Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Jan Tomášek

Drawing graphs on surfaces of small genus

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: doc. Mgr. Zdeněk Dvořák, Ph.D.

Study programme: Computer Science

Specialization: Programming

Prague 2013

[Sample: Bound Sheet – a signed copy of the "bachelor thesis assignment". **This assignment is NOT a part of the electronic version of the thesis. DO NOT SCAN**.]

I am grateful for all the support I received. Most of all, I would like to thank my supervisor Docent Zdeněk Dvořák for his help during elaboration. Special thanks to my sister Marie Tomášková who helped me improve my English grammar. I would also like to thank Magister Petr Jílek for introducing me to discrete mathematic and graph theory.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague May 24, 2013 signature

Název práce: Kreslení grafů na plochách malého rodu

Autor: Jan Tomášek

Katedra / Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: doc. Mgr. Zdeněk Dvořák, Ph.D.

Abstrakt: Přesto, že existuje mnoho algoritmů pro kreslení grafů na obecné plochy, existuje překvapivě malo funkčních implementací těchto algoritmů. Práce přináší přívětivé grafické rozhraní pro kreslení grafů na kouli a toru, které umožní přehledně kontrolovat výstup a posuzovat jeho kvalitu. Program podporuje zobrazení v mnohoúhelníkové reprezentaci a v třídimenzionálním prostoru. Práce se dále zabývá existujícími algoritmy pro nalezení pěkného nakreslení grafů na tyto plochy a diskutuje jejich použitelnost v praxi. Jeden algoritmus je vybrán a implementován včetně rozboru implementace a použitých datových struktur.

Klíčová slova: kreslení grafů, plochy, torus

Title: Drawing graphs on surfaces of small genus

Author: Jan Tomášek

Department / Institute: Computer Science Institute of Charles University

Supervisor of the bachelor thesis: doc. Mgr. Zdeněk Dvořák, Ph.D.

Abstract: Although there are many algorithms for drawing graphs on arbitrary surfaces, surprisingly few of them have been practically implemented. This thesis designs a user-friendly graphical interface for drawing graphs on sphere and torus, which allows checking the result of algorithms. The program supports displaying the surfaces in the polygonal representation and in their 3-dimension embedding. Furthermore we survey the algorithms for drawing graphs on surfaces and discuss their implementability. We chose one of the algorithms and evaluate its performance.

Keywords: graph drawing, surfaces, torus

Table of Contents

Preface

There are many visualization tools for graph theory but they are designed for graphs drawn on plane only. However graph theory along with topology allows us to define graph drawing on more blizzard surfaces. So our Graph editor allows us to draw graphs on sphere and torus. These surfaces can be always described in polygonal representation, but such visualization is sometimes a bit uncomfortable. Therefor our software adds three-dimensional projection of this drawing. So our program can be used for demonstrating corresponding between polygonal representation and its projection.

Another function of our program is that it can find plane drawing of the graph if it is possible. In first chapter we define this problem more exactly and survey existing algorithm for drawing graphs. We chose one algorithm which can be used on sphere and torus. Than we evaluate performance of this implementation. However this algorithm is exponential in worst case it is surprisingly fast in average case on small graphs.

1. Introduction

1.1. Graph theory

In 1735 Leonhard Euler was trying to come up with solution for a problem called “Seven Bridges of Königsberg”. The problem was to find a walk through the city that would cross each bridge once and only once. He was looking for some elegant and practical abstraction of this problem. His abstraction laid down the foundation of graph theory. Euler was the first person who used terms like vertex and edge. He formed a brand new mathematical discipline. Graph theory can be used for modeling almost any structure which contains objects and relations between them.

Nowadays graph theory is one of the most evolving disciplines of mathematics. Graph theory is used for modeling lots of real-world problems like electric circuits, road networks, railway networks, social and information systems. Without graph theory there would be no satellite navigation in our cars, Google etc.

1.2. Basic definitions and labeling

In this paper we use standard graph theory with usual definitions and usual labeling according to Graph Theory by Reinhard Diestel [3]. Graphs are labeled with these letters: G, H or I. Vertices are labeled with u respective v. They can include a subscript if needed. In complexity formulas n denotes the number of vertices and m the number of edges.

1.2.1. Bridges

A bridge B with respect to a subgraph H of graph G is either:

1. A connected component C of $(G \setminus H)$ together with the edges (u, v) where $u \in C$ and $v \in H$, or
2. An edge (u, v) of $(G \setminus H)$ where $v \in H$ and $u \in H$ but $(u, v) \notin H$

1.3. Planar graphs

In some applications (electric circuits, subway design) a plane drawing of graph is needed. This means that there are no crossing edges in the drawing of the graph. For example in subway design, each crossing of the edges means, that the edges must be conducted at different levels. In printed circuit board design every crossing in design implies adding a new layer. It is better to have planar layout to reduce price. Graph G is called planar if there is some plane drawing of G. For example on Figure 2 and Figure 3 there are two possible drawings of K_4. The second drawing on Figure 3 is plane so K_4 is planar. On Figure 1 we can see non-plane drawing of non-planar graph $K_{3,3}$.

Also, many of graph problems are very hard in general, but become simpler for planar graphs. For example graph coloring is NP-hard in general but planar graphs can be colored using four colors only in polynomial time.

Furthermore, avoiding crossings can improve visualization, as crossings can be confused with vertices.

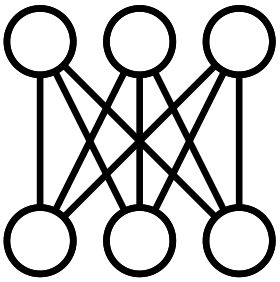

Figure 1: Non-planar graph

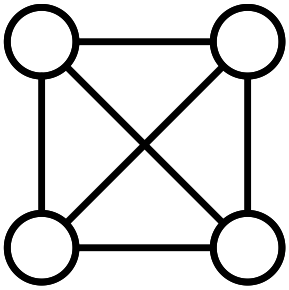

Figure 2: Planar graph with non-planar drawing

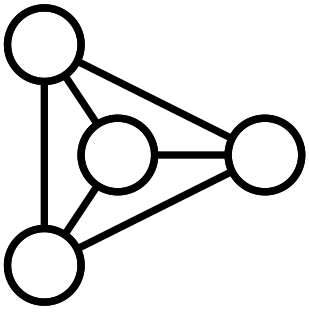

Figure 3: Planar Graph with plane drawing

1.4. Surfaces

Planar embedding can be generalized to embedding into. Examples of surfaces without boundary are the sphere (Figure 4), the torus (Figure 5) and the Klein bottle. Examples of surfaces with boundary are: the Möbius strip (Figure 6) and a cylinder. Any connected surface without boundary can be categorized in one to these three categories:

1. The sphere (Figure 4)
2. Connected sum of g tori, for $g > 0$. Example : torus(Figure 5), double torus
3. Connected sum of k real projective planes for $k >0$. Example: Klein bottle, projective plane

Surfaces in category 1 and 2 are called orientable and parameter g is called genus of the surface. Surfaces in category 3 are called nonorientable and parameter k

is their genus. So genus and orientability determines the surface without boundary up to homeomorphism.

