# Data Structures and Algorithms II

Assignment 4

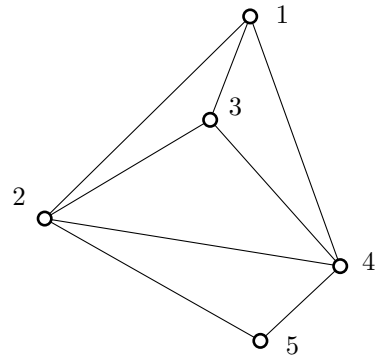
**Wachmann Elias**

January 24, 2022

# 1 Task Description

**Triangulation 3-coloring**
Your are given a triangulation of a point set. Your task is to design an efficient algorithm that constructs a valid 3-coloring of the points of the triangulation or determines that such a 3-coloring does not exist. A 3-coloring of the points is valid if any two points that are connected with an edge have different colors. The n points of the triangulation are labeled with the integers $\{1, \ldots, n\}$. The triangulation is given by a list of edges with additional triangle points (see Figure 1 for an example):

- Every edge is given by the labels of its two end points (first the smaller point label, then the larger one).
- For every edge, the labels of the point(s) with which the edge forms a triangle (a bounded triangular face) in the triangulation is given (two labels for interior edges and one label for edges on the boundary of the convex hull).




| edges | triangle points |
|---|---|
| $1-2$ | $3$ |
| $2-4$ | $3,5$ |
| $3-4$ | $1,2$ |
| $1-3$ | $2,4$ |
| $4-5$ | $2$ |
| $2-5$ | $4$ |
| $2-3$ | $1,4$ |
| $1-4$ | $3$ |

**Figure 1:** Example of a triangulation and a list of its edges with triangle points.

Explain and describe your algorithm in detail, analyze its runtime and memory requirements, and give reasons for the correctness of your solution.

# 2 Description of algorithm

**Note:** It is assumed from the example, that the edges are always given with the lower vertex number in the first place e.g. 1-5 instead of 5-1. If this is not a given, the edge representation could be changed to conform to the above constraint in linear time, thus not increasing the asymptotic runtime.

**General remark:** The algorithm takes a list of edges with additional triangle points `EDGES` (as described above) as input and outputs a 3-coloring of the graph `COLORS` if it is possible. If no valid 3-coloring of the given graph exists, the algorithm exits and indicates that no such coloring is possible (`COLORS` is not returned in this case).

The algorithm starts with the **Setup** step, then the points on the first edge and it's triangle points are colored in **Start**, after which **Loop** takes care of the remaining points. Afterwards the validity of the coloring is checked and if the coloring is valid `COLORS` is returned.

1. **Setup:**
   First get number of points. The number of points is given by the maximum index of all triangle points from every edge in `EDGES`.Use this number to create an array `COLORS` with size $n_v$ for the colors of the vertices and set each entry to NULL (= no color). Store edges (keys / hash) and triangle points (+optional flags) (values) in a hashmap `EDGE_DIC` for fast access. Setup the counter `counter` which tracks how many vertices need to be colored in. Setup empty queue `NEXTEDGES` for neighboring edges.

2. **Start:**
   An arbitrary start edge, for example the first in `EDGES` is chosen as `cur_edge`. The two endpoints $p_1$ and $p_2$ of `cur_edge` are colored with $c_1$ and $c_2$ respectively. Then the triangle points are colored[1] in. Finally the `counter` is decreased by the number of vertices which were colored in and `cur_edge` is deleted from `EDGE_DIC`.

3. **Loop:** The Loop is executed while `counter` is not 0.
   All neighbors of `cur_edge` which were not visited before (meaning they were never `cur_edge`) are added to the queue `NEXTEDGES`. Neighbors to the `cur_edge` are edges which form a triangle with the `cur_edge` if the two endpoints (one form the `cur_edge` and the triangle point) are connected.[2] Now `cur_edge` is set to the last edge in `NEXTEDGES` (according to the FIFO-principle). The colors $c_1$ and $c_2$ of $p_1$ and $p_2$ of `cur_edge` are checked. If they are the same the algorithm terminates because no valid 3-coloring of the given graph exists. Otherwise the algorithm tries to color the triangle point(s) of `cur_edge` (at least one [outer edge] and at most 2 triangle points [inner edges]). If the triangle point is already colored with $c_3$, it is check if the color is neither $c_1$ nor $c_2$, if the color would match either $c_1$ or $c_2$ the algorithm terminates as in the aforementioned case where $c_1$ equals $c_2$. If the triangle point is properly colored nothing happens and if applicable the second triangle point is checked. In the case that the triangle point isn't colored already, it is colored with $c_3$ such that $c_3$ differs from $c_2$ and $c_1$. Finally the `counter` is decreased by the number of vertices which were colored in and `cur_edge` is deleted from `EDGE_DIC`. Then the loop repeats until all vertices are colored in which is indicated by the `counter` reaching 0.

