PI Project

Minimum Partition into Plane Subgraphs

Abstract :

This paper presents an algorithm trying to partition an input graph with coordinates into smaller graphs whose edges do not intersect. This problem turns out to be NP-complete.
The Method we use does not return a perfect partitioning but does run in $O(m^2)$ time, m being the number of edges.

Table of contents :

IGGIDR Amine, BOUIGEON Hugo

I. Algorithmic solution

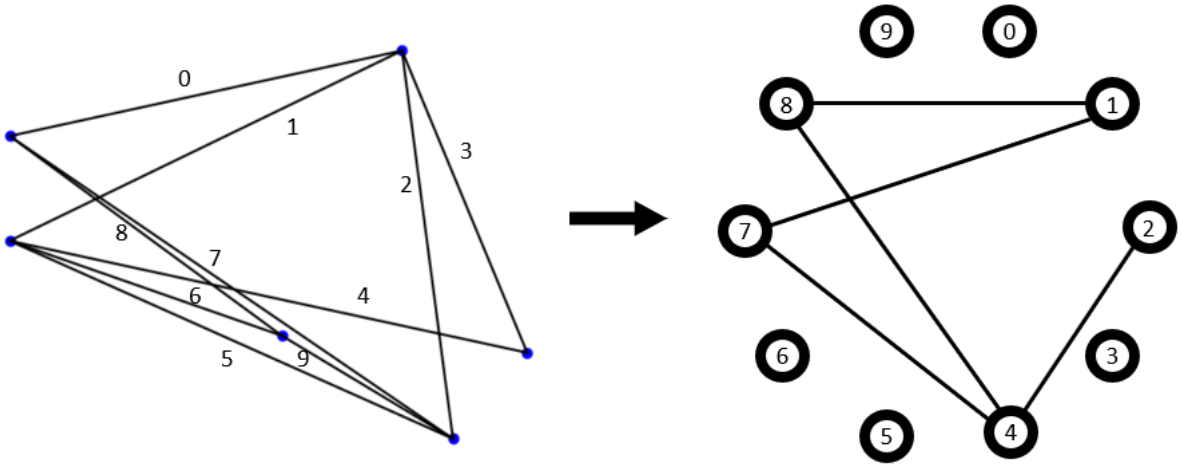
1. Description

The algorithm we made can be divided into three main steps :

1. Converting the graph into a more useful one
2. Finding a good coloration for the new graph
3. Interpreting that information back onto our original graph

Step 1:

We preprocess the input data. Given a graph G = (V,E) as input, we **transform this graph into a new non-oriented graph G'=(V',E')**, where **the vertices of V' represent the edges of the initial graph: V' = E. Vertices in the new graph are then connected to one another if their representing vertices in G are intersecting.** See an example of this process below:



Step 1: Conversion into a new graph, were edges become vertices and intersection become edges

Once this process is done, the problem to solve becomes finding a minimal partition of the new graphs vertices, such that any two vertices adjacent to one another cannot be in the same group of vertices.

In other words : we have to find a k-coloration of the new graph such that k is as small as possible.
This is an already well known problem, notorious for being NP-complete.

Step 2:

Once our graph has been converted, we apply a greedy algorithm to find a proper vertex coloring. The algorithm works as follows :

a) Check if the graph is bipartite (in linear time)