Planar graphs are equivalent to graphs drawn on sphere. One implication is obvious: if we have a drawing of graph G in plane we can take this drawing and draw it on sphere. More interesting is the second implication. We put sphere s on plane p and find some vertex-less point on sphere O. We rotate sphere to get this point O on the top. For each vertex V of graph G from sphere drawing we imagine line l going through O and V. In the point of intersection of line l and plane p we draw a "copy of vertex" V. We can imagine this as putting a light source in point O and then drawing shadows of vertices on plane p.

Orientable surfaces are easy to imagine. Torus shape is what most people call doughnut. Another equivalent definition of orientable surface with genus g is by g times adding a handle on sphere. So torus is equivalent to sphere with one handle. This is also easy to imagine: When the sphere with handle gets smaller and smaller the handle starts to look like a torus. In road network terms we can say that road network drawn on torus without edge crossing is equivalent to road network on Earth with one bridge (usual bridge not bridge from graph theory).

It gets a bit trickier with non orientable surfaces. Already projective plane is very hard to imagine. There is no proper embedding of Projective plane in three-dimensional Euclidean space. We have to use 4D to have nice non-crossing projection. Another problem is the non-orient ability. This means that you cannot define which side is left or right. Alternative definition of Projective plane is sphere with cross-cap. Cross-cap is a "magic portal" - lines going through this "portal" come in mirror order out on the other side. So if we use this "portal" our left hand turns to right. This is why we can't define left and right side.

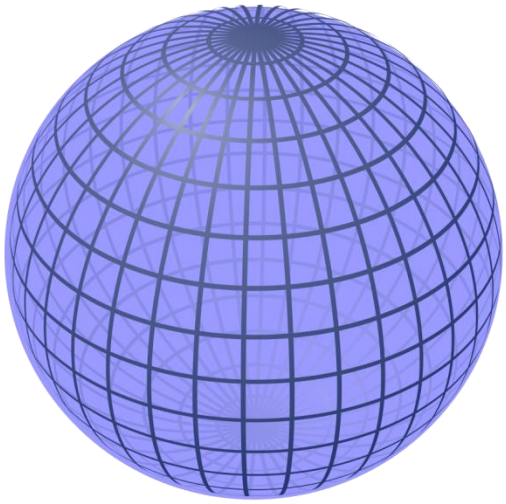

Figure 4: Sphere

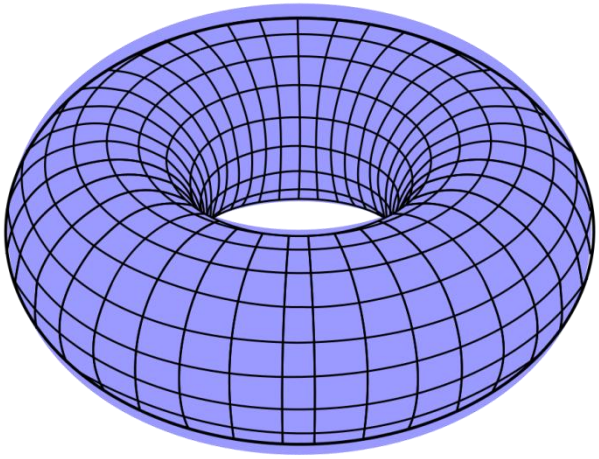

Figure 5: Torus

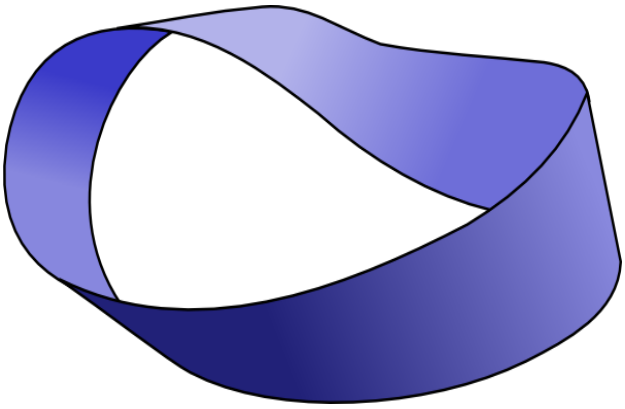

Figure 6: Möbius strip

1.5. Polygonal representation of arbitrary surfaces

Each surface can be represented by gluing sides of a polygon. Each side of the polygon is labeled with an arrow and a letter. There are always two sides with the

same letter. Sides with the same letter are connected to each other. Arrow then shows in which direction they are connected. The connection will determine if it is orientable or non-orientable surface. They always connect the beginning of an arrow with the beginning of the other arrow and the end with the end. Sphere can be represented with two side polygon with arrows going in the opposite directions. If we flip one arrow we get a projective plane. For torus (on Figure 5) and Klein bottle four sides are needed. We can see an example of constructing a torus from its polygonal representation in Figure 7.

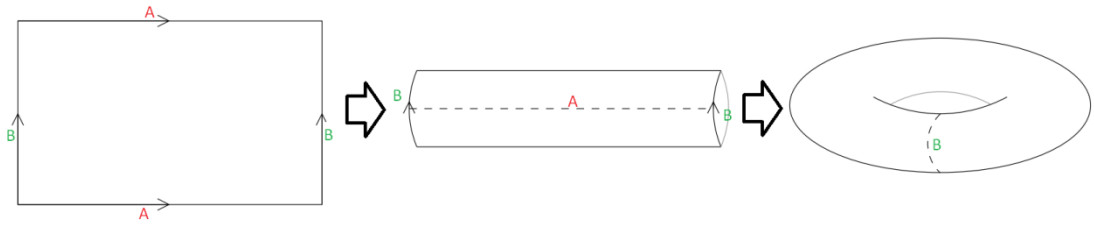

Figure 7: Constructing torus from polygonal representation

1.6. Combinatorial embedding

When looking for non-crossing embedding in surface, the first problem to solve is to find a combinatorial embedding (also called the rotation system), i.e., for each vertex we need to determine in which circular order we should draw the incident edges around the vertex.

