial optimization, heuristics, and system software.

Luka Fürst received his Ph.D. in computer science from the University of Ljubljana in 2013. He is employed as a teaching assistant at the University of Ljubljana, Faculty of Computer and Information Science. His research areas include graph grammars, general graph theory (with a particular emphasis on graph symmetries), software engineering, machine learning, and computer programming education.