# Triangulating polygons

Anton van Niekerk

December 10, 2017

**Abstract**

I describe a program designed to colour the vertices such that a target number of triangles in the graph are neutrally coloured. This program is then applied to a particular problem.

## 1 Summary of problem and solution

### 1.1 Problem

The problem is to colour the uncoloured vertices in figure 1 so that there are exactly 2 neutral triangles in the coloured graph.

### 1.2 Algorithm

There are 11 uncoloured vertices, each being able to be coloured 3 different colours. That means that there are $11^3 = 1331$ possible colourings of the graph. There are also 31 different triangles in the graph, meaning that a brute force algorithm that checks each triangle for neutrality in every possible colouring of the graph must check

$$31 \times 11^3 = 41216 \tag{1}$$

different triangles for neutrality if no solution exists.

The algorithm I came up with for colouring the graph is instead to colour the vertices one by one in the sequence defined by the numbering in the figure as follows (see also comments in the code itself for where each step occurs):

1. The current vertex is assigned a colour (starting with the first uncoloured vertex) from the list {red, blue, yellow}.

2. To reduce the number of triangles to be checked for neutrality, only the triangles with the current vertex as a corner are checked for neutrality, with the total new neutral triangles added to the running total.

3. If the total number of neutral triangles don't exceed the target (2 in this case), then the next vertex will be coloured.

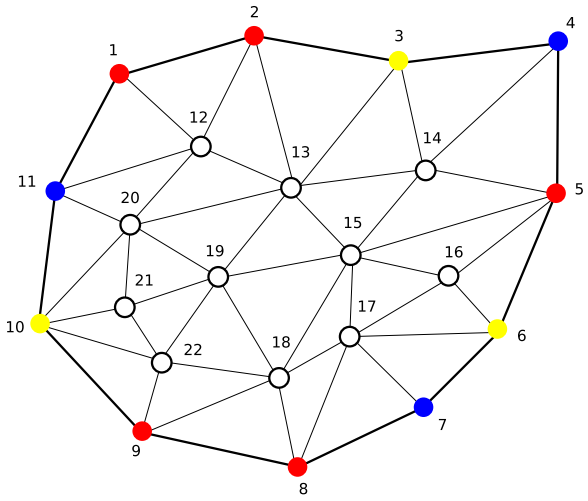


Figure 1: The graph for the coding problem, with the vertices numbered as implemented in Java.

4. If the target number of neutral triangles is exceeded, then the current vertex will be assigned the next colour from the list.

5. If all colours are used up for the present vertex without achieving the target, then the current vertex will be made colourless, and the previous vertex will be assigned the next colour from the list.

6. If the final vertex is assigned a colour and the target is achieved, we break out of this algorithm and leave the graph coloured so that it solves the problem.

7. If all colours in the list are attempted for the first vertex without achieving the target, it means that all possible colourings for the graph have been attempted, and there is no solution. We therefore return the graph uncoloured and return an error code.

## 1.3 Algorithm efficiency

Although I came up with this algorithm on my own, it was inspired at least in part by the tree-based searching algorithms as set out in [1]. In fact, this algorithm can be viewed as following a path down a decision tree, where each branch represents the decision of whether to colour a particular vertex a certain colour. The leaves of the tree then represent a particular colouring of the graph. If a node is reached, and the total number of neutral triangles exceed the target number, or we reach a leaf and the target is not met, then we go up the branch to the earliest ancestor node where the target is not yet exceeded, and follow the

tree decision branches until the next violation of the target condition occurs, or the target is met.

Because this algorithm can be viewed as traversing a tree, in some sense the average behaviour is that it needs to check of order of the logarithm of the total number of total possible triangles as calculated in equation 1. However, the true behaviour is somewhat more difficult to estimate, given that the algorithm can run up and down the branches of the tree multiple times. What can be said is that if no solution exists, then in the worst case much fewer triangles than the number of equation 1 need to be checked, since triangles that were checked at an ancestor node don't need to be checked again. So in some sense, the worst case behaviour is also the logarithm of the number of triangles checked in the brute force algorithm.

### 1.4 Answer to the triangulation coding problem

**The answer is no**. That is, my algorithm finds that there is no colouring of the vertices such that there are exactly 2 neutral triangles. The method I wrote to find the solution can be applied to any graph, for any goal, and finds that the targets for which the problem has a solution are

$$\{3, 5, 7, 9, 11, 13, 15, 17, 19, 21\}, \tag{2}$$

as can be found out by running the code.

### 1.5 Time spent on problem

Including planning, coding, debugging and writing this readme file, I spent approximately 20 - 24 hours on this problem.

## 2 Description of code

### 2.1 Installation and execution instructions

This code is written in Java, and should be able to compile using version 5 or later.

To run in Eclipse, create a new Java project. Make sure the compiler compliance is set to 1.5 or above. Right-click on the src folder and create a new package with the name "triangulationProblem", then import the unzipped package folder included in the email into the package. The code should compile correctly.

The class that needs to be run to solve the problem is `NeutralTriangles`. The class `UnitTest` has test methods that can be run as JUnit Tests. I had to add JUnit4 to my build-path for `UnitTest` to work without error.

### 2.2 Classes and methods

The code is structured so that it may form part of a larger project, or extended to do many other possible graph computations. As written it can already be

applied to colour an arbitrary connected graph, with the aim of achieving some *goal* number of neutral triangles. The classes, most important methods, and their descriptions are given below:

- `GraphProperties`: Contains the *enum* `VertexColour` with the three colours a vertex can have. `VertexColour` contains the methods `getColourName` for returning the string value of a colour, and `getColour` for returning the *enum* element corresponding to a string.