There are several practical implementations of linear-time algorithms for planar embeddings, for example an algorithm from Boyer and Myrvold [1].

Much less has been done for other surfaces. We know that determining the genus of graph is NP-hard in general. But there are many of theoretical polynomial algorithms which determine for a fixed surface whether a graph can be embedded in this surface without crossing. Some of them even yield an obstruction (minimal subgraph which prevents the existents of an embedding) when no embeddings exists. First polynomial algorithm for embedding graphs in a fixed surface was presented in 1979 by I.S.Filotti[6]. This algorithm is nice in theory but it has very large degree of a polynomial. It can find embedding in $O(n^{\alpha+\beta})$ where α, β are constants for fixed surface. This gives quite big degree of a polynomial even for torus it gives $O(n^{188})$ algorithm. So it is impossible to use this result even for small graphs. This

result was improved in 1991 by H. Djidjev and J. H. Reif [4]. They presented an algorithm, where the degree of polynomial is fixed for each orientable surface. 8 years later in 1999 it was improved once again by Bojan Mohar. Mohar described an algorithm, which finds embedding of graph into fixed surface if and only if such embedding exists or returns subgraph that cannot be embedded. This algorithm works in linear time. So it gives us a constructive proof that for each surface without boundary are only finitely many minimal forbidden subgraphs. Even though this algorithm is linear it will be extremely hard to implement it correctly and the implementation will have enormous multiplicative constant

Let's take a look on polynomial algorithms designed for one surface only. In in paper of Martin Juvan, Joze Marincek and Bojan Mohar [8] we can find an algorithm for embedding a graph into the torus in linear time. And if the algorithm fails to find this emending it will return a smallest obstruction. Even authors of this algorithm admitted in last chapter, that the multiplicative constant hidden in this algorithm is enormous. This linear time torus embedding algorithm was improved by Martin Juvan and Bojan Mohar[7]. The algorithm was simplified a lot and the algorithm was formed into nice, one page pseudo code at the end of the paper. Due this simplify process it has lost linear time complexity. The algorithm can be theoretically implemented with time complexity $O(n^4)$. This time complexity looks alright for small graphs. Trick which helped to simplify this algorithm, was using recent results of Fiedler, Huneke, Richter and Robertson about the genus of graphs in the projective plane. This paper shows, that sometime embedding in projective plane is easy transformable to torus embedding. So the torus algorithm first tries embedding into plane, then embedding into projective plane and then combines them to yield a torus embedding. But there is no fast algorithm to find this projective plane embedding. And we have to use some of universal algorithms from previous paragraph. So we are in same trouble.

Another way to go is to construct an algorithm using forbidden minors for surface. We know that there are finite sets of forbidden minors for each surface. This was proved by Robertson and Seymour [2]. Then there is the problem of genus reduced to searching minors. This can be done in polynomial time. But those sets are known for plane and projective plane only. There are two forbidden minors for plane ($K_5$and $K_{3,3}$) and 103 forbidden minors for projective. For torus we know more than

200 000 forbidden minors and there is no prove that this list is definitive. And there is no presumption how big these lists can be. So they might be enormous. In algorithms for finding embedding using those lists will the size be consider as constant, but it can be a really big constant. And these algorithms have to try every forbidden minor from the list, so the size of lists spoils real electivity.

That leaves us with simple brute force exponential algorithms. With some additional care (adding heuristics, parallelization, etc.) this leads to solution which can be used for small graphs up to 100 vertices no problem at all. First such algorithm is described in Practical toroidality testing by E. Neufeld [11].This first attempt was not good enough. It was hard to implement and its running time was more than 4 hours for graph with 100 vertices and 300 edges. This was significantly improved by Jennifer Roselynn Woodcock in A Faster Algorithm for Torus Embedding [11]. It is based on quadratic algorithm for embedding graph into plane. This plane embedding algorithm does following: First if a graph G does not contain any cycles it is a tree or forest. And it is trivial to embed a tree. Then a subgraph H of graph G, which is cycle, is chosen and embedded into plane. Afterwards bridges of G with respect to H are embedded. When we want to generalize this algorithm for torus embedding we just do not use cycle but embedding of graphs homeomorphics to some planar obstruction (K_5 or $K_{3,3}$). But this does not give us quadratic time complexity for torus. Because we do not know which admissible face do we have to use. This leads to testing more options and backtracking. So it runs in exponential time in general. But if it is cleverly implemented, it can find emendings of 100-vertex graphs in a few seconds. We also have to test all possibilities for start embedding. There are six possibilities to embed K_5 and two possibilities to embed $K_{3,3}$ into torus. This is a perfect place to parallelize this task for six respective two threads. Each thread has to become its own copy of graph. And they can compute without any writing or reading conflicts. Another place to use more threads is by embedding paths to faces (each vertex can repeat on each face). Again we need to check all possibilities to connect path. Ideal solution is to have thread pool with as many threads as many processors cores we want to use. And every time we come to some search decision and we have some threads spare we dedicate this job to free thread from pool.

1.7. Straight line drawing construction

The next task after finding a rotation system for a graph is to draw it on a surface. We want to assign exact positions for each vertex. We have to keep the rotation order for each vertex. Moreover we can have some additional requirements for this drawing. A natural request can be a straight line drawing. Straight lines can be required in an electric circuit construction for example. From topology we know that if there exists a drawing of a graph, it can be transformed into a drawing with straight lines only. This is demonstrated on Figure 8. Read in his paper gives the first algorithm to find such drawing for a plane using a planar rotation system. But his algorithm requires a 2-connected graph without vertices of degree two. This can be solved by a 'clever' triangulation of the graph. First we triangulate faces with a vertex of degree two on the boundary. Then, we give diagonal to rest of faces. With some modifications this algorithm can be also used on a torus. This was discovered by William Kocay in 2001[9]. Both algorithms can run in $O(n)$ and return the drawing in $O(n^2)$ space.

