A Detailed Description of our Tools

In following we give a detailed description of the tools which can be used to verify the proof of Theorem 2. Moreover, we exemplify how the tools can be used. Even though our C++ code is platform-independent, we assume that the reader uses a Unix/Linux operating system and only give usage examples for this particular setup. (We ran our experiments in Fedora 27 and openSUSE 15.)

A.1 Enumerating Abstract Order Types

extend_order_type We provide a C++ program `extend_order_type` which reads all abstract order types on a fixed number of points n from the input file, extends it in all possible ways, and writes all so-obtained abstract order types on $n+1$ points to an output file (without duplicates). The program (see the folder `cprogram/scripts/extend_order_type/`) can be built using qmake[1] and make. The following bash command builds the program:

```
$ qmake && make
```

The File Format Concerning the file format, we have to explain a little more theory: An abstract order type can be encoded by its triple orientations

$$\Lambda_{i,j,k} \in \{-1, 0, +1\} \text{ for each } 1 \leq i, j, k \leq n.$$

Since encoding this "big lambda matrix" uses a cubic amount of bits, it is more efficient to encode the "small lambda matrix", which has the following entries:

$$\lambda_{i,j} := |\{k \in \{1, \ldots, n\} \setminus \{i, j\} \colon \Lambda_{i,j,k} > 0\}| \text{ for each } 1 \leq i, j, k \leq n.$$

Diagonal elements $\lambda_{i,i}$ are omitted (or can be set to zero). This data structure was first introduced by Goodman and Pollack [GP83].

Small lambda matrices of (non-degenerated) abstract order types fulfill $\lambda_{i,j} + \lambda_{j,i} = n - 2$ (i.e., for each two fixed points i, j, any other point either lies on the left or on the right side of the directed line through i and j), hence only entries $\lambda_{i,j}$ with $1 \leq i < j \leq n$ need to be stored. Moreover, the first point can be assumed to lie on the boundary of the convex hull, and other points can be assumed to be sorted around the first point. Such a labeling of points is called "natural labeling" and yields $\Lambda_{1,i,j} = +$ for $1 < i < j \leq n$ and $\lambda_{1,j} = j - 1$ for all $1 < j \leq n$. Consequently, elements from the first row of the small lambda matrix need not to be stored. Note that the lexicographically minimal small lambda matrix (over all labelings) – which we compute to distinguish different order types – is also naturally labeled.

Altogether, we encoded the entries $\lambda_{i,j}$ for $1 < i < j \leq n$ as 8-bit (1-byte) integers in lexicographic order, i.e,

$$\lambda_{2,3}, \lambda_{2,4}, \ldots, \lambda_{2,n}, \lambda_{3,4}, \lambda_{3,5}, \ldots, \lambda_{n-1,n}.$$

For more information we again refer to the articles by Aurenhammer, Aichholzer, and Krasser [AAK02, AK06], and the dissertation of Krasser [Kra03].

[1] In fact, no Qt-specific features are used. Alternatively to qmake one could also use cmake or just use a stand-alone Makefile

Usage of the Program The program `extend_order_type` can be used as follows:

```
./extend_order_type [n] [order types file] [parts] [from part] [to part]
```

The following describes the parameters.

- "n" is the number of points in the abstract order types from the input file,
- "order types file" is the path to the input file,
- "parts" is the number of threads in total,
- "from part" is the id of the first thread to be run, and
- "to part" is the id of the first thread not to be run.

The difference "to part"-"from part" is precisely the number of threads to be started on the local machine. As an example, to start a computation on 4 machines with 4 threads each (16 threads in total), one can simply run one of the following commands on each of the machines:

```
./extend_order_type [n] [order types file] 16 0 4
./extend_order_type [n] [order types file] 16 4 8
./extend_order_type [n] [order types file] 16 8 12
./extend_order_type [n] [order types file] 16 12 16
```

We also provide our python script `create_jobs.py`, which we used to automatically create job files for the parallel computations on the cluster.

