BACHELOR

Planarity testing

implementation and analysis of the Auslander-Parter algorithm

Zeijlemaker, Sjanne

Award date:
2018

Eindhoven University of Technology

Planarity testing

Implementation and analysis of the Auslander-Parter algorithm

Author:
Sjanne Zeijlemaker

Supervisor:
Dr. J.C.M. Keijsper

July 12, 2018

Abstract

In 1961, Auslander and Parter first introduced an algorithm to test arbitrary graphs for planarity. We explore the mathematical background of this algorithm, prove its running time to be cubic and present a Java implementation of the algorithm. Suitable test data is generated using planar and non-planar graph generation algorithms. Based on this test data, we examine the correctness of the algorithm and its practical running time. Finally, a visualisation tool is presented that shows the steps of the Auslander-Parter algorithm on an arbitrary graph.

Contents

1 Introduction

A graph is considered planar if can be drawn in the plane such that none of its edges intersect. Planar graphs are used in many practical applications, ranging from electronic circuit board design to the visualisation of transport networks. On a theoretical level, planarity forms the basis of some important graph theory results, such as Euler's Theorem and the Four Colour Theorem.

A complete characterisation of planarity was first presented in 1930 by Kuratowski [Kuratowski, 1930]. However, despite its theoretical importance, Kuratowski's theorem yielded no computationally efficient planarity test. The first polynomial-time algorithm for planarity testing was proposed in 1961 by Auslander and Parter [Auslander and Parter, 1961] and later corrected by Goldstein [Goldstein, 1963]. Shirey provided an implementation of Goldstein's algorithm in [Shirey, 1969] and proved its running time to be cubic. Hopcroft and Tarjan [Hopcroft and Tarjan, 1973] further improved Auslander-Parter's method into a linear-time algorithm. While this is asymtotically optimal for planarity testing, more recent algorithms such as Boyer–Myrvold [Boyer and Myrvold, 2004] and Fraysseix-Mendez-Rosenstiehl [De Fraysseix et al., 2006] outperform Hopcroft-Tarjan's algorithm in practice.

Although the Auslander-Parter algorithm is not as efficient as some state-of-the-art planarity tests, it has some interesting qualities. Using the Jordan Curve Theorem, it provides an intuitive characterisation of planarity which does not make use of Kuratowski subdivisions. Its implementation is relatively straightforward and does not rely on complex data structures or programming techniques. Nevertheless, the algorithm has not been analysed as extensively as its modern counterparts due to its restrictive running time. This report aims to fill this gap by providing a detailed mathematical analysis and an implementation of the Auslander-Parter algorithm, as well as a method to generate planar and non-planar graphs for testing purposes.

2 Basic definitions

All definitions in this section are adapted from [Tamassia, 2013], unless indicated otherwise.

A *graph* $G = (V, E)$ is an ordered pair consisting of a finite set V of vertices and a finite set E of edges. Each edge is an unordered pair (u, v) with $u, v \in V$. An edge is called a *self-loop* if $u = v$. G is called *simple* if it contains no self-loops and if has at most one edge between each pair of vertices. Edge e is *incident* with a vertex v if $v \in e$. The number of edges incident with v is called the *degree* of v, denoted by δ_v. Vertices u and v are said to be *adjacent* if there exists an $e \in E$ such that $e = (u, v)$. A *subgraph* of G is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

In this thesis we will only consider simple graphs, although it should be noted that all algorithms can easily be extended to support multigraphs with self-loops.

A *walk* is a sequence of vertices $v_0, v_1, \ldots, v_k$, such that $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, k-1$. The *length* of a walk is equal to the number of edges it contains. A walk that visits each vertex no more than once is a *path*. A *cycle* is a *closed* walk, i.e. a walk such that $v_0 = v_k$. A cycle C is *simple* if it is a closed path. A graph is called *acyclic* if it contains no simple cycles.

Graph G is *connected* if for any $u, v \in V$ there is a path in G from u to v. G is *k-vertex connected* if removing any $k-1$ vertices leaves the graph connected. Here we will only consider 2-vertex connectivity, also referred to as *biconnectivity*. A *biconnected component* is a maximal biconnected subgraph. A *cut vertex* or *articulation point* is a vertex v such that removing v and its incident edges leaves the graph disconnected. Figure 1 shows the decomposition of a graph into biconnected components. Proposition 2.1 below lists a few basic characteristics of biconnected components.

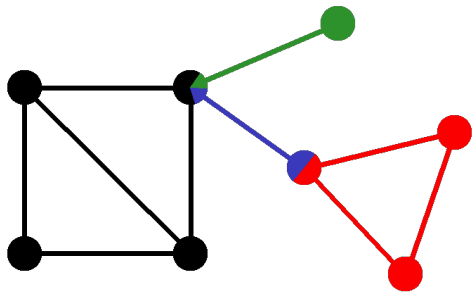


Figure 1: A graph with four biconnected components, each marked with a different colour.

Proposition 2.1 (see e.g. [Bondy and Murty, 2008])**.** *Let G be a graph, then*

1. *any two biconnected components of G have at most one vertex in common*
2. *the biconnected components of G partition the edges of G.*

Proof. 1. Suppose there exist biconnected components $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ that share at least two vertices. Define $B = (V_B, E_B) = (V_1 \cup V_2, E_1 \cup E_2)$ and let $v \in V_B$. Then $B' = (V_B \setminus v, E_B)$ is a connected graph, because at most one of the common vertices of B_1 and B_2 has been removed. Since v is chosen arbitrarily, it follows that B has no articulation points. But then B_1 and B_2 are not maximal ↯.

2. An edge with its incident vertices is a biconnected subgraph. Therefore, each edge is contained in at least one biconnected component. Suppose there exists an edge $e = (u, v)$ which is part of two biconnected components. Then these components have vertices u and v in common, which contradicts part 1 of this proposition.