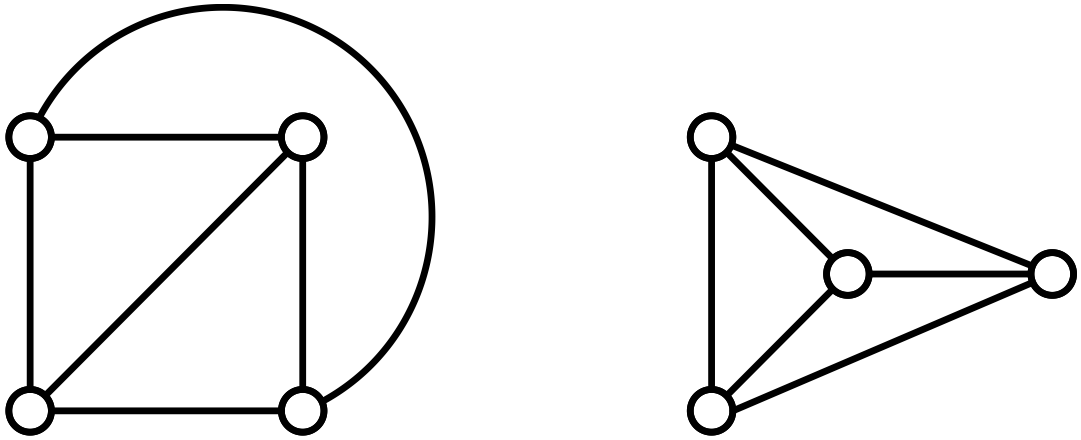

Figure 8: Normal vs. strait line drawing of K_4

1.8. Nice drawing using springs

Another way to go is to use a physical model. This is an excellent example of thinking out of the box. The idea is to imagine vertices as small balls connected with springs instead of edges. There is no proof that this will produce a correct non-crossing embedding. And it may not be successful. But in average case it produces an embedding that is 'nice to look at'. A nice example is shown on **Chyba! Nenalezen zdroj odkazů.**. The key is to guess the magical constant right. There are

many possibilities how to use this idea. Usually, we use a spring with some finite equilibrium length between vertices connected by an edge, and an "infinite" (or in a real implementation just really big) spring when there is no edge. There is now way to guess this equilibrium length; some experimentation is required. According to my experiments a tenth of available space to draw graph is good starting point for graphs with about 100 vertices. The other constant we need to decide is a spring constant. Small constant needs a huge time to converge and too large constants may never converge. We may search some real physical constants in physical-tables. But more practical is to choose a big one and after some iterations make it a bit smaller. This is faster in the beginning and stable in the end. My experiments show that a million iterations are always enough. An even better way to go is to use genetic programing for determining constants. The only thing we need to do is to get some fit function. That means to find a function that gets a drawing and tells us how good this drawing is. What we can try is the area divided by perimeter of each face. We also may add some penalization for crossing. With a reasonable starting population and a right mutation function this can converge very fast. The problem is that for each board size and each graph density, we may need other constants. Partial solution can be to define an equilibrium length not as number but as a fraction of board size. Another optimization to try is to run the algorithm several times with shuffled input and then pick up the right one according to the fit function. This algorithm can be used as standalone for finding drawings or as a final adjustment for result of any previous drawing algorithm. But this can spoil required attributes of the drawing (it can even add crossing edges). Main advantage of this spring algorithm is simplicity of implementation and running time. Its running time is $O(k\ m)$ where k is the iteration count (in most implementations a constant) and m is the number of edges in the graph. This algorithm can also be used for non-planar graphs and it with right constants it will produce a nice drawing.

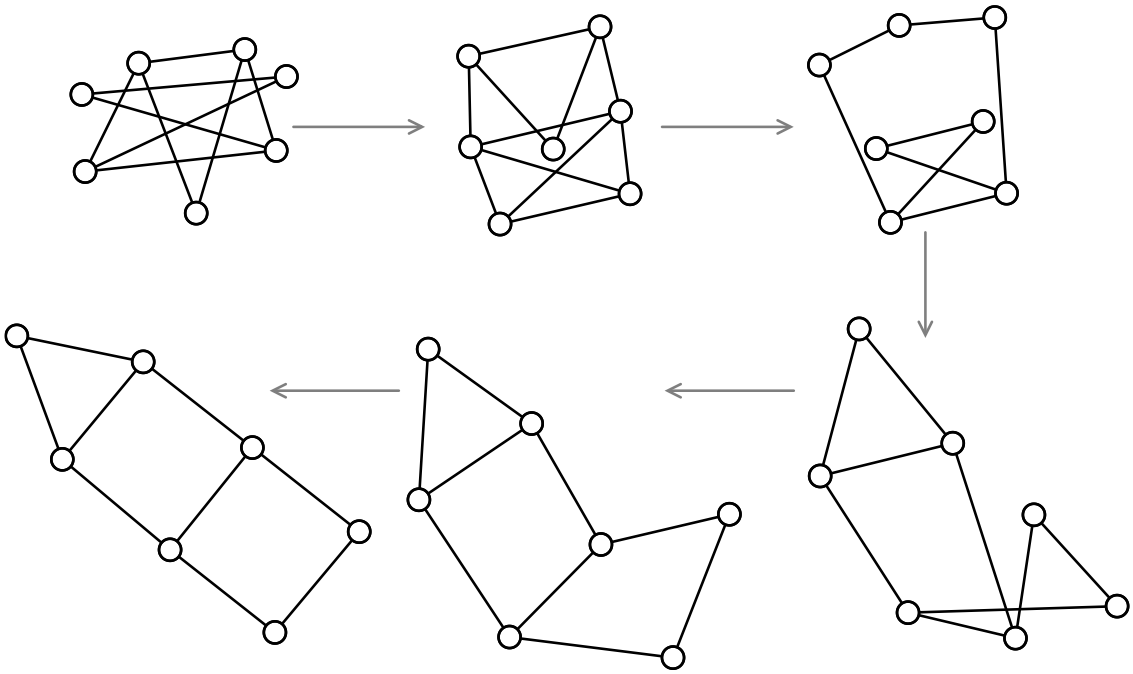

Figure 9: Process of drawing using springs

1.9. Graph Editor

This paper also gives a user interface for graphs on surfaces. There are lots of different programs for graph visualization. But this Graph Editor gives unique 3D visualization for graphs drawn on a sphere or a torus. We can obviously draw these graphs in 2D using polygonal representation however this 3D view can help our imagination and improve our ability to think about those problems in different ways. This software also can be used for explaining relations between 2D representation and 3D representation. Since this program is designed to improve the readability of graphs you can change color of vertices, edges or entire surfaces.

This program has also built in functions for finding nice drawings on torus which boost graph readability.These functions are written as a plugin with a specified interface. So the program can be used to test other implementations of those functions. There is also documented data structure to store graphs drawn on arbitrary surfaces. In this program only sphere and torus are used but this data structure is designed to work with any orientable surface.