Complete Enumeration To generate all abstract order types, we start with a binary file `n3_order_types.bin` with content "0x00" (one byte) – this encodes the unique order type on 3 points, which is described by the small lambda matrix

```
-  0  1
1  -  0*
0  1  -
```

The entry $\lambda_{23} = 0$, which is marked with a star (*), is the one entry which is actually encoded as "0x00" in the file. This file is also available in the folder `data/order_types/`.

The following command now enumerates all abstract order types on 4 points:

```
$ xxd n3_order_types.bin
00000000: 00                                       .
$ ./extend_order_type 3 n3_order_types.bin 1 0 1
n: 3
starting threads: 1
[0/1] started
[0/1] n 3       ct 0    extensions 0
[0/1] finished
all threads done.
total solutions: 2/1
$ xxd n3_order_types.bin.ext0_1.bin
00000000: 0001 0001 0001
```

The command xxd displays a hex dump of the given file. The generated output file `n3_order_types.bin.ext0_1.bin` contains 6 bytes in total, encoding the two order types on 4 points (with 3 bytes each). The first three bytes encode the order type of 4 points in convex position, which has the following small lambda matrix:

```
-  0  1  2
2  -  0* 1*
1  2  -  0*
0  1  2  -
```

The remaining three bytes encode the other order type of 4 points, which has a triangular convex hull and one interior point. Its small lambda matrix is the following:

```
-  0  1  2
2  -  1* 0*
1  1  -  1*
0  2  1  -
```

When renaming the file `n3_order_types.bin.ext0_1.bin` to `n4_order_types.bin`, one can now analogously enumerate all order types of 5, 6, ... points with

```
./extend_order_type 4 n4_order_types.bin 1 0 1
./extend_order_type 5 n5_order_types.bin 1 0 1
...
```

A.2 Enumerating Triangulations

Plantri Having the graph generator plantri [BM99][2] installed, it can be run with parameters "[number of points] -g", to enumerate all triangulations on the specified number of points in graph6 format. With the additional parameter "-c4", only 4-connected triangulations are enumerated. As an example, following command enumerates all 4-connected triangulations on 8 vertices:

```
$ ./plantri 8 -g -c4
./plantri 8 -g -c4
G|tJH{
G|thXs
2 triangulations written to stdout; cpu=0.00 sec
```

To store graphs in files, one can simply pipe the standard output to the desired file:

```
$ ./plantri 8 -g -c4 > n8c4.g6
./plantri 8 -g -c4
2 triangulations written to stdout; cpu=0.00 sec
```

Filter Triangulations Having the mathematics software system SageMath [S+18] installed, we used the Sage-scripts `filter_3tree.sage` and `filter_maxdeg.sage` to filter stacked triangulations and triangulations with maximum degree $|V| - 1$, respectively.

[2]Plantri is available from `https://users.cecs.anu.edu.au/~bdm/plantri/`. For more information on plantri, we refer to `https://users.cecs.anu.edu.au/~bdm/plantri/plantri-guide.txt`, and for more information on the graph6 format, we refer to `https://users.cecs.anu.edu.au/~bdm/data/formats.html`.

The script `filter_3tree_relabel.sage` is a slight modification of `filter_3tree.sage`, which relabels the vertices of the given graph in a way, such that the vertices 0, 1, and 2 span the initial triangle, and the k-th vertex is stacked into a triangular face of the subgraph induced by the vertices $0, 1, \ldots, k-1$.

The triangulations enumerated by plantri do not necessarily fulfill this property. The triangulations shown in Listings 3 and 4 were relabed using `filter_3tree_relabel.sage`. The respective files are available in the folder `data/triangulations/`.

The scripts `filter_property1.sage` and `filter_property2.sage` can be used to filter triangulations which fulfill Properties 1 and 2 from Section 4.4.

Edge-List Encoding We have chosen a different plain text format, which is easier to load in C++: We encode a graph by its edge list. For each graph, we write the start and end vertices of the edges $\{u_1, v_1\}, \ldots, \{u_m, v_m\}$ simply as "$u_1\ v_1\ u_2\ v_2\ \ldots u_m\ v_m$" in a line, followed by a line-break. The following example gives an illustration (continues with the 4-connected 8-vertex triangulations from before):

```
$ sage encode.sage n8c4.g6
0 1 0 2 0 3 0 4 1 2 1 4 1 5 1 6 2 3 2 6 2 7 3 4 3 7 4 5 4 7 5 6 5 7 6 7
0 1 0 2 0 3 0 4 1 2 1 4 1 5 2 3 2 5 2 6 2 7 3 4 3 7 4 5 4 6 4 7 5 6 6 7
```

This encoding can be performed using the Sage-script `encode.sage`.