4. **Check validity of coloring of remaining edges:**
   For the graph in Figure 1 **without Point 5** (without edges 2-5 and 4-5) the al-

[1] Depending on the edge type either one (outer edge e.g. 1-2 in Figure 1) or a maximum of two triangle points (inner edge e.g. 1-3 in Figure 1) are colored in.

[2] This definition of neighbors guarantees that the so formed triangle already has two colored points, thus making the coloring of the third trivial.
**Example:** The edge 1-2 in Figure 1 has 2 neighbors (1-3 and 2-3) according to the above definition, whereas an inner edge has 4: e.g. 3-4 has 1-3, 1-4, 2-3 and 2-4 as neighbors

gorithm could produce a impossible coloring and end.[3] To avoid these edge cases one could check all edges and their triangle points for matching colors - after the loop has terminated - as described in 'Loop' above. However color-checking only the remaining edge-triangle-point-pairs in `EDGE_DIC` is sufficient, because these are the edges which were never `cur_edge`. Asymptotically this reduction in remaining edge-triangle-point-pair color checks doesn't matter as we'll see below.
At last if the 3-coloring is correct the list of colors `COLORS` will be returned.

## 3 Space complexity / memory requirements

**Note:** As derived from Euler's formula in the lecture for a connected, simple, planar graph the following holds true for $v \geq 3$:

$$e \leq 3v - 6 \tag{1}$$

Where $e$ is the number of edges, and $v$ the number of vertices in the graph. For this reason asymptotically the number of edges as well as the number of vertices are the same ($\mathcal{O}(e) = \mathcal{O}(v)$). For this reason the analysis in the current section, as well as section 4 will use the size of the input, more precisely the number of edges $n$ in `EDGES`.

The memory requirements of the algorithm is given as follows:

- `EDGES` [input] stores all $n$ edges of the graph which results in $\mathcal{O}(n)$ space complexity.
- `COLORS` [output] stores asymptotically (as shown above) $\mathcal{O}(n)$ colors of the vertices which results in $\mathcal{O}(n)$ space complexity
- `EDGE_DIC` stores all the edges from the [input] `EDGES` as key/hash-value pair in a hashmap which requires $\mathcal{O}(n)$ space.
- `NEXTEDGES` stores at most $n - 1$ edges[4] which results in $\mathcal{O}(n)$ space complexity.
- Local variables: `counter` and `cur_edge` have $\mathcal{O}(1)$ space complexity.

All together the data structures require $\mathcal{O}(n)$ space.

## 4 Runtime complexity

The runtime complexity analysis is based on the number of edges $n$ in the input `EDGES`. As shown above in Equation 1, $n$ (= number of edges $e$) is asymptotically the same as $v$ the number of vertices.
The Analysis is split into the same four parts as in section 2:

[3] The `counter` would reach 0 if the start edge would be 1-3 (The loop would be skipped and thus no checks would be done)
[4] $n - 1$ is for example true in the trivial case with $n = 3$

1. **Setup:** The number of the points is found by a linear scan over all $n$ elements in `EDGES`, thus resulting in $\mathcal{O}(n)$ time complexity. Then `COLORS` and `EDGE_DIC` are allocated and filled with NULL and the edges with corresponding triangle points respectively, also resulting in $\mathcal{O}(n)$ time complexity each (note: amortized time complexity of inserting into a hashmap is of $\mathcal{O}(1)$). The instantiation of `NEXTEDGES`, `counter` and `cur_edge` happens asymptotically in constant time.
   Time complexity of this section: $\mathcal{O}(n)$

2. **Start:** An edge is chosen from `EDGES` and assigned to `cur_edge` which takes constant time. Then the vertices of this edge as well as the triangle points (at most 2) are colored. This means that the corresponding colors in the `COLORS` array are set (at most 4) which takes $\mathcal{O}(4)$ time at most. Finally `counter` is decreased by the number of colored points (3 or 4) and `cur_edge` is deleted from `EDGE_DIC` which both take constant time.
   Time complexity of this section: $\mathcal{O}(1)$