2. User guide

2.1. Minimal system requirements

This program requires x86, amd64 or compatible machine with installed Windows XP or higher, .NET framework 4.0 or higher and 3D graphic card. Any integrated 3D card is ok, but with a growing number of vertices we recommend a non-integrated card with 256MiB or more dedicated memory. Program can run with any resolution supported by windows but for comfortable work we recommend 1920 x 1200 or higher. This program was developed for standard desktops with a mouse and keyboard, but a touch screen or tablet should work as well.

2.2. Installation / Uninstallation

The Program can be installed using setup.exe located in root directory on the install CD. This installation checks framework versions and installs newer if needed. This installation process will automatically add a shortcut to the start menu. The Program can be executed by clicking start - > all programs - > Graph Editor -> Graph Editor.

To uninstall the Graph Editor you can use standard Windows mechanism: Start -> control panel -> programs and features -> select graph editor -> right click and chose uninstall.

2.3. Main window

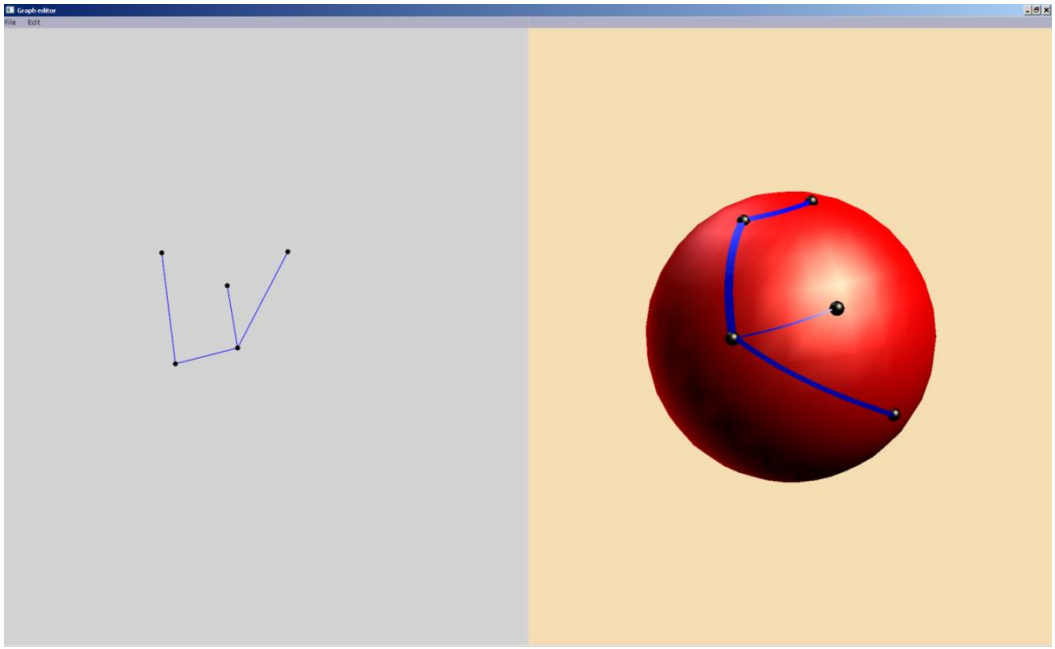

Figure 10: Main window of the Graph Editor

The main window (Figure 10) is split into two parts. On the left side there is a 2D visualization part of the graph. And on the right side there is a 3D visualization. We can adjust the size of these parts by dragging and dropping the "split line" in the middle. On the top of the window we can find a menu bar which allows us to access some basic functions. In the file menu we can find standard file functions like save, load and new. All these functions are implemented using standard windows open or save dialogs.

In Edit menu you can find these actions:

- Undo
  - Returns one step back in editing history
- Redo
  - Restores one action removed by Undo action
- Change surface color
  - Opens a color dialog, where we can select a color for surface
- Change background color
  - Opens a color dialog, where we can select background for 3D visualization
- Set surface

    - You can change the projection surface (You can chose Sphere or Torus)
- Change wire model/normal model
    - In wire mode view all 3D objects are visualized with lines only
    - WARNING: this can significantly slow 3D visualization

2.4. 3D graph control

The 3D view can be controlled using mouse or keyboard. To operate the surface using your mouse, click anywhere in the 3D part and move left, right, up or down. An alternative for that is to use the arrow keys which let you move left, right, up and down. Zoom level can be adjusted with the add and subtract buttons on the numeric keyboard.

2.4.1. Keyboard shortcuts

For better work with our program we have added some basic keyboard shortcuts. We are trying to use shortcuts, which are commonly used in other software. Our shortcuts are listed in following table.

F1	Sets projection surface to sphere
F2	Sets projection surface to torus
F5	Run embedding plugin
Ctrl + O	Open dialog which lets you open saved graph
Ctrl + N	Opens new graph
Ctrl + S	Saves graph
Ctrl + Y	Redo
Ctrl + Z	Undo

Table 1: Keyboard Shortcuts

2.5. Graph editing

The graph can be edited in 2D view only. User interface is designed as easily as possible, so most of the controls are very intuitive. There is a context menu by each component, such as a vertex or a line. Those menus can be used for some basic control of these components. The context menu is opened by right clicking on the corresponding component.

2.5.1. Vertices

Vertices are visualized as small dots in 2D view and as spheres in 3D view. We can add a new vertex using the middle mouse button on the selected position. Vertex can be moved by dragging and dropping using the mouse. While moving a vertex we can see a different cursor (it looks like a hand).

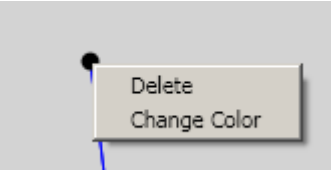

Figure 11: Context menu for vertex

Using the vertex context menu (Figure 11) we can change the color of the vertex. The Color can be picked in a standard color dialog. We can also remove the selected vertex using the delete option.

2.5.2. Edges

Edges are visualized as straight or polygonal lines between two vertices. These lines are blue in default settings. To add a new edge between vertex A and B press and hold the middle mouse button on vertex A and drag to vertex B and release it there.

We can also use a polygonal line instead of a straight line. This can be done using these steps:

1. Choose an edge to convert
2. Open the context menu for that edge
3. Select the add point option

This adds a small gray “sub-point” in the middle of selected line. This small “sub-point” can be moved anywhere. After splitting, each line can be operated separately. These sub-points can be removed identically to vertices.