Drawing Graphs Last but not least, we provide the script `draw.sage` which we used to automatically generate drawings for stacked triangulations; see Listings 3 and 4. The idea is to start with a Tutte embedding and then use global optimization heuristics[3] to optimize a certain quality function, which simultaneously maximizes edge lengths and vertex-edge distances.

A.3 Testing n-Universality

test_universal_sets We provide a C++ program `test_universal_sets` to find n-universal point sets. The program (see the folder `cprogram/scripts/test_universal_sets/`) can be built analogously to `extend_order_type` (using qmake and make), except that Minisat is required to be build as a library first.

Building the Minisat Library As described in the README file delivered with Minisat [ES03][4] (cf. `cprogram/minisat-2.2.0`), one needs to set the MROOT variable. This can be done for example with the following command:

```
$ export MROOT=$PWD
```

In the `simp` folder from Minisat, one then can run

```
$ make lib_release.a
```

to build the library. Having the `lib_release.a` built, we are now ready to build our program `test_universal_sets`. Note that, if minisat-2.2.0 is not placed inside the basis directory (where the `otlib.pro` is located), one might need to slightly adapt the project file `test_universal_sets.pro` so that the minisat headers and library are found. In particular, the following two lines might need to be adapted:

[3] Cf. `http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html`
[4] Available from `http://minisat.se/MiniSat.html`

```
INCLUDEPATH += $$OTLIBDIR/minisat-2.2.0
LIBS += $$OTLIBDIR/minisat-2.2.0/simp/lib_release.a
```

Usage of the Program

```
./test_universal_sets [n] [order types file] [graphs file] [phase]
                      [parts] [from part] [to part]
```

The parameters can be described as follows:

- "n" is the number of points in the abstract order types from the input file,
- "order types file" is the path to the input file for abstract order types,
- "graphs file" is the path to the input file for graphs,
- "phase" specifies the actions which should be performed (see below),
- "parts" is the number of threads in total,
- "from part" is the id of the first thread to be run, and
- "to part" is the id of the first thread not to be run.

Abstract order types are again encoded by their small lambda matrix as described above. For the graph file we have chosen the plain-text edge-list format described above. The phase parameter describes, what the program should test:

- If the program is ran with parameter "phase=1", then point sets are filtered out which do not fulfill the necessary conditions described in Phase 1 of Section 4.
- If the program is ran with parameter "phase=2", then for each point set all graphs from the list are tested for simultaneous embeddability. As described in Phase 2 of Section 4, we stop as soon as one graph is not embeddable, and we use a priority queue to speedup the computations.

  The parameter "phase=2" is also used to test Phase 4 of Section 4.
- If the program is ran with parameter "phase=3", then all pairs of order types and graphs are tested for embeddability. Unlike for the computations of Phase 2, we do not change the order of the list of graphs. For each given order type from the input file, a line of zeros and ones is written to a plain-text output file, where the j-th zero/one in the i-th line encodes whether the j-th graph can be embedded on the i-th point set. In the following we refer to this file as "stat"-file.

  The parameter "phase=3" is also used to test Phase 5 of Section 4.

For Phase 6 of Section 4 one can simply concatenate the stat-files obtained in Phases 3 and 5, and run CPLEX/Gurobi – no additional computations with our C++ program are necessary.

Loading Realizations from the Order Type Database It is also possible to load files from the order type database [Aic], which provide point set realizations of all order types. To do so, one simply needs to change the line

```
// #define REALIZATIONS
```

to

```
#define REALIZATIONS
```

in the source file `test_universal_sets.cpp`. In the binary files `otypes04.b08`, ..., `otypes08.b08` (available at [Aic]) each order type of $n = 4, \ldots, 8$ points is encoded by one of its realizing point sets: the points $(x_1, y_1), \ldots, (x_n, y_n)$ are encoded as "$x_1 y_1 \ldots x_n y_n$" using 1 byte per coordinate (values inbetween 0 and 255). For $n = 9$, $n = 10$, and $n = 11$, each coordinate is encoded using 2 bytes (values inbetween 0 and 65536).

