Research Project: Number of Perfect Matchings in Planar Graphs

Emil Kastholm Fischer
15137 - emfi@itu.dk

December 2021

1 Introduction

This report will cover a known algorithm for counting *perfect matchings* in a planar, undirected graph. The algorithm is based on the trick that linear algebra can be applied on the adjacency matrix of a planar graph, given the graph is *adjusted* in a certain way. This adjustment is, more specifically, to *orient* the graph in a way that is called *Pfaffian*. To Pfaffian orient the graph proves to be the main part of this algorithm and is therefore the part that this report primarily focuses on. However, first a couple of expressions and techniques needs to be defined to create a basis for the algorithm.

Given an undirected graph $G = (V, E)$, a matching in G is a subset of edges $M \subseteq E$ such that every vertex in V is incident to at most one edge in M. Such a matching is said to be *perfect* if every vertex in V is incident to exactly one edge in M.

In the 1960s, Fisher, Kasteleyn and Temperley derived a method for counting perfect matchings for a particular class of graphs. This is called the FKT Algorithm, and it works on all graphs which are "Pfaffian orientable" — this graph class contains the set of all planar graphs as a proper subset.

In general, orienting an undirected graph, simply means to assign a direction to each edge of the graph. This results in a directed graph with no self-loops.

Let $\overrightarrow{G}$ be an orientation of G and C be an even-length cycle in G. We say that C is *oddly oriented* by $\overrightarrow{G}$ if, when traversing C in any direction, the number of oriented edges *not* going in the chosen traversing direction is odd. Note that the choice of direction (clockwise or counter-clockwise) is not important due to the fact that C is of even length, meaning that if there is an odd amount of clockwise edges, then there is also an odd amount of counter-clockwise edges. Now, an orientation $\overrightarrow{G}$ of G is Pfaffian if, for any two perfect matchings M, M'

in G, then every cycle in $M \cup M'$ is oddly oriented by $\overrightarrow{G}$. Note that a union of any two perfect matchings in a graph, always consists of a collection of single edges and even cycles [2, Fact 1.9]. For a Pfaffian oriented graph $\overrightarrow{G}$, we now need to adjust its adjacency matrix A to become a *skew adjacency matrix* A_s. This will change the sign of the entries so we can use the matrix' determinant to calculate the number of perfect matchings in the graph. For computing A_s, we iterate over all entries a_{ij} in A and apply

$$a_{ij} = \begin{cases} +1 & \text{if } (i,j) \in E(\overrightarrow{G}) \\ -1 & \text{if } (j,i) \in E(\overrightarrow{G}) \\ 0 & \text{otherwise} \end{cases}$$

The number of perfect matchings in G can then be calculated by

$$\sqrt{det(A_s)},$$

where $det(A_s)$ denotes the determinant of A_s. The proof of this is not covered by this project, but is given in [2, page 6-7, *Proof of Theorem 1.11*].

2 Implementation

The previous section implied a polynomial-time algorithm when the input graph is given with a Pfaffian orientation. In general, it is NP-hard to find a Pfaffian orientation for a graph, or even to check that a given orientation is Pfaffian [4, page 180]. We focus on the special case of planar graphs, for which a polynomial-time algorithm, for finding a Pfaffian orientation, is known. Every planar graph has at least one Pfaffian orientation [2, Theorem 1.14], and our algorithm constructs such an orientation. According to [2, Lemma 1.13], any planar oriented graph is Pfaffian oriented provided that every face has an odd amount of edges oriented clockwise. This means that we only need to traverse each face and orient an odd amount of edges clockwise and the rest of the edges on that face counter-clockwise. The FKT algorithm uses exactly this fact combined with the method for calculating the number of perfect matches covered in the previous section.

The FKT algorithm also uses the fact that every planar embedded graph G, has a dual graph. This graph consists of a vertex set where there is placed one vertex in each face of G (including the outer infinite face). Meaning every vertex in the dual graph, corresponds to exactly one face in G. Every pair of vertices in the dual graph is then connected by an edge, if their corresponding faces in G, are adjacent.

Given a planar graph G, the FKT algorithm proceeds as follows:

1. Compute a planar embedding of G.

2. Compute a spanning tree T_1 of G.
3. Arbitrarily orient each edge in T_1.
4. Compute a graph T_2 (with no edges yet) with the same vertex set as the dual graph of G.
5. Create an edge in T_2 between two vertices if their corresponding faces in G share an edge that is not in T_1 (note T_2 is a tree).
6. For each leaf l in T_2 (unless l is the root):
   (a) Orient the edge in G that l's edge intersects.
   (b) Remove l from T_2.
7. Return the square root of the determinant of the Pfaffian skew adjacency matrix.

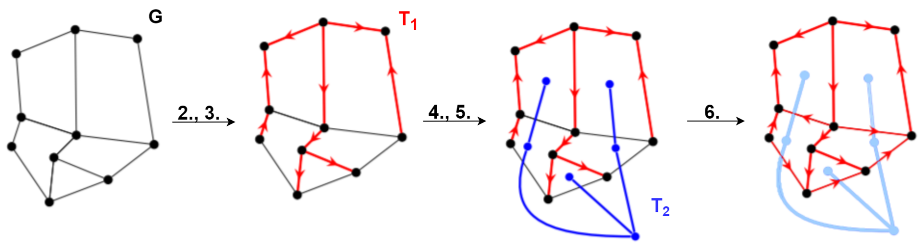

Figure 1: Visualization of the FKT algorithm [1, slide 70 - 81]

This concludes the needed theory for constructing the concrete implementation - meaning the rest of this section will focus solely on the implementation of the FKT algorithm.

2.1 Embedding and Detecting Faces

The very first step of implementing the algorithm is to embed the input-graph on the plane. Here I have simply used a library called *Networkx* in Python. The input-graph is represented as an adjacency matrix; if the graph is planar, the framework embeds it such that edges are drawn as straight lines. Due note that such an embedding always exists for a planar graph [6, Fáry's theorem].

Detecting the faces of the embedding needed a bit more thought. This algorithm relies on the fact that every vertex has a list of all its incident edges, sorted clockwise. Then first every edge is doubled into pairs of directed edges, where each edge point in opposite directions. This means an undirected edge between two vertices u, v would become two directed edges - one from u to v and one from v to u. The algorithm then iterates over each vertex v_{origin} in the graph, while keeping an initially empty set of edges that is marked as used:

1. If all edges incident with v_{origin} are used, then skip this iteration.
2. Define two pointers to two vertices u, v respectively.
3. Set u to v_{origin} and v to any adjacent vertex of v_{origin} along a non-used edge.
4. Define a set S that will ultimately contain the edges of the face currently being traversed.
5. Loop until u becomes v_{origin} again
    (a) Define a pointer to the edge e going from u to v.
    (b) Define a pointer to the edge e' going from v to u.
    (c) Add e to S and mark it as used.
    (d) In v's list of outgoing edges, find the last non-used predecessor of e' and set e to become this edge.
    (e) Define a pointer to a vertex v' to be the vertex that e points to.
    (f) Set u to v and v to v'.
6. return S.

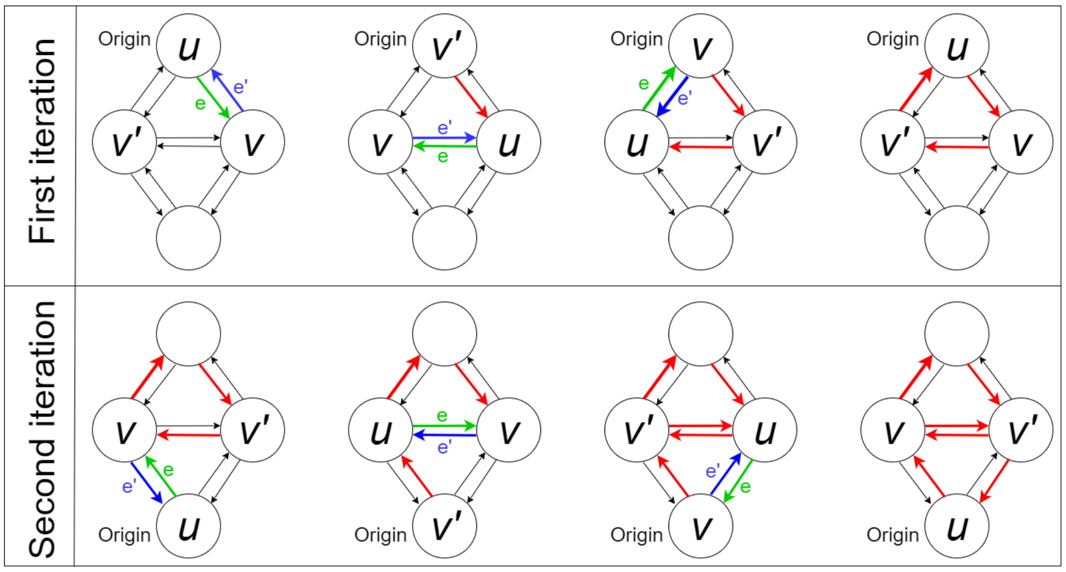

Figure 2: Visualization for algorithm for detecting faces

In Figure 2 is the above explained algorithm, for detecting faces of a planar graph, visualized. This algorithm is inspired from the insights given at [3, page 36 - 37]. Note that in the visualization are the red edges the ones that has been added to S (and therefore also marked as used). This successfully detects the two faces, of the given example graph, in clockwise order. Due note that this would have a final iteration, which is not visualized, where the exterior of the graph would be detected in a counter-clockwise order. This is because the outer infinite face “sees” the mirrored version of the graph. Meaning this face is also detected correctly even though it does not seem to have immediate importance for the FKT algorithm.

In the given example in Figure 2, all edges are part of an interior face of the input graph. Another way to put this is that all edges are adjacent to exactly two faces (including the outer infinite face). This is not always true for planar graphs.

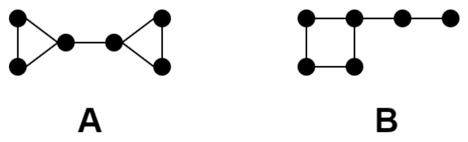

Figure 3: Examples of problematic graphs for the algorithm

In Figure 3 are two “problematic” graphs **A** and **B**. Each of these graphs has at least one edge that is not adjacent to two faces. This raises a problem whether the faces' edges would still be listed in clockwise order. However, it turns out that **A** is in fact not problematic and **B** can be easily handled by a pre-processing trick. The single edge in **A**, “bridging” the two interior faces, is only part of the outer infinite face and will always only be that, since that, in the iteration when detecting edges for the outer infinite face, the algorithm will circle one face, cross the bridging edge, circle the other face, cross the bridging edge again, and return to its *origin* - by this using this single edge both ways and ensuring it will not be used again.

For graphs like **B**, before the start of detecting faces, all vertices with a degree of one is removed together with its neighbor. This process is repeated until no vertices has a degree of one. Note that this can safely be done since it will not change the number of perfect matchings due to the fact that an edge, incident to a vertex with a degree of one, has to be included in the match for the match to be perfect. For the concrete example for **B**, the process would happen once - removing the two vertices to the far right.

2.2 Implementing the FKT Algorithm

Recall that a tree T_2 with the same vertex set as the dual graph needs to be created (see Figure 1) and note that all these vertices are placed in a face in the input-graph (including the outer infinite face). This means that, using the algorithm from the previous section for detecting faces, we can construct the dual graph of the input graph and by that construct T_2. Now we say that an edge e between vertices u and v is *coherent* to a face if, when traversing the face in the clockwise direction, the traversing order goes from u to v via e - this can be pictured as two co-oriented edges where $e_{u,v}$ is coherent to one face, while $e_{v,u}$ is coherent to another face. With this definition in mind, we construct a dictionary from an edge to a face such that any edge points to its coherent face. Then while running the algorithm for detecting faces, when traversing $e_{u,v}$ on a face f, we set $e_{v,u}$'s coherent face to be adjacent to f. Resulting in a structure where we know each adjacent face of each face.

Now everything is set up for efficiently implementing the FKT algorithm. Creating the spanning tree T_1 of the input-graph G is trivial. Then we identify all edges that are in G but not in T_1 — let us denote this set of edges as S. We can then iterate through each face f of G using *breadth first search.* While iterating we check whether each adjacent face f' of f shares an edge with f that are in S. If yes, an edge is added between f and f' corresponding vertices in T_2. This successfully constructs T_2. Finally we can - by starting at the leafs of T_2 and using breadth first search again - iterate through T_2 and Pfaffian orient G while making G's adjacency matrix, skew-symmetric. Then return the square-root of this matrix' determinant.

3 Analysis

Assuming the input consists of an $n \times n$ adjacency matrix of a planar graph, an embedding can be computed in $O(n)$ time [5]. Note that for this I have used Networkx, where the time complexity is not included in the documentation.

The algorithm for detecting all faces clockwise, which was covered in Section 2.1, essentially doubles all the edges and iterates over each of them at most once. This results in a running time given by $2E \cdot O(1) = O(E)$, where E is the amount of edges.

For creating the spanning tree T_1 of the input-graph, I have used Networkx again. This library uses the general method for computing minimum spanning trees for (weighted) graphs. This means it uses a priority queue, to iterate through the vertices of the graph, resulting in a linearithmic running time $O(n \cdot log(n))$. This can easily be implemented as a linear time algorithm $O(n)$, since this has nothing to do with weighted graphs. The algorithm for this would essentially loop through the graph (using BFS or DFS) and collect all edges that does not create a cycle.

When creating the tree T_2 from Section 2, we need to note that this is a subgraph of the dual graph. Since we have already constructed the dual graph (see Section 2.2), all we need to do is to delete all edges in the dual graph that intersects an edge in T_1. Resulting in a linear running time $O(n)$ where n is the size of the dual graph.

In the last step for making the input graph Pfaffian (Step 6 in Section 2), we need to iterate over every edge on every face. Because an edge is at most adjacent to two faces, this has a running time of $2|E| = O(|E|)$.

The last step of the algorithm (Step 7 in Section 2) is to calculate the number of perfect matchings from the skew adjacency matrix. The running time for this is bounded by the polynomial time for calculating the determinant. The naive method for determinant calculation on a $n \times n$ matrix is $O(n^3)$. This can however be reduced to $O(n^\omega)$, where $2 < \omega < 3$, via matrix multiplication. For matrices with a planar structure, this can be optimized even further. This however is not covered in this report, as this could be a whole project by it self.

4 Conclusion

A further implementation of the FKT algorithm could be constructed, where the focus lies in improving each step as much as possible.

As shown in the previous section, every step up until calculating the determinant, is linear. However, because we only operate on planar graphs, certain techniques can be applied to improve the running time for calculating the determinant. Implementing this could be part of a larger thesis project.

The implementation I have done can be seen as a prototype. In this prototype, I have used the Networkx library. Since this library is a black box, and no complexities are included in the documentation, I would have to implement these parts myself, for the sake of the argumentation that I have in fact created a optimal implementation within known algorithms.

Further implementations could include implementing a web service, where a user could input an adjacency matrix and the server would output the number of perfect matchings. Another implementation could be to visualize the algorithm as an animation when it runs.

References

[1] Presentation by Montanaro A, Counting perfect matchings in planar graphs, University of Bristol.
`https://people.maths.bris.ac.uk/~csxam/presentations/matchings.pdf`

[2] Jerrum M, Counting and Markov Chains, Carnegie Mellon University.
`https://www.math.cmu.edu/~af1p/Teaching/MCC17/Papers/JerrumBook`

[3] Berg M, Computational Geometry - Algorithms and Applications, 3rd Edition, Springer.
`https://people.inf.elte.hu/fekete/algoritmusok_msc/terinfo_geom/konyvek/Computational%20Geometry%20-%20Algorithms%20and%20Applications,%203rd%20Ed.pdf`

[4] Vazirani V, Pfaffian Orientations, 0-1 Permanents, and Even Cycles in Directed Graphs, Elseviers Science Publishers B.V.
https://reader.elsevier.com/reader/sd/pii/0166218X8990053X

[5] Ghaffari M, Distributed Algorithms for Planar Networks I: Planar Embedding, MIT & CMU
`https://groups.csail.mit.edu/tds/papers/Ghaffari/podc16-planar1.pdf`

[6] Fáry I, On straight line representation of planar graphs.
`http://acta.bibl.u-szeged.hu/13574/1/math_011_fasc_004_229-233.pdf`