The 2D torus representation is in fact a polygonal representation of an arbitrary surface. A square turns into a torus by joining its left bounding line segment with the right one and its top segment with the bottom one. And every line can go through this “join” in 2D (see Figure 12, Figure 13). This causes effect on torus only. We can select X or Y join. If you want a line to go round the torus, instead of going

straight, open the context menu of this line and choose “use X(Y) join”. This property is set per each line, so one edge can go through this join multiple times. Using splitting and join we can draw in the Graph editor anything we can draw on paper. We can see the effect on following pictures.

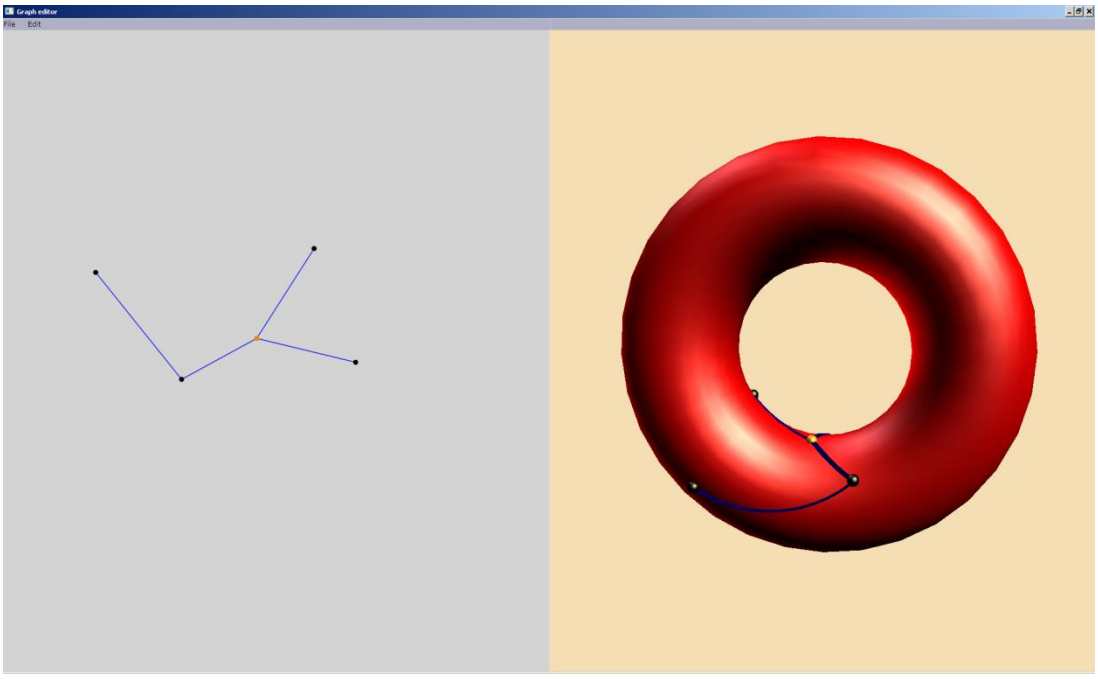

Figure 12: Line without going throw split line

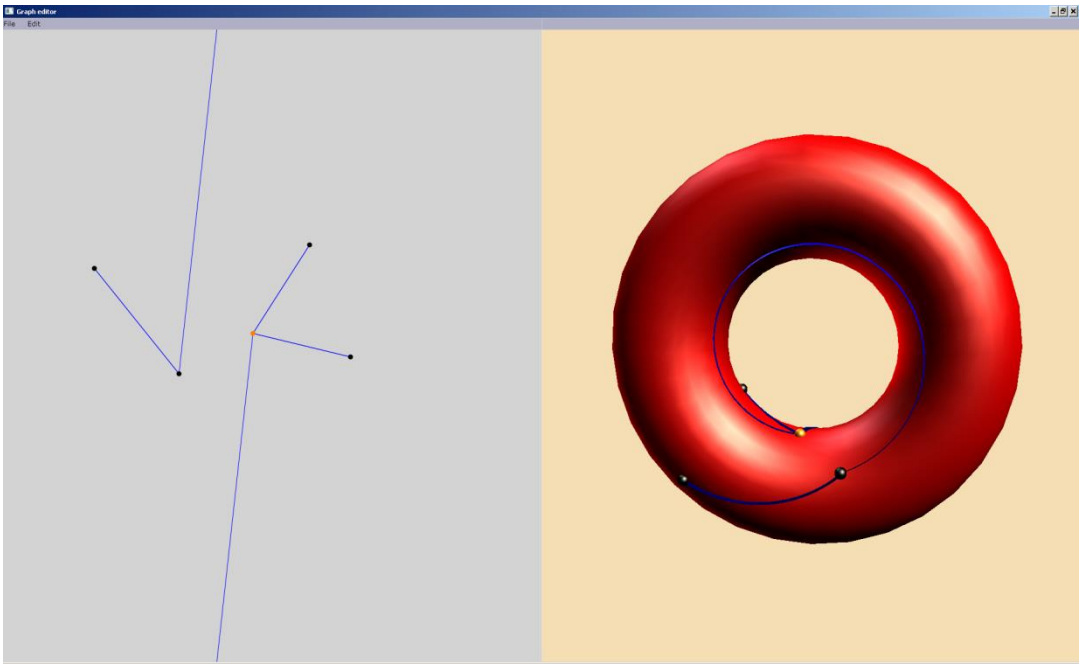

Figure 13: Line going throw split line

There are some additional options in the edge context menu like ‘change color’ or ‘delete edge’. If we are deleting an edge which is composed of some small

lines, this will delete all of them. If we want to delete one of these small lines we need to find the corresponding “sub-point” and delete this point.

3. Implementation

The program was developed in Visual studio 2010 with service pack 1. It is written in C# 4.0 and for 3D visualization we are using Microsoft XNA game studio. Code is written for .Net framework 4.0 client profile.

One of the goals of this program is to provide an interface for future plugins. To reach this goal, solution is split into 4 projects:

- Graph definition
- Graph editor
- Plugin interface
- Plugin

3.1. Graph definition

Classes in this project describe a graph with all parameters like position and color. But there is no code for drawing. This class has no connection to the main project so it can be used with any other graph drawing interface. The main class is called Graph definition. This class holds all vertices, edges and some general information (like surface type). All classes in this project are serializable so they can be used for storing and loading. The entire graph description is done by object relations. So every edge has a pointer to its vertices and all sub-points. It could be done with an adjacency matrix or list of neighbors. These representations are more compact and it can be faster to operate with them. But neither of them allows us to represent multi-graphs. Multi-graphs are not allowed in this program, however the description is universal. Moreover this object representation can boost code readability. In plugins we may need some other representation to run these algorithms faster. Nevertheless we decided that it can be done in these plugins if needed.