- `Vertex`: Has the constructors for creating a new instance of a vertex in the graph. A vertex can have the properties *name*, which is an integer (as in figure 1) and *colour*, which is one of the elements of `VertexColour`, or can be null. It also has getters and setters for these properties. This class overrides `Object`'s `equals` method to check equality of a vertex based on its *name* only, as that is all that is required in this program.

- `PolygonGraph`: Is an instance of the graph and its properties. It contains three separate hashmaps with values being the vertices of the graph, arrays of vertices that are the nearest neighbours of each vertex (i.e., are connected with an edge to that vertex), and arrays of the triangles that a vertex forms part of (each triangle being an array with three instances of `Vertex`). The key for each of these hashmaps is the integer assigned to the vertex in the diagram (see figure 1).

  The graph is not built with a single constructor call, but rather is constructed by repeatedly calling the method `setNrstNbrs`, which takes two `int`s representing the *name*s of vertices in the graph. This method represents two vertices being joined by an edge in the graph. If one or both of the vertices are new, it is added to the hashmap *vertices*. Both vertices are added to each other's nearest neighbours in the hashmap *nearestNeighbours*. If there is a third vertex that is the nearest neighbour to both vertices, they form a triangle, and an array with all three vertices is added to the hashmap *triangles* with the *name* of each vertex used as a key.

  Other methods in this class are getters and setters allowing one to get and set a vertex's *colour* and *name* without needing to directly access the vertex, as well as an array with its nearest neighbours.

- `NeutralTriangles`: Contains the static methods that are designed to answer the specific class of questions posed by the problem. `isTriangleNeutral` determines if a triangle's vertices have all three colours and is therefore neutral. `countNeutralTriangles` uses `isTriangleNeutral` to count the number of neutral triangles that a given vertex belongs to.

  `colourGraph` takes as input a graph, the integer *name* of the first uncoloured vertex by the chosen numbering convention, the target number of neutral triangles and the current number according to the colouring of

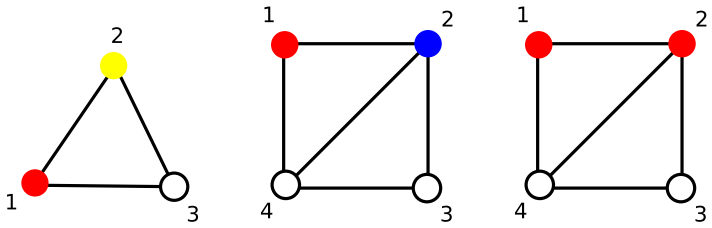


Figure 2: The graphs used for testing the algorithm and its failure in the unit test.

the graph. It then colours the graph according to the target, if possible, as described in more detail in section 1.

`drivewyzeGraphSetup` sets up the specific graph for the triangulation coding problem, as shown in figure 1. Finally the `main` method runs the program and outputs the solution, as well as all target numbers of neutral triangles that are possible.

- `UnitTest`: This class contains test methods testing the functionality of the program. It also has some static methods that create the small graphs shown in figure 2 to test `colourGraph`.

# 3 Concluding remarks

## 3.1 Things I would change on a second attempt

There are a few potential improvements that could be made:

1. I was not very careful with public and private methods. If I had more time, I would fix that.
2. I was also not too careful with using `Integer` vs. `int` in my code.
3. Currently the graph is initialized in the code itself. Ideally the nearest neighbour relations between vertices, as well as their colours would be fed in with a CSV file. There would then be a method that could take that input and generate the graph and all its properties.
4. Triangles are currently just `ArrayLists` containing 3 `Vertex` objects. It would be more appropriate if there was a class `Triangle` to contain the vertices.
5. The class `PolygonGraph` contains three hashmaps, using the integer *name* of the vertex as a key. Ideally, the key to these hashmaps would be the `Vertex` object itself, rather than the integer we have assigned to it.

6. The method `colourGraph` returns the integer value of 0 when the goal of 0 neutral triangles cannot be achieved, rather than the expected value of $-1$. This is an error that should be fixed.

### 3.2 Acknowledgements

The code was written in Eclipse Platform version 3.8.1. The diagrams in this readme was created using Inkscape 0.48, and the readme itself was created in LaTeX.

I referred to several sources while writing the code. For unit testing I used the top answer in [2]. For writing, compiling and debugging Java, I referred to [3]. For creating the figures in the readme, I referred to [4]. LaTeX related questions were referred to in [5].

## References

[1] Arne Storjohann, University of Waterloo CS240 Lecture Notes Based on lecture notes by R. Dorrigiv and D. Roche, 2013.

[2] stackoverflow.com/questions/8751553/how-to-write-a-unit-test

[3] docs.oracle.com/javase/tutorial/java/javaOO/constructors.html
docs.oracle.com/javase/tutorial/java/javaOO/enum.html
stackoverflow.com/questions/3811012/can-not-compile-enums-in-eclipse
stackoverflow.com/questions/2642589/how-does-a-arraylists-contains-method-evaluate-objects
stackoverflow.com/questions/18946657/cannot-instantiate-the-type-set

[4] inkscape.org/en/doc/tutorials/shapes/tutorial-shapes.en.html
tavmjong.free.fr/INKSCAPE/MANUAL/html/Z-Order.html

[5] tex.stackexchange.com/questions/74353/what-commands-are-there-for-horizontal-spacing
tex.stackexchange.com/questions/36030/how-to-make-a-single-word-look-as-some-code