3. **Loop:** The Loop is executed at most $n$-times because every point is at least a 'triangle point' to one edge. Per loop pass non, or at most one triangle point gets colored. First, the neighboring edges to `cur_edge` are added to `NEXTEDGES`. In each pass at most 4 edges are added (adding elements into a queue with size $n$ which takes at most $n$ elements can always be achieved in constant time[5]); It's important that edges are only added once to `NEXTEDGES`. This can be accomplished by setting a flag in the corresponding hashmap value in `EDGE_DIC` (takes constant time). The check if an edge was visited before can be done by trying to access the hash in `EDGE_DIC` (constant time access). Next `cur_edge` is set as the last edge from `NEXTEDGES` (this edge gets deleted in the queue [pop-operation]) which also happens in $\mathcal{O}(1)$ time.
   Now all the color checks of the points in the edge and the triangle point are performed. First the colors of the edge points are checked, then 2 checks (if triangle point color is the same as one of the edge points and if triangle point is already colored) are performed for each of at most 2 triangle points. This yields at most 5 checks which are all performed in constant time as well. Should colorable points exist, they are colored in constant time, as described above. Last but not least `counter` is decreased by the number of colored points (0 or 1) and `cur_edge` is deleted from `EDGE_DIC` which takes constant time. The overall time complexity of the loop is determined by the time complexity of each pass multiplied by the number of passes ($n$). Every step inside the loop happens in constant time, resulting in the following overall complexity:
   Time complexity of this section: $\mathcal{O}(n)$
   Note: If no valid 3-coloring of the graph exists the algorithm simply stops and skips the next step, thus not impacting the asymptotic runtime.

4. **Check validity of coloring of remaining edges:** As just described checking the validity of the coloring of an edge and it's triangle point(s) is accomplished in constant time. Time complexity wise checking all $n$ edges would take $\mathcal{O}(n)$ time.

[5] simply use an array with size $n$ and two points to the start and end of the queue. Insert at the end-pointer and pop from the front.

Checking fewer, as stated in section 2, will indeed decrease the runtime but will not have an effect on the asymptotic bound of $\mathcal{O}(n)$. The return of `COLORS` can be done in constant time.
Time complexity of this section: $\mathcal{O}(n)$

The overall runtime of the algorithm is now given by the sum of the aforementioned 4 parts which are run successively.
All together the time complexity of the algorithm is $\mathcal{O}(n)$.

## 5 Correctness of the algorithm

To guarantee a correct 3-coloring of the graph the algorithm must not get trapped in a coloring schema which will not yield a valid 3-coloring, even though one exists for a given graph. This is achieved by choosing the correct edges (only neighbors of the initial edge as `cur_edge`) after the initial edge [6]
After coloring an initial edge and it's triangle points, which is trivially correct[7], neighboring edges are colored. However, all neighboring edges consist of one triangle point and one vertex from the initial edge which are already colored, making the coloring of the remaining point also trivial. Now the correctness is easy to show:
The initial edge and it's triangle point are colored correctly and so all remaining uncolored points in the neighboring edges can be colored correctly. In this way the algorithm can't end up in a dead end, because every edge has at least 2 and at most 4 neighboring edges. These edges are all added to the queue `NEXTEDGES`, which intern guarantees that the algorithms colors from the initial edges 'outwards' (first it's neighbors, then their neighbors and so on). In this order every edge will be put into `NEXTEDGES` and thus in the worst case every edge will be `cur_edge` before all points are colored.

Note on the case that not valid 3-coloring exists: For the case that no valid 3-coloring is possible like in the given example in Figure 1, the algorithm (e.g. starting from edge 1-3) will color 2 and 4 with the same color, however the algorithm will immediately notice this when trying to color one of the neighboring edges.
For the special case, that there are no edges left, like in the given example in Figure 1 **without vertex 5** (also starting from edge 1-3), all vertices would be colored and 2 and 4 again in the same color. This discrepancy is then spotted in the fourth step of the algorithm which checks all edges which were not previously the `cur_edge`. Then the algorithm would terminate and report that no valid coloring exists if either the edge 2-3, 3-4 (neighbors of 1-3) or the edge 2-4 is checked.

[6] A detailed definition of neighboring edges is given in section 2
[7] Simply choose 2 different colors for the vertices in edge and the remaining for the triangle point(s)