3.1.1. Edge

Edge class describes one edge between two vertices. Because polygonal lines are allowed, an edge is constructed from a list of Line2D.

3.1.2. Line2D

Each line stores its start vertex, end vertex and color.

3.1.3. Point

It stores a position and color. Position is stored relatively in interval [0,1] in x and y axes. So all positioning is done relatively to size of the window. This allows us to have universal embedding for different sizes of window. If the point is in the end of an edge, it is called Vertex and has its own inherited class.

3.2. Graph Editor

This is the main project. The entire interface is implemented here including the main window. The main window is created using WPF and XAML. There are two components on main window: canvas for 2D visualization and a XNA hosting component for 3D visualization. WPF was chosen over classic windows forms because of its ability to render its graphic on a graphics card. This means that visualization is fast and smooth. It will also use less CPU. XAML also allows us to split implementation and design as much as possible. The window design is described in a separate .xml file. It allows us to give this separable xml to some specialized form designer. XNA could be maybe replaced with more classical DirectX or some other faster solution. However XNA was chosen because of better abstraction and a nice managed interface. It shows that for such small 3D scenes XNA is fast enough and its readability is a huge benefit for developers.

3.2.1. Graph

This class holds a graph. It stores the definition of a graph using GraphDefinition and caches 3D objects for 3D visualization. 3D objects are cached to speed up the program. Lines are cached as a 3D object unless they change. And there is only one sphere to represent vertices in 3D. This significantly lowers how many times garbage collection is needed. These caches are flushed when we close

the program. Two main methods are draw2D and draw3D. These methods call draw2D or draw 3D respectively on each component.

3.2.2. Drawers

Each graph component from GraphDefinition has its own "Drawer" which implements the IDrawer interface. This interface forces methods to draw2D and draw3D. There is a "PointDrawer", "VertexDrawer" and "LineDrawer". Implementation of drawing in 2D mode is easy and it always uses corresponding WPF component line for drawing edge and ellipse for drawing vertices. However 3D drawing is more complicated. Vertices are visualized as small balls half dipped in surface(sphere or torus). Lines are visualized using tall squared towers. Those towers are curved around the surface. This is all done by the standard 3D matrix technic.

3.3. Embedding algorithm

We are facing the problem of embedding a graph G in the surface. The idea is to start with some subgraph H of G where embedding is obvious and then iteratively add rest of graph into embedding H. When H contains all vertices and edges from G we return H as the result. In each iteration we find all faces in H and we search for components of graph $I(I = G \setminus H)$ and count the number of admissible faces (this component can be embeddable in this face). If all components have zero admissible faces algorithm fails and there is no embedding. In each iteration we take a component B with a lower admissible face count. We find some path from this component B and we add am embedding of this path to the embedding of H.

Detailed pseudo code of presented algorithm can be found in paper "A Faster Algorithm for Torus Embedding"[12]on page 30.

3.3.1. Data Structures for embedding

The most important data structure in this algorithm is the data structure which represents am embedding or a partial embedding. This structure needs to handle any graph. We need an ordering of neighbor vertices. In this chapter we discuss some possible approaches. Our structure needs to have a reasonable memory space complexity. But we also need these functions, with as good time complexity as

possible: add edge, remove edge, get next vertex in neighborhood array, subtract two embeddings, search path, search cycle.

First possible solution can be adjacency matrix. There will be huge memory overhead. This matrix is O (n^2) big. This is optimal for graphs with big density (m is in O (n^2)). But emendable graphs have small density (m is inO (n)) so it would be a waste of memory. And there is no possibility to implement the ordering of vertices in this structure. A plus side of this representation is that subtracting will be easy and can be done for each cell separately and in parallel (this may be a good structure for some parallel algorithm).

Another way to go is an adjacency list – one array indexed with vertex number that contains indices to a second array where all adjacent vertices can be found. This will be optimal in memory complexity for the original graph, but it will have overhead for small subgraphs (components). And removing edges or vertices will take O (n) time (we need to shift rest of data).

So we decided to use a dictionary where the key is the vertex's name represented as a number and the value is an array of adjacency vertices. We may want a set instead of an array (it will be automatically resistant to duplicities), but we need an ordering of these adjacency vertices. This is optimal structure for memory usage. For a graph with one vertex there is just one record. In C# this dictionary is represented using a hash table. So there is a small hash table multiplicative constant, but since we are using a number as key it should not be that bad.

3.3.2. Other used structures

We also need to define is a representation for bridge with respect to (for this paragraph just a bridge). We define a class for that. This structure always contains an embedding (defined in previous paragraph), a set of attachment vertices, number of admissible faces and a pointer to one of the admissible faces. There is also a comparer defined on this class, this allows us to sort bridges by number of admissible faces. Choosing the bridge with a lower number of admissible faces is critical for the correctness of the algorithm.

All other objects are represented using some built-in structure. For example a face is just an array of bounding vertices, and edges are represented as a standard pair.

3.3.3. Sphere Embedding implementation

Since the torus embedding algorithm is based on a sphere. We discuss the sphere algorithm first and then we explain the extension to Torus.

In this section we define three graphs. G is the original graph; H is already a valid embedding and graph I is G minus H. G is throughout the entire algorithm immutable and read only. At the beginning we find a Cycle C in G. Finding a cycle is done by DFS. Some embedding C will be the initial H. I is set to G minus H.

Than we find all bridges of G with respect of H. We count the number of admissible faces. Since each bridge stores its attached vertices, finding out if some face is admissible for this bridge is just checking if every attached vertex is occurring in this face. If there is some bridge with no admissible face, there will be no embedding and we can end the algorithm. Otherwise there are some embeddable bridges. We need to find the one with lowest possible admissible faces. Then we find some path, in this bridge, which starts and ends in face-vertices. This path can be trivially embedded into H. And we can continue to next iteration. I can be computed from definition once again, but this would be a waste of time, so we just delete this path from I.

3.3.4. Torus embedding implementation

Any embedding of a graph in a plane is also valid on torus, so we try our planar embedding algorithm first.

For torus we need to begin with something other than a cycle. A good approach is to use K_5 or $K_{3,3}$. There is also a slight problem with embedding a path into the selected face. Some faces can have repeating vertices (a vertex can occur in a face on a torus at most twice). This means we need to try both possibilities for both end vertices of the path. This leads to exponential time, because we need to search through all possibilities. We are using a standard heuristic for search problems – fail first heuristic. So it is a good idea to try bridges with only one possibility first. This can reduce the search tree significantly. The rest of the algorithm is the same.

3.3.5. Searching for planar obstructions

From Kuratowski's theorem we know that a graph is embeddable in a plane if and only if there is no sub-graph homeomorphic to K_5 or $K_{3,3}$. So we are removing edges from Graph G until we get some planar graph. Than we put last edge back and we get some non-planar minimal non-planar graph of G. This graph must be homeomorphic to K_5 or $K_{3,3}$.

3.3.6. Possible embedding of planar obstructions

There are more possibilities how to embed these graphs into torus. And generating of these embedding will be hard to implement. So we put all these embedding in code as constants.

3.3.7. Parallelization

In these days almost anyone has a multi core machine. So parallelization can help a lot. In this program we are using standard .NET mechanisms for parallelization. The Entire computation is divided into tasks. These tasks are handled over to a build-in tread pool. Each possible embedding of K_5 or $K_{3,3}$ is tried in another task. If one is successful we interrupt all of the others tasks and yield this first result.

4. Results

4.1. Running time

The algorithm was tested on randomly generated graphs with number of vertices in range from 20 to 100. Each graph was run with one, two or four threads. Results are divided into cases where there is a embedding into a torus and where there is not.

#vertices	#edges	Avg. 1 core	Avg. 2 cores	Avg. 4 cores
20				
40				
60				
80				

100				

Table 2: Average time in milliseconds for toroidal graphs

#vertices	#edges	Avg. 1 core	Avg. 2 cores	Avg. 4 cores
20				
40				
60				
80				
100				

Table 3: Average time in milliseconds for non-toroidal graphs

4.2. Example embedding

TBA

5. Conclusion

5.1. Algorithm result

As we can see the results are good for graphs with about 100 vertices. Test shows that parallelization works well and it is a good way to speed up this algorithm up. It was tested on an 8 core machine and the time was almost 8-times better than with a single core. So the parallelization is optimal for a constant number of processors. With a number of cores which is asymptotically same as the number of vertices this parallelization is useless.

5.2. How can the algorithm be improved

This algorithm was implemented in a just in time compiled language and there were no constant optimizations made. So with some care about using a cache, some data structure compression or even with SIMD instructions I expect the running time would lower about ten times. More speeding up could be done with adding some more heuristics. We may try to start with the spring algorithm to get a sketch of embedding and then improve it with our algorithm. With a huge number of testing graphs we can get some statistic data to determine which option will lead to a result faster. For example some of possible drawing of K_5 can be more perspective

than other. So we can try a better possibility of the drawing first. We can also statistically determine which option of path embedding we should use first. Or even a cleverer way of finding a path in a bridge can be done. For example we can try to search for a longest path in a bridge. But all these optimizations will help in the average case only. According to the number of possible embeddings it is also better to search for $K_{3,3}$then for K_5. But our method to search for them will yield one of them. So searching for $K_{3,3}$ and only if there is no $K_{3,3}$ searching for K_5 could help a lot.

There is also no reason why we need to begin with K_5 and $K_{3,3}$. We need to start with some small graph with finite embedding possibilities – like a cycle in plane. Forbidden minors for surfaces with a smaller genus is an elegant solution for the torus. This cannot be extended for other surfaces (we do not know these forbidden minors). So we may search for some other starting point which can be more universal.

5.3. Future research

It would be great to find some polynomial algorithm for each fixed arbitrary surface which can be used in the real word. The torus algorithm is a good place to start. I hope that this ideal algorithm could be based on the presented algorithm. In a plane it works in quadratic time so we can hope that somehow we can do it in polynomial time on a torus too. In a plane it is correct and quadratic because of choosing the right bridge first. We know that this is not enough on a torus. But there is good reason to think that some clever order of chosen bridges will help. A good start may be to go through the plane algorithm and try to generalize these conditions for an arbitrary surface or just for the torus.

Another interesting research will be to finally determine entire list of forbidden subgraphs for each surface. Previous proofs about forbidden minors for plane or projective plane were made by some case analytic of graph drawing. But we know that there are more than 300 000 obstructions for torus embedding, so no such approach will work. I think this is a nice challenge to find some universal formula for constructing such a list and then prove it is correct. This list will then give an excellent benchmark test input for all embedding algorithms.

Bibliography

[1] J. Boyer and W. Myrvold, 2004 On the cutting edge: Simplified O(n) planarity by edge addition. Journal of Graph Algorithms and Applications

[2] J. Combin Graph minors. XIII. The disjoint paths problem, Theory, EMBEDDING GRAPHS IN AN ARBITRARY SURFACE 21 Ser. B 63 (1995) 65–110.

[3] Reinhard Diestel, 2010, Graph Theory, 4th edition , Heidelberg: Springer-Verlag, ISBN 978-3-642-14278-9

[4] H. Djidjev, J. H. Reif, 1991, An efficient algorithm for the genus problem with explicit construction of forbidden subgraphs, 23rd Annual ACM Symposium on Theory of Computing, New Orleans, LA

[5] J. R. Fiedler, J. P. Huneke, R. B. Richter, N. Robertson, s, J. Graph Theory 20 (1995) 297–308.

[6] I. S. Filotti, G. L. Miller, J. Reif, 1979, On determining the genus of a graph in O(vO(g)) steps, in "Proc. 11th Ann. ACM STOC," Atlanta, Georgia pp. 27–37.

[7] Martin Juvan and Bojan Mohar, An algorithm for embedding graphs in the torus, Department of Mathematics, University of Ljubljana

[8] Martin Juvan, Joze Marincek, Bojan Mohar, Embedding a graph into the torus in linear time, Department of Mathematics, University of Ljubljana

[9] William Kocay, Daniel Neilson, Ryan Szypowski, 2001, Drawing Graphs on the Torus. Ars Comb. 59

[10] Bojan Mohar, 1999, A Linear Time Algorithm for Embedding Graphs in an Arbitrary Surface, SIAM J. Discrete Math. 12, 6-26.

[11] E. Neufeld, 1993, Practical toroidality testing. Master's thesis, Department of Computer Science, University of Victoria, Victoria.

[12] Jennifer Roselynn Woodcock, 2004, A Faster Algorithm for Torus Embedding. B.Sc. University of Victoria

List of figures