A.4 Integer Programming

Gurobi Having Gurobi/Gurobipy installed [Gur18][5][6], we used the script `test_min_cover.py` to create a (Mixed) Integer Linear Programming instance from a stat-file (created from our C++ program). The script parses the input file, writes the instance to an "lp" file, and then starts the Gurobi solver to find an (optimal) solution. An instance, which is stored in an lp-file, can also be read and solved on a different machine for example via the following command:

```
$ gurobi
...
gurobi> m=read("n11_phase5_statistic_stacked.txt.instance.lp")
gurobi> m.optimize()
Optimize a model with 17533 rows, 423 columns and 1031205 nonzeros
...
Explored 295 nodes (25169 simplex iterations) in 69.54 seconds
Thread count was 6 (of 6 available processors)

Solution count 5: 12 13 14 ... 50

Optimal solution found (tolerance 1.00e-04)
Best objective 1.200000000000e+01, best bound 1.200000000000e+01, gap 0.0000%
```

When solving an instance using Gurobi (also with CPLEX), the current upper and lower bound on the optimal value is printed to the console every few seconds. When aborting the solving process (CTRL+C), the currently best solution is printed.

CPLEX An instance, which is stored in an lp-file, can be read and solved via the CPLEX Interactive Optimizer [IBM18][7] as exemplified in the following:

```
CPLEX> read instance.lp
...
CPLEX> optimize
...
```

[5] Cf. Section 12 "Python Interface" from `http://www.gurobi.com/documentation/8.1/quickstart_linux.pdf`

[6] For information on free academic versions, checkout `http://www.gurobi.com/academia/academia-center`

[7] For information on free academic versions, checkout `http://www.ibm.com/developerworks/community/blogs/jfp/entry/CPLEX_Is_Free_For_Students`

References

[AAK02] O. Aichholzer, F. Aurenhammer, and H. Krasser. Enumerating Order Types for Small Point Sets with Applications. *Order*, 19(3):265–281, 2002.

[Aic] O. Aichholzer. Enumerating Order Types for Small Point Sets with Applications. `http://www.ist.tugraz.at/aichholzer/research/rp/triangulations/ordertypes/`.

[AK06] O. Aichholzer and H. Krasser. Abstract Order Type Extension and New Results on the Rectilinear Crossing Number. *Computational Geometry: Theory and Applications*, 36(1):2–15, 2006.

[BM99] G. Brinkmann and B. D. McKay. Fast generation of some classes of planar graphs. *Electronic Notes in Discrete Mathematics*, 3:28–31, 1999.

[ES03] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of Theory and Applications of Satisfiability Testing – SAT 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[GP83] J. E. Goodman and R. Pollack. Multidimensional Sorting. *SIAM Journal on Computing*, 12(3):484–507, 1983.

[Gur18] Gurobi Optimization, LLC. Gurobi Optimizer, 2018. `http://www.gurobi.com`.

[IBM18] IBM ILOG CPLEX Optimization Studio, 2018. `http://www.ibm.com/products/ilog-cplex-optimization-studio/`.

[Kra03] H. Krasser. *Order Types of Point Sets in the Plane*. PhD thesis, Institute for Theoretical Computer Science, Graz University of Technology, Austria, 2003.

[S+18] W. A. Stein et al. *Sage Mathematics Software (Version 8.1)*. The Sage Development Team, 2018. `http://www.sagemath.org`.

[SSS19a] M. Scheucher, H. Schrezenmaier, and R. Steiner. A Note On Universal Point Sets for Planar Graphs. In *Proc. 35th European Workshop on Computational Geometry (EuroCG'19)*, pages 21:1–21:9, 2019.

[SSS19b] M. Scheucher, H. Schrezenmaier, and R. Steiner. A Note On Universal Point Sets for Planar Graphs. In *Graph Drawing and Network Visualization*, volume 11904 of *LNCS*, pages 350–362. Springer, 2019.