It follows that each edge belongs to exactly one biconnected component, hence the biconnected components induce a partition on the edges of G.

□

Graph $G = (V, E)$ is *bipartite* if its set of vertices V can be partitioned into two classes U and W such that for every $e = (u, w) \in E$ either $u \in U$ and $w \in W$ or $u \in W$ and $w \in U$.

Theorem 2.2 (For example [Harary, 1969])**.** *A graph G is bipartite if and only if it contains no cycles of odd length.*

Proof. Let G be bipartite with bipartition $\{U, W\}$ of the vertices and let C be an arbitrary cycle in G. Then all vertices of C must alternately be in U and W. Since C is closed, this means that C must contain an even number of vertices and edges.

Suppose that $G = (V, E)$ contains no odd cycles and assume, without loss of generality, that G is connected. Define the distance between two nodes u and v as the number of edges on the shortest path from u to v. Let u be a vertex of G. Partition V into two sets U and W: vertices with an odd distance to u and vertices with an even distance to u respectively. There can be no edge between any nodes u_1 and u_2 from U, because then the concatenation of this edge and the shortest paths from u to u_1 and u_2 forms an odd cycle. The same argument holds for all pairs of vertices from W. Then $\{U, W\}$ is a bipartition of G. □

Many graph algorithms make use of a special type of graph called a tree. A *tree* T is a connected acyclic graph. Exactly one of its vertices is labelled as the *root* r. A *forest* is a graph which only consists of connected components that are trees. A vertex v is a *child* of vertex u if it is adjacent to u and further away from the root than u. u is then called *v's parent*. A *leaf* of a tree is a vertex with no children. A *descendant* of u is a vertex that can be reached by starting at u and repeatedly going from parent to child. An *ancestor* of u is a vertex that can be reached by starting at u and repeatedly going from child to parent.

Let T be a tree containing vertex x. A *subtree rooted at x* is a subgraph of T consisting of all descendants of x and the edges between them. It is denoted by T_x.

A *drawing* Γ of a graph G maps each vertex v to a distinct point $\Gamma(v)$ of the plane and each edge (u, v) to a simple open Jordan curve $\Gamma(u, v)$ with endpoints $\Gamma(u)$ and $\Gamma(v)$. A graph drawing is called a *planar drawing* or *embedding* if no two distinct curves intersect at a point which is interior to either of them. A graph is *planar* if it admits a planar drawing. A planar drawing fixes the the clockwise circular order of the edges incident to each vertex, the *rotation scheme*. If two planar drawings fix the same rotation scheme they are considered equivalent. A planar embedding is an equivalence class of planar drawings, described by the rotation scheme of the vertices.

Theorem 2.3 (Euler)**.** *For each planar connected graph $G = (V, E)$ the following equality holds:*

$$|V| - |E| + |F| = 2,$$

where $|F|$ denotes the number of faces induced by the embedding of G.

Lemma 2.4. *For each planar connected graph $G = (V, E)$ with $|E| \geq 2$ the number of edges is limited by $|E| \leq 3 \cdot |V| - 6$.*

Proof. Let f_i denote the number of faces that are incident to i sides of edges. Since $E \geq 2$, each face is incident to at least three sides, so the total number of faces is given by $|F| = \sum_{i=3}^{2\cdot|E|} f_i$. Every edge has exactly two sides, so it follows that

$$2 \cdot |E| = \sum_{i=3}^{2\cdot|E|} i \cdot f_i \geq \sum_{i=3}^{2\cdot|E|} 3 f_i = 3 \cdot |F| \quad \Rightarrow \quad |F| \leq \frac{2}{3} \cdot |E|. \tag{1}$$

By applying Theorem 2.3 we find that $2 = |V| - |E| + |F| \leq |V| - \frac{1}{3} \cdot |E|$, hence $|E| \leq 3 \cdot |V| - 6$. □

The following concepts and results are well known and can be found in any textbook on algorithmic graph theory, such as [Cormen, 2009].

A commonly used graph algorithm is *Depth-First Search*. DFS traverses all vertices of a graph in a Last-In, First-Out order and keeps track of parent-child relations between vertices with regard to discovery time. The parent-child relations imposed by the algorithm form a forest containing all vertices in V. A DFS partitions the edges of a graph into two classes. The edges contained in the DFS forest are called *tree edges* and the others *back edges*.

An alternative approach to graph traversal is a Breadth-First Search. The core algorithm is similar to DFS, but it traverses the vertices in a First-In, First-Out order. Whereas Depth-First Search always goes to the maximum depth before returning, BFS traverses the graph layer by layer.

DFS and BFS are closely related to the data structures *stack* and *queue*. These are dynamic sets, sets that can change over time, for which the delete operator removes a pre-specified element. A stack uses a First-In, First-Out policy, i.e. the element last inserted is first removed from the set. A queue first deletes the element that was first added to the set. Stacks and queues are often used in the implementation of DFS and BFS respectively. Algorithm 1 shows such an implementation in pseudo code.

Algorithm 1: DFS/BFS

input : A graph G with vertices V and edges E
output: A DFS/BFS forest of G

```
1  DFS(G, u)
2      mark u as discovered
3      put u on the stack
4      while the stack is not empty do
5          v = delete(stack)
6          mark v as discovered
7          for each vertex w adjacent to v do
8              if w has not been discovered then
9                  set v as parent of w
10                 put w on the stack
11 BFS(G, u)
12     mark u as discovered
13     add u to the queue
14     while the queue is not empty do
15         v = delete(queue)
16         if v has not been discovered then
17             mark v as discovered
18             for each vertex w adjacent to v do
19                 if w has not been discovered then
20                     set v as parent of w
21                     add w to the queue
22 startTraversal(G)
23     for each vertex v in G do
24         if v has not been discovered then
25             DFS(G, v)/BFS(G, v)
```

Lemma 2.5. *Let T be the DFS tree of connected graph G and T_r a subtree of T rooted at r. There are no edges between the subtrees of any two children of r.*

Proof. Let u and v be two children of r and assume, without loss of generality, that the DFS algorithm visits u before v. Then DFS first searches the entire subtree of u before returning to r. If there was an edge between T_u and T_v, v would be in u's subtree, hence not a child of r. ↯ □

The time it takes to complete an algorithm is called the *running time* of the algorithm and is expressed in the number of elementary operations, i.e. operations whose execution time is constant. These include basic arithmetic and data movement. Here we are only interested in the worst-case running time of algorithms, because it gives a hard bound on execution time. It is guaranteed that the algorithm will never take longer, whereas a best- or average-case analysis might only provide an accurate approximation in very specific cases.