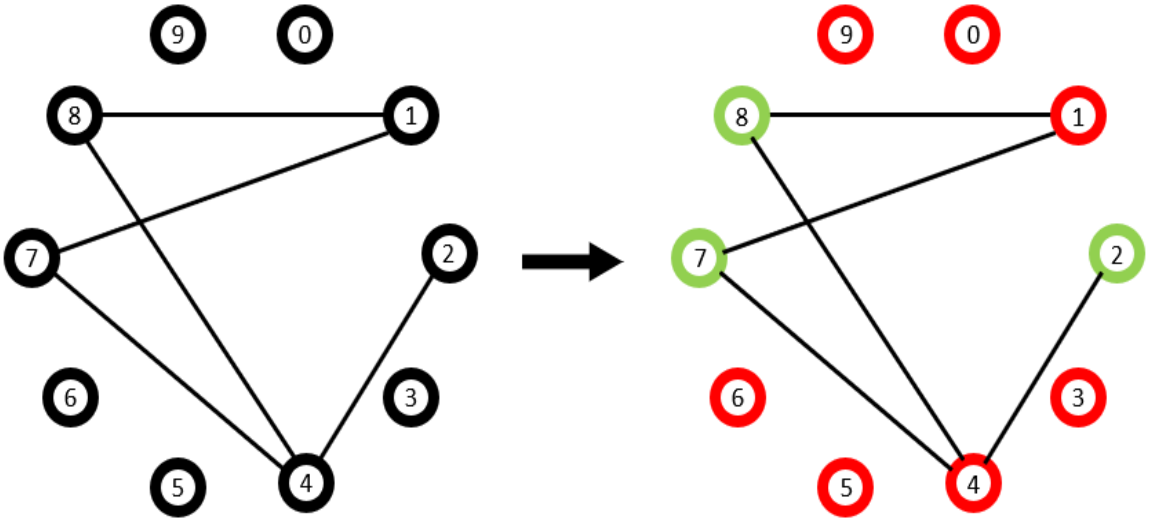
b) Sort the vertices by decreasing degree (usually results in smaller partitions)

c) Color first vertex with first color

d) Do as follow for the remaining vertices

   i) Color the currently picked vertex with the lowest color (we attribute to each color their apparition number) not already used by the vertices adjacent to it.

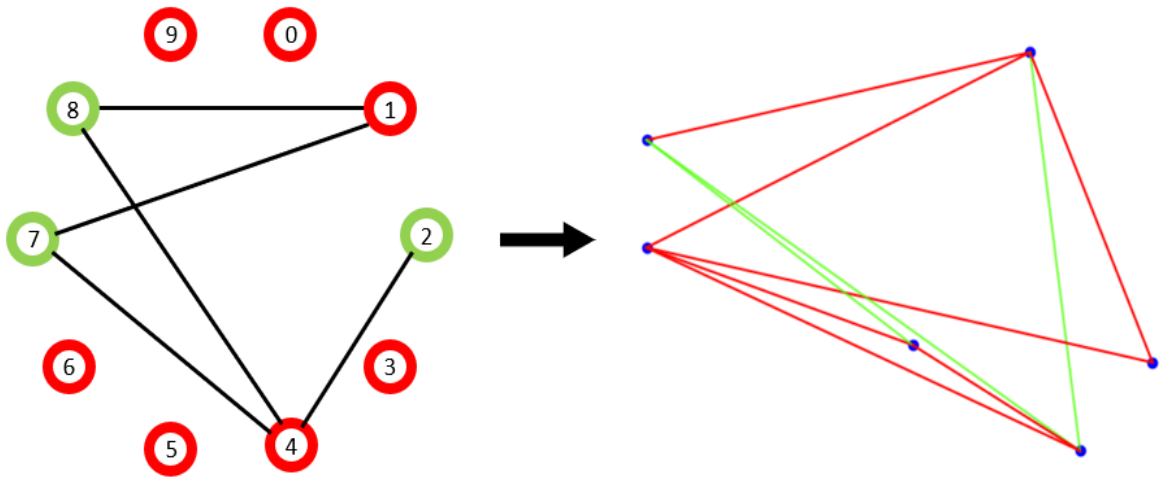
   ii) If all previously used colors appear on the adjacent vertices, assign a new color to it.



Step 2: Application of the partitioning algorithm

Step 3:

The output can then be converted back into the original graph and gives us a partition into plane subgraphs of the input graph.



Step 3 : reconversion to original graph

2. Implementation

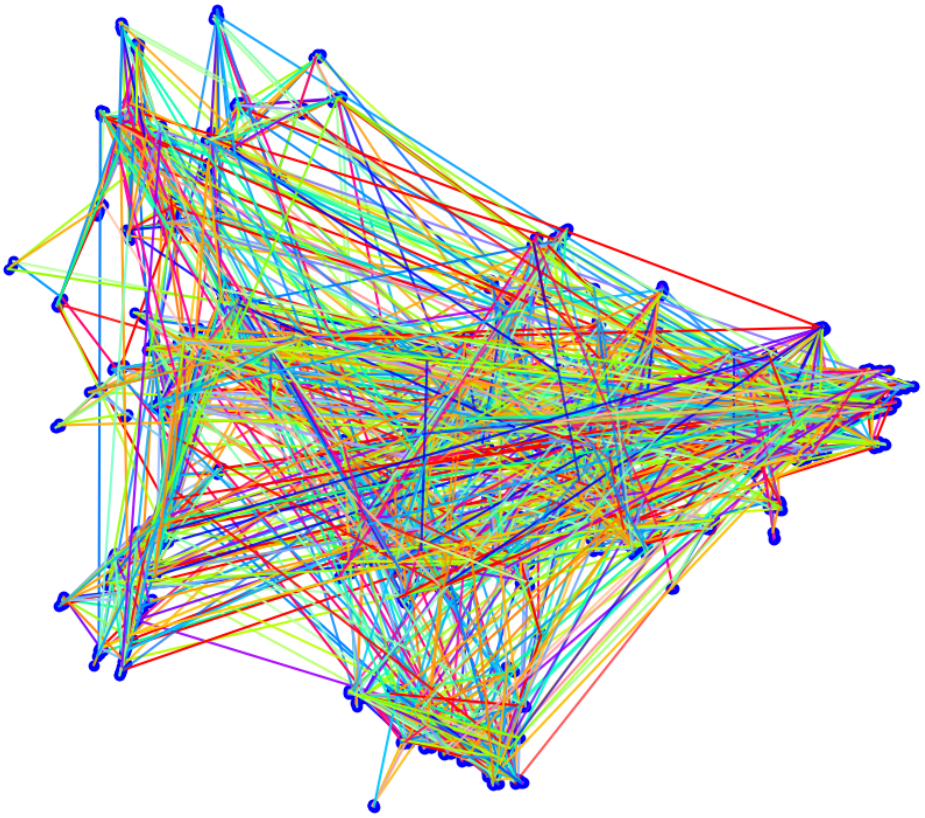
The time consuming part in the step 1 of our algorithm is **finding all intersections** of the input graphs edges. Finding if two edges intersect can be done in constant time.
For that we had to be wary of some exceptions such as two lines being collinear or, sharing an edge, or overlapping.
Finding all intersections can therefore be done in **quadratic time $O(m^2)$** (where m is the number of edges) by checking for any two edges if they intersect.

The resulting graph G' uses an adjacency-list structure of worst case **spatial complexity $O(m^2)$** (m is the initial number of edges so it is the number of vertices of the new graph).

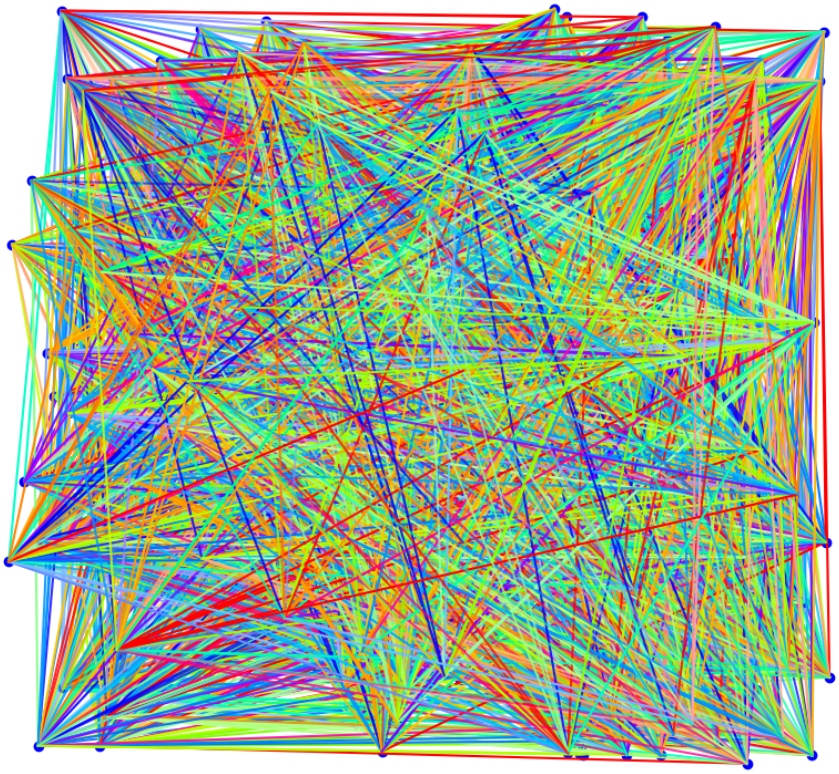
We then proceed to a breadth-first search (BFS) in order to **check if the graph is bipartite**, because bipartite graphs are 2-colorable. This runs in linear time O(m) and does not lengthen the runtime of our algorithm significantly.

After that, we sort the vertices in order of decreasing degree (linear complexity). We then color the vertices one by one giving us a **worst case complexity of O(d(G')m)** where d(G') is the maximum degree of G's vertices.
Since d(G') can be equal to m-1, we thus have a complexity of **$O(m^2)$** for this step.
Our spatial and time complexity are both in **$O(m^2)$.**

II. Experimental results



instance with 2518 edges (76 colors)

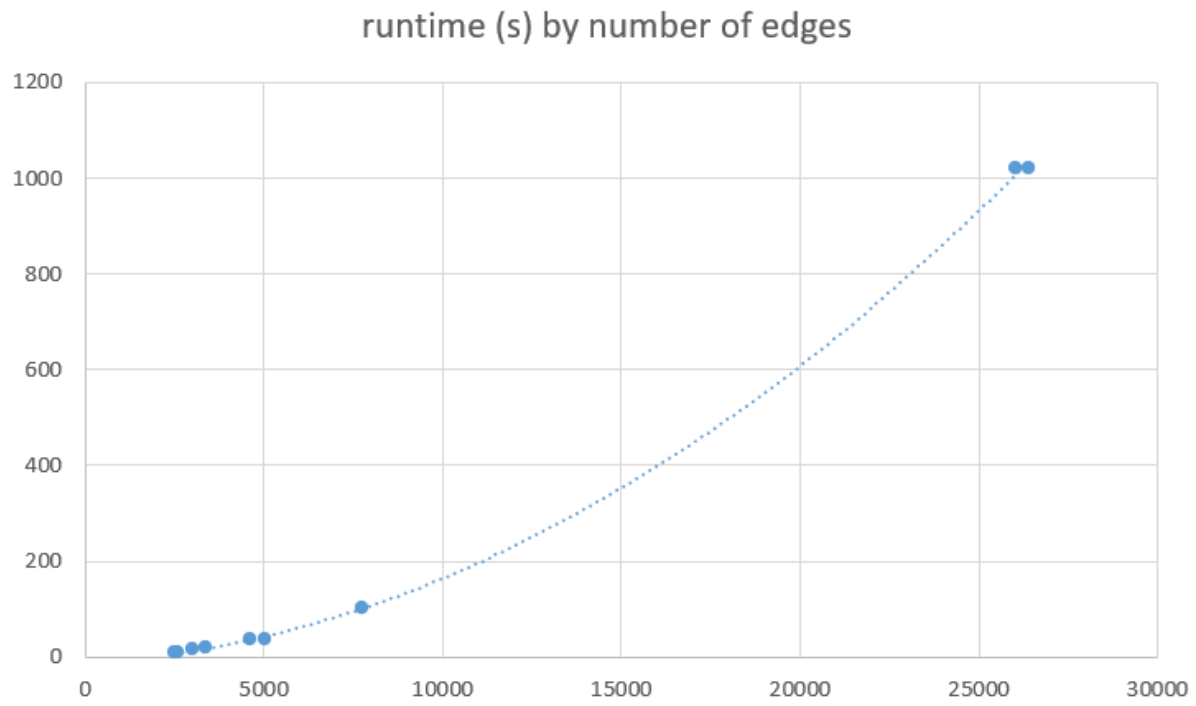


instance with 4641 edges (100 colors)

Instance	3382	4641	5013	2615	7730	3020	2518	26025	26405
partition cardinality	105	100	68	51	131	161	76	316	120

Do note that the special complexity of our algorithm did not permit us to run it on the instance with 30017 edges that was given (resulting in a memory error).

III. Runtime performances



The algorithm runs in quadratic time complexity as proved in the analysis.