Running times can be expressed with the *O-notation*, which expresses an asymptotic upper bound. Let n denote the size of the input of an algorithm and f a function that maps the input size to the number of elementary operations it takes to complete the algorithm. We say that $f(n) = O(g(n))$ for some function g if there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)\ \forall_{n \geq n_0}$. In plain, this means that if the input size grows large enough, the number of elementary operations required will grow proportional to g.

Lemma 2.6. *DFS and BFS have a running time of $O(|V| + |E|)$ on a graph $G = (V, E)$.*

Proof. A vertex is marked as discovered when it is first processed and the algorithm only visits undiscovered vertices. This means a vertex can never be visited more than once. For each vertex, all adjacent vertices are inspected to see whether they have been visited. This combines to a running time of $O(\sum_{i=0}^{|V|-1}(1 + \delta_{v_i})) = O(|V| + 2|E|) = O(|V| + |E|)$. □

3 Characterisation of planarity

Now that we have defined what constitutes a planar graph, a method is needed to establish whether a given graph is planar. This section introduces the most commonly used characterisation of planarity.

Graph $G = (V, E)$ is *complete* if $e = (v_i, v_j) \in E$ for all pairs of distinct vertices $v_i, v_j \in V$. The complete graph on n vertices is denoted by K_n. Bipartite graph $G = (U \cup W, E)$ is complete if $e = (u_i, w_j) \in E$ for all pairs of vertices $u_i \in U, w_j \in W$. If $|U| = m$ and $|W| = n$, then the corresponding complete bipartite graph is denoted by $K_{m,n}$.

A *subdivision* of graph G is a graph that is obtained by repeatedly subdividing edges of G. *Subdividing* edge $e = (u, v)$ means adding a new vertex w to the graph and replacing e with edges $e_1 = (u, w), e_2 = (w, v)$. Figure 2 illustrates the subdivision of an edge in K_2. The reverse operation is *smoothing out* a vertex of degree 2. Note that a subdivision or smoothing in a planar graph results in a planar graph.

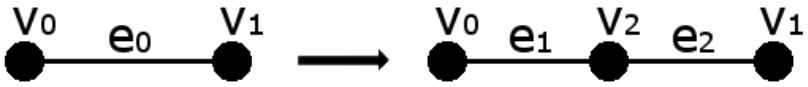


Figure 2: Subdivision of edge e_0.

Let G be a graph that contains a subgraph G' that is a subdivision of K_5 or $K_{3,3}$ (see Figure 3). Then G' is called a *Kuratowski subgraph* of G.

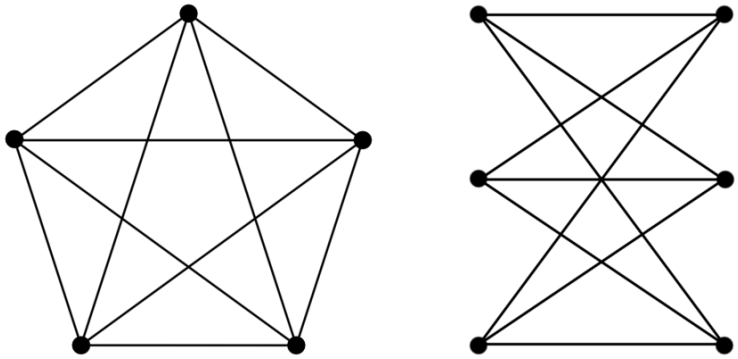


Figure 3: From left to right: K_5 and $K_{3,3}$, the complete graph on 5 vertices and the complete bipartite graph on 3,3 vertices.

Theorem 3.1 (Kuratowski's)**.** *A graph is planar if and only if it does not contain any Kuratowski subgraphs [Kuratowski, 1930].*

Using Lemma 3.2 below, it is easy to see that a graph containing a subdivision of K_5 or $K_{3,3}$ is not planar. The other implication of the theorem is more involved and beyond the scope of this project.

Lemma 3.2. *K_5 or $K_{3,3}$ are not planar.*

Proof. K_5 has $\frac{1}{2} \cdot 5 \cdot 4 = 10$ edges and 5 nodes, so by Lemma 2.4 it is not planar.

Since $K_{3,3}$ is bipartite, it does not contain any odd cycles (Lemma 2.2). However, faces incident to exactly 3 sides of lines can only be the result of a cycle of length 3. In this case, Equation 1 from the proof of Lemma 2.4 reads as

$$2 \cdot |E| = \sum_{i=4}^{2\cdot|E|} i \cdot f_i \geq \sum_{i=4}^{2\cdot|E|} 4 f_i = 4 \cdot |F| \qquad \Rightarrow \qquad |F| \leq \frac{1}{2} \cdot |E|.$$

By applying Theorem 2.3 we find that $2 = |V| - |E| + |F| \leq |V| - \frac{1}{2} \cdot |E|$, so for any bipartite planar graph it holds that $|E| \leq 2 \cdot |V| - 4$. The bipartite graph $K_{3,3}$ has 9 edges and 6 nodes, so it is not planar. □

4 The Auslander-Parter algorithm

In theory, Kuratowski's characterisation of planarity provides a solid method to verify whether a graph has a planar embedding. However, in practice it is both difficult and computationally inefficient to search for subdivisions of K_5 and $K_{3,3}$. In 1961, Auslander and Parter published a more intuitive approach that uses a different characterisation of planarity. The original algorithm as proposed in [Auslander and Parter, 1961] might loop indefinitely. In [Goldstein, 1963], Goldstein formulated an iterative version of the algorithm correcting this mistake. In this report, we will mostly adhere to the version described in [Battista et al., 1999], which makes use of the properties of biconnected graphs to simplify the algorithm.

4.1 Theoretical background

Define the merging operator $+$ on two graphs as follows: if $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ then $G_0 + G_1 = (V_0 \cup V_1, E_0 \cup E_1)$.

From now on, we will only consider simple cycles, which will simply be referred to as 'cycle'. The definitions in this paragraph are based on [Tamassia, 2013].

Lemma 4.1. *Graph G is planar if and only if its biconnected components are planar.*

Proof. If G is planar then it is evident that its biconnected components are also planar.

Suppose that G has n biconnected components denoted by $B_1, \ldots, B_n$, all of which are planar. The subgraph consisting of only B_1 is planar by assumption. Suppose subgraph $G' = B_1 + \cdots + B_k$ is planar for a certain $0 < k < n$ and consider $G'' = G' + B_{k+1}$. There are two possible cases:

- B_{k+1} and G' have no vertices in common. Then G'' consists of two disjoint planar subgraphs, so it is planar.
- If B_{k+1} and G' are not disjoint, their intersection exists of one vertex (Proposition 2.1). Call this shared vertex v. Since we only consider graphs without self loops, B_{k+1} and G' must be edge-disjoint. G' and B_{k+1} are planar, so they have an embedding. This means that in both G' and B_{k+1} there is a way to arrange the vertices around v such that none of the edges intersect. Concatenating these rotation schemes for v then results in a valid rotation scheme for v in G'', so G'' has a valid planar embedding.

In both cases G'' is planar, so G is planar by induction. □

Lemma 4.2. *A biconnected graph on three or more vertices contains a cycle.*

Proof. Suppose G is a biconnected graph with $|V| \geq 3$ that contains no cycles. If G is biconnected it is certainly connected, so G is a tree. Then by definition there is a path between each pair of vertices. This path must be unique: if there were two different paths between any vertex pair, the parts where they are disjoint would form cycles. Since G has three or more vertices, there must be at least one vertex pair that is two or more edges apart. If we pick such a pair and remove any vertex u from the interior of the unique path between them, the resulting graph $G - u$ is no longer connected. But then G has an articulation point. ↯ □

Let G be a biconnected graph containing a cycle C. A *segment* of C is either a connected component in $G \setminus C$ together with the vertices connecting it to C or a *chord*: an edge between two non-consecutive vertices in C. A cycle is called *separating* is it has at least two segments. The vertices that a segment S and cycle C have in common are called the *attachments* of S. Because of biconnectivity each segment has at least two attachments.

Let S_1, S_2 be segments of cycle C with attachments a_1, b_1 and a_2, b_2 respectively. The attachments are called *interleaving* if they appear in the cyclic order a_1, a_2, b_1, b_2 in C. Figure 4 illustrates the difference between interleaving and non-interleaving attachments. Segments S_1 and S_2 are called *conflicting* if their attachments interleave. Two segments that do not conflict are called *compatible*.

Theorem 4.3 (Jordan Curve Theorem)**.** *Every simple closed curve in the plane separates its complement into two connected, non-empty sets: the interior region and exterior region [Jordan, 1893].*

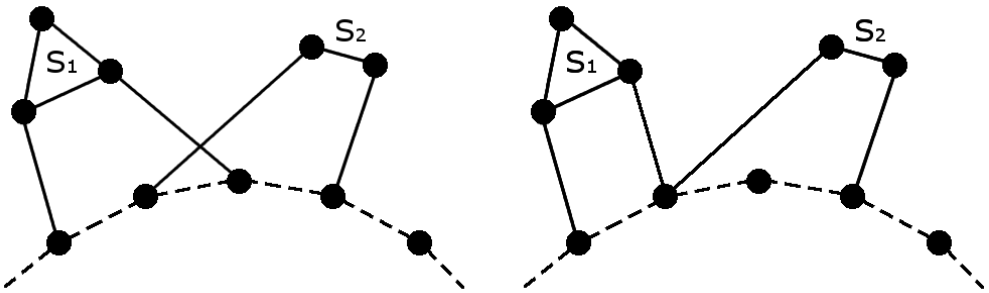


Figure 4: The graph on the left has two segments with interleaving attachments. The attachments in the graph on the right do not interleave.

Corollary 4.3.1. *No two points from the exterior and interior region of a simple closed curve can be connected by a line that doesn't intersect with the curve.*

Suppose that a planar graph G contains a cycle and let Γ be a planar embedding of G. Then the cycle forms a simple closed curve in Γ. By Corollary 4.3.1 each of its segments must be embedded entirely in the interior or exterior region of this curve.

Lemma 4.4. *Let C be a cycle with segments $S_1, S_2, \ldots, S_k$ such that $C + S_i$ is planar for every $1 \leq i \leq k$. Let Γ denote a planar embedding of C. Then $S_1, S_2, \ldots, S_k$ can be embedded on the same side of Γ if and only if their attachments do not interleave.*

Proof. Suppose C has a pair of conflicting segments S_i, S_j. Let a_i, b_i and a_j, b_j denote the interleaving attachments of S_i and S_j respectively. S_i and S_j are connected, so there must be an a_i - b_i path in S_i and an a_j - b_j path in S_j. The subgraph consisting of C and these two paths is a subdivision of K_4. Suppose S_i and S_j can be embedded on the same side of Γ. Then we may without loss of generality embed them in the exterior region of Γ. Define a new vertex v and embed it in the interior region of Γ. We can connect v to each of the attachments a_i, b_i, a_j, b_j with an edge without violating planarity, as illustrated in Figure 5. But then we have obtained a valid embedding of a subdivision of K_5 ↯.

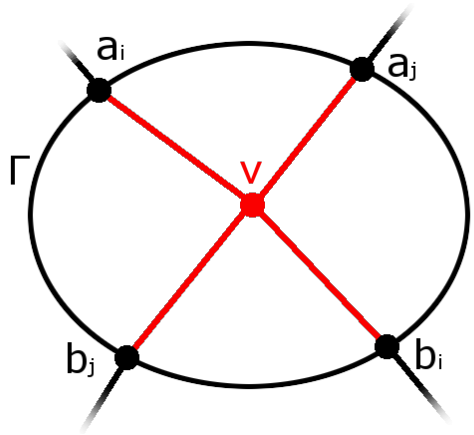


Figure 5: An embedding Γ of a cycle with two conflicting segments drawn in the exterior region of Γ. Point v can be drawn in the interior of Γ and connected to attachments a_i, b_i, a_j and b_j without compromising planarity.

Suppose no two segments from $S_1, S_2, \ldots, S_k$ interleave. Let $\Gamma(C)$ denote a valid embedding of C. Without loss of generality, we will show that we can then embed $S_1, S_2, \ldots, S_k$ in the exterior region of $\Gamma(C)$ by induction on k.

If C has only one segment S_1, we may embed it in the exterior region of $\Gamma(C)$, because $S_1 + C$ is planar.

Suppose that for any cycle C with $n < k$ compatible segments there exists an embedding such that $S_1, S_2, \ldots, S_n$ are embedded in the exterior region of $\Gamma(C)$. Now consider a cycle C' with $n+1$ compatible segments and embedding $\Gamma(C')$. Since all segments of C' are compatible, there must be a segment S_i whose attachments appear in some cyclic order $a_1, a_2, \ldots, a_m$ in C' such that there is no attachment from any other segment between a_1 and a_m. We can embed this segment outside of $\Gamma(C')$. Now consider the cycle C'', which is constructed by concatenating the a_1 - a_m path in C' that does not contain $a_2, \ldots, a_{m-1}$ and the path corresponding to the outside boundary of the embedding of S_i. Note that the exterior region of the embedding of C'' does not contain any vertices of S_i or C'. Since all segments of C' were compatible and no segment had attachments between a_1 and a_m, the remaining segments $S_1, S_2, \ldots S_{i-1}, S_{i+1}, \ldots, S_{n+1}$ form a set of n compatible segments of C'' and $C'' + S_j$ is planar for every $j \in \{1, 2, \ldots, i-1, i+1, n+1\}$. Then by our induction hypothesis, $S_1, S_2, \ldots S_{i-1}, S_{i+1}, \ldots, S_{n+1}$ can be embedded outside of C''. □

An *interlacement graph* is a graph that stores the conflicts between segments of a cycle. Each segment is represented as a vertex. An edge is added for each pair of conflicting segments.

The above definitions can be combined into a new characterisation of planarity that does not rely on Kuratowski's theorem.

Theorem 4.5. *Let G be a biconnected graph with a cycle C. G is planar if and only if the following conditions hold:*

- *The interlacement graph of the segments of C is bipartite.*
- *For each segment S, the graph $C + S$ is planar.*

Proof. Suppose G is planar and let Γ be a planar embedding. Note that a cycle in a graph drawing is essentially a simple closed curve in the plane. From Theorem 4.3 it follows that each segment of C must be drawn either completely inside the interior region or completely inside the exterior region of C. This partitions the segments into two distinct sets which are compatible by Lemma 4.4, so the interlacement graph of the segments is bipartite. If G is planar then each subgraph obtained by merging a segment S with C is also planar.

Suppose the interlacement graph of the segments of C is bipartite and the graph $C + S$ is planar for each segment S. Then the segments can be partitioned into two sets such that each set contains no pair of conflicting segments. From Lemma 4.4, it follows that all segments in each set can be drawn on the same side of C without compromising planarity. Assign one set to the exterior region and one to the interior region of C to obtain a planar drawing of G. □

4.2 Outline of the algorithm

The Auslander-Parter algorithm, from now on referred to as AP, is composed of the following steps:

i. Divide the graph into biconnected components.

ii. For each component:

   a) Find a cycle and the resulting segments.

   b) Compute the interlacement graph of the segments.

   c) Test if the interlacement graph is bipartite.

   d) Repeat for each subgraph formed by merging a segment with the cycle.

Each section below discusses one of these steps in detail.

i. Biconnected components

The initial step of the AP algorithm is to find all biconnected components of the graph G. From Lemma 4.1 it follows that individually testing these components for planarity is sufficient to determine whether G is planar. It is possible to identify all the articulation points with an extended version of the Depth-First Search algorithm introduced by Hopcroft and Tarjan [Hopcroft and Tarjan, 1973]. It is based on the following observations, which provide a constructive method to find the articulation points of a graph.

Lemma 4.6. *For any connected graph G with DFS tree T, the following properties hold:*

1. *Any vertex $v \in T$ that is not the root is an articulation point of G if and only if v has a child w such that there is no back edge from w or any descendant of w to a proper ancestor of v.*
2. *Leafs of T are not articulation points.*
3. *The root of T is an articulation point if and only if it has at least two children.*

Proof. 1. Suppose v is an articulation point and each child w_i of v has a descendant x_i with a back edge to a proper ancestor u_i of v. Then $v, w_i, \dots x_i, u_i, v$, where ... denotes a path of tree edges, is a cycle. Therefore, removing v leaves G connected. ↯

Let w be a child of v such that w and its descendants have no back edges to any proper ancestor of v and let u be a proper ancestor of v. G is connected, so there exists a path from u to w. Since there are no back edges between the ancestors of v and the subtree of w and no edges from the subtree rooted at w to any other subtree of v (Lemma 2.5), every u - w path must pass through v. It follows that there are no remaining u - w paths in G if v is removed, so v is an articulation point.

2. Let vertex l be a leaf of T. T is a subgraph of G and connected by definition, so l must have at least one edge in G. If l has exactly one edge, it must be the edge to its parent in T. The vertex l and this edge can be removed without disconnecting part of G. Suppose that there exists an edge $e = (l, u) \notin T$. Since l has no descendants, there are two possible cases for u:

   - u is not an ancestor of l. In that case there must be a subtree of T rooted at a certain vertex x such that u and l are in different subtrees of x. Applying Lemma 2.5 to x shows that there can be no edges between u and l, so e cannot exist. ↯
   - u is a proper ancestor of l. T is connected, so there is a path p between u and l in T. This means that $e + p$ is a cycle in G, so removing l leaves G connected.

3. If root r has no children it is an isolated vertex, so it can be removed without increasing the number of components. If r has one child, treat it as a leaf and apply part 2. In both cases r is not an articulation point.

   Let a and b be two distinct children of r. From Lemma 2.5, it follows that there are no edges between the subtrees rooted at a and b. Then the only path in G between a and b is $p = a, r, b$, so removing r splits G into components T_a and T_b.

□

From Lemma 4.6 an algorithm can be constructed that finds all the articulation points of a graph. We do a regular DFS on G with an additional variable *low* for each vertex. When a vertex u is discovered, $low[u]$ is set to the discovery time of u, $time[u]$. Let v be a child of u. When the DFS has finished processing v, $low[u]$ is updated as follows:

- If v was visited for the first time, update $low[u] = min(low[u], low[v])$.
- If v had been visited before, is not u's parent and was discovered before u, update $low[u] = min(low[u], time[v])$.

Note that these updates correspond to (u, v) being a tree edge and a back edge respectively. They are implemented on lines 7,10 and 16,18 in Algorithm 2.

Lemma 4.7. *Let $G = (V, E)$ be a graph. At the end of the DFS, $low[u]$ equals the lowest discovery time of all vertices that can be reached via a back edge from a descendant of u for each $u \in V$.*

Proof. Let $T = (V, E_T)$ be the DFS tree of G.

If u is a leaf of T, it has no descendants, so each adjacent edge in G, apart from the edge to its parent, is a back edge. Therefore, for each adjacent vertex v that is not u's parent we update $low[u] = min(low[u], time[v])$. Initially, $low[u] = time[u]$, so $low[u] = min(time[u], min_v(time[v]))$, which equals the lowest discovery time of all vertices that can be reached via a back edge from u. Since u has no proper descendants, this is equivalent to the lowest discovery time of all vertices that can be reached via a back edge from a descendant of u.

Suppose that u is not a leaf of T and that for each descendant v of u it holds that $low[v]$ equals the lowest discovery time of all vertices that can be reached via a back edge from a descendant of v. For each back edge of u, we update $low[u] = min(low[u], time[v])$ and for each tree edge, we update $low[u] = min(low[u], low[v])$, so in total we have

$$low[u] = \min(time[u], \min_{v \in \{v|(u,v) \in E_T\}} low[v], \min_{v \in \{v|(u,v) \notin E_T\}} time[v]),$$

which equals the lowest discovery time of all vertices reachable through back edges from u and back edges from descendants of u's children. In other words: $low[u]$ equals the lowest discovery time of all vertices that can be reached via a back edge from a descendant of u. □

From Lemma 4.6 and Lemma 4.7, it follows that vertex u is an articulation point if and only if u has a child v such that $low[v] \geq time[u]$. This means we can extend the DFS to find articulation points by checking for each child v of a vertex u whether $low[v] \geq time[u]$ after the algorithm is done processing v. If so, u should be marked as an articulation point. If u has no children, it is a leaf, so by part 1 of Lemma 4.6 it is not an articulation point. To test part 3 of Lemma 4.6 an additional variable $children[u]$ is needed, which counts the number of children of vertex u. When the DFS finishes processing the root, it checks whether it has 2 or more children and marks it as an articulation point accordingly.

To also find the biconnected components of G we will keep track of the edges the DFS visits using a stack. Suppose the DFS is currently evaluating vertex u. Then for each neighbour v of u, edge $e = (u, v)$ is placed on top of the stack if

1. v has not been visited before
2. v has been visited before, is not u's parent and was discovered before u.

Lemma 4.8 below shows that these criteria guarantee that each edge is placed on the stack exactly once. Line 7-8 and 16-17 of Algorithm 2 implement the stack procedure described above.

Lemma 4.8. *Each edge of G is put on the stack exactly once during the execution of Algorithm 2.*

Proof. Criteria 1 and 2 to put an edge e on the stack correspond to e being a tree edge and a back edge respectively. DFS partitions the edges of G into tree edges and back edges, so each edge must be placed on the stack at least once.

A DFS examines all neighbours exactly once for each node in G. This means each edge is traversed exactly two times, once from each incident vertex, so it can be added to the stack at most twice. Suppose $e = (u, v)$ is an edge in G, then one of the following must be true.

- If e is a tree edge, assume without loss of generality that $time[u] < time[v]$. Then e is first added to the stack when the DFS is examining the neighbours of u and v has never been visited before. After that, both u and v have been visited, so criterion 1 cannot be satisfied again.
- If e is a back edge, only criterion 2 applies. When the DFS is examining the neighbours of vertex u, e might be added to the stack because v has been visited before, is not u's parent and was discovered before u. When the DFS is examining the neighbours of vertex v, e might be added to the stack because u has been visited before, is not v's parent and was discovered before v. If both events occur, this implies that $time[v] < time[u]$ and $time[u] < time[v]$ ↯.

□

When the DFS is done visiting neighbour v of a vertex u, it checks whether $low[v] \geq time[u]$. If this is the case, all edges up and including (u, v) are popped from the stack. Together with the adjacent vertices, these edges form a biconnected component. Isolated vertices form biconnected components without any edges. To make sure that these are still detected by the algorithm, we check for each vertex whether it has no parent and no children. If so, it is added to the list of biconnected components. Note that this step could be omitted, since an isolated vertex never compromises planarity. The steps to find articulation points and biconnected components are combined in Algorithm 2. Theorem 4.9 proves that it correctly partitions the edges of an arbitrary graph into biconnected components.

Theorem 4.9 ([Hopcroft and Tarjan, 1973])**.** *Algorithm 2 correctly assigns each edge of a graph $G = (V, E)$ to a biconnected component.*

Proof. Suppose G has no edges. Then for each vertex v in G, the for loop starting on line 6 is not executed, so the algorithm correctly finds the biconnected components of G.

Suppose the algorithm works correctly for all graphs with $n - 1$ edges. Let $G = (V.E)$ be a graph with n edges and let r be the root of the DFS tree of G. For each child v of r it holds that $time[r] = 0 \leq low[v]$, so for each child of the root line 11 is executed and the stack is emptied up and including (r, v). As a result, the stack must be completely empty at the end of the algorithm. From Lemma 4.8 we know that each edge appears on the stack exactly once, so each edge is assigned to exactly one biconnected component at the end of the algorithm.

Let u be the vertex at which line 11 of the algorithm is first executed, i.e. the vertex at which edges are popped from the stack for the first time.

Let E_1 denote the set of edges that were popped from the stack, let E_{2a} denote the edges that remain on the stack and let E_{2b} denote the remaining edges. Define $E_2 = E_{2a} \cup E_{2b}$. Then u is the articulation point that separates E_1 from E_2. Define $V_i = \{v \in V | \exists_{w \in V} : (v, w) \in E_i\}$ for $i = 1, 2$ and define graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Note that $u \in V_1$ and $u \in V_2$. We can distinguish two cases based on the size of E_2.

- Suppose $E_2 = \emptyset$, i.e. all edges of G are popped from the stack at once. Suppose G is not biconnected. As noted before, line 11 is executed for each child of the root, so the root cannot have more than 1 child, otherwise edges would be popped from the stack on more than one occasion. From Lemma 4.6 it follows that root r is not an articulation point. Then there exists a biconnected component in G that does not contain r and that can be disconnected from all other parts of the graph by removing a single articulation point $p \neq r$. Call this component $B = (V_B, E_B)$ and define $G' = ((V \setminus V_B) \cup p, E \setminus E_B)$.

  Assume the order in which the algorithm visits the vertices of B, G' and G to be consistent. Then sequence of steps on G is the same as the sequence of steps on B, until an edge between p and a vertex in G' is traversed by the algorithm. After that, the behaviour of the algorithm is the same as when executed on B starting at vertex p. The remaining steps on G are the same as the remaining steps on G'. This means that the total sequence of steps executed on G is the composite of the algorithm's behaviour on B and G'. However, since p is an articulation point for biconnected component B, $low[p] \geq time[v]$ for a child $v \in B$ of p. This causes the algorithm to pop all edges from the stack when it returns to p from v. But for the root r we also have that $low[r] \geq time[w]$ for each child w of the root, so edges are popped from the stack at least twice when the algorithm is executed for G↯.

  Then it follows that if all edges of G are removed from the stack at the same time, G is a biconnected graph, so the behaviour of the algorithm is correct in that case.

- Suppose $E_2 \neq \emptyset$. Since all edges in E_1 are popped from the stack at once, it follows from the proof above that G_1 is biconnected. This means that every biconnected component of G is a biconnected component of G_1 or a biconnected component of G_2. Graphs G_1 and G_2 contain on at most $n - 1$ lines, so the algorithm works correctly for both of them.

  Assume once again that the vertex numbering of G_1, G_2 and G is consistent, then the order in which the algorithm visits the vertices of G_1, G_2 and G is also consistent. The sequence of steps taken by the algorithm on G_1 can be split into a part where the edges of E_1 are placed on the stack and a part when the edges of E_1 are popped from the stack. Similarly, the sequence of steps taken by the algorithm

on G_2 can be split into an initial part that places the edges of E_{2a} on the stack and a sequence of remaining steps. Call these sequences of steps $S_{E_{2a}}$ and S_r. The sequence of steps taken on G' consists of $S_{E_{2a}}$, followed by the steps taken on G_1 starting at u, followed by S_r. This is simply the composite of the algorithm's behaviour on G_1 and G_2, so the algorithm must also operate correctly on G.

□

Algorithm 2: findBiconnectedComponents

input : A graph $G = (V, E)$
output: A list of graphs representing the biconnected components of G

```
1  dfsBiComp(G, u, parent)
2      mark u as visited
3      low[u] = time[u] = time
4      increment time
5      children[u] = 0
6      for each vertex v adjacent to u that is not u's parent do
7          if v has not been visited then
8              add (u, v) to the stack
9              dfsBiComp(G, v, u)
10             low[u] = min(low[u], low[v])
11             if low[v] ≥ time[u] then
12                 mark u as an articulation point
13                 create new biconnected component B
14                 pop all edges up and including (u, v) from the stack and add them to B
15             increment children[u]
16         else if v ≠ parent and time[v] < time[u] then
17             add (u, v) to the stack
18             low[u] = min(low[u], time[v])
19     if u has no parent then
20         if children[u] ≥ 2 then
21             mark u as an articulation point
22         if children[u] == 0 then
23             create new biconnected component B
24             add u to B
25 findBiconnectedComponents(G)
26     time = 0
27     for each vertex u in V do
28         if u has not been visited then
29             dfsBiComp(G, u, -1)
30
```

Once all biconnected components have been identified, the algorithm executes the steps below for each of them separately.

ii. a) Cycle and segments

The next step of the algorithm finds a cycle in a biconnected component B. Recall from Lemma 4.2 that B must contain at least one cycle if it contains three or more vertices. If not, B contains at most one edge, so it must be planar. In that case the algorithm returns 'planar' and skips the next steps for this component.

Lemma 4.10. *A graph $G = (V, E)$ contains a cycle if and only if a DFS on G finds a back edge.*

Proof. Let $(u, v) \in E$ be a back edge. The DFS tree T of G is connected by definition, so there is a path p between u and v in T. Then p and (u, v) form a cycle in G.

Suppose G contains a cycle $v_0, v_1, \ldots, v_k = v_0$. Let v_0 be the first vertex from this cycle that is visited in the DFS. $v_1, v_2, \ldots, v_{k-1}$ can be reached from v_0, so they are descendants of v_0 in the DFS tree. Then it follows from Lemma 2.5 that either (v_0, v_{k-1}) or (v_0, v_1) is a back edge. □

If B contains a cycle it can be found by running a DFS until a back edge is encountered. Lemma 4.10 proves that a back edge in a DFS indicates the presence of a cycle. Suppose a back edge (u, v) is found from node u. Then we can construct the corresponding cycle by tracing back through the recursive steps of the DFS and saving the vertices we visit until we arrive at vertex v. Since each node is only recursed on once during a DFS, it is guaranteed that we find a simple cycle.

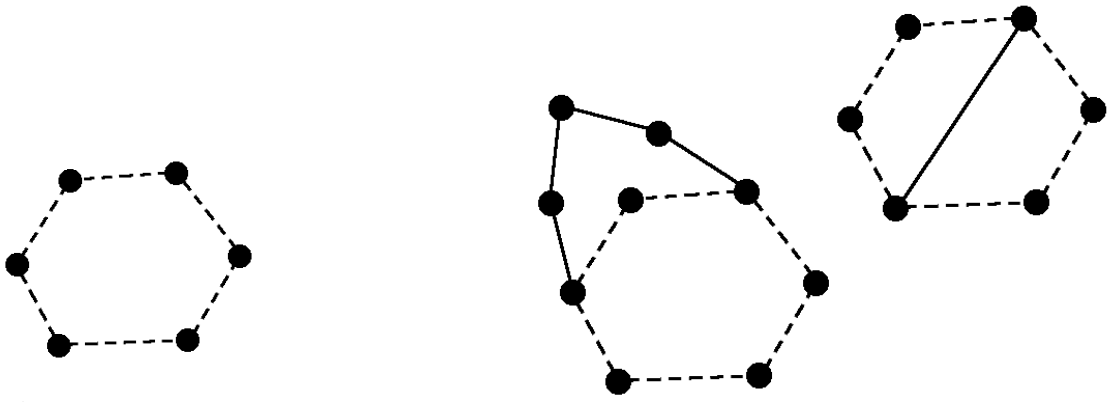
Finding the segments attached to the cycle C can also be done with a DFS, but requires a number of adaptations:

- Before each initial call of DFS, create an empty segment. During the DFS, add the traversed vertices and edges to this segment.
- If a vertex u is contained in C, do not visit its neighbours. This prevents edges in C from being added to a segment. If u does not have a parent in the DFS tree, terminate the DFS. If it does, u and the edge $(parent[u], u)$ should be added to the current segment. In the latter case, u is an attachment of the segment and should be marked as such.
- Due to the requirement above, chords will not be found. They only contain vertices from C and these are not recursed on. A separate method should be added that checks for each edge in B whether it is a chord.

Algorithm 3 implements these adaptations. It is started with an initial call to $findSegments$.
There are two base cases in which the algorithm does not need to continue after finding the segments:

- C has no segments. Then cycle C forms the entire biconnected component, so it is planar.
- C has one segment which is a path. Embedding this path entirely in either the interior or exterior region of C results in a planar drawing of the biconnected component, so it is planar.

Planarity is guaranteed in both cases, so no further computations are needed. The current invocation of the algorithm is terminated. Figure 6 shows an example of each base case.



(a) A cycle with no segments.

(b) Two cycles whose only segment is a path.

Figure 6: An illustration of the two base cases of the Auslander-Parter algorithm.

Algorithm 3: findSegments

input : Biconnected graph $G = (V, E)$; cycle C
output: A list of the segments attached to C

```
1  dfsSegments(G, C, u, parent)
2      mark u as visited
3      if u ∈ C then
4          if u has a parent then
5              if S does not yet contain u then
6                  add u to S
7              mark u as an attachment of S
8              add edge (parent, u) to S
9          return
10     add u to S
11     if u has a parent then
12         add edge (parent, u) to S
13     for each vertex v adjacent to u do
14         if v has not been visited or v ∈ C then
15             dfsSegments(G, S, v, u)
16         else if S does not contain edge (u, v) then
17             add edge (u, v) to S
18 findChords(G, C)
19     for each edge e = (u, v) in E do
20         if u ∈ C and v ∈ C then
21             if u and v are not adjacent in C then
22                 create a new segment S
23                 add u, v and e to S
24                 mark u and v as attachments of S
25                 add S to the list of segments
26 findSegments(G, C)
27     for each vertex u in V do
28         if u has not been visited then
29             create an empty segment S
30             dfsSegments(G, C, u, -1)
31             if S is not empty then
32                 add S to the list of segments
33     findChords(G, C)
```

ii. b) Interlacement graph

Once a cycle and its segments have been found, a method is needed to establish whether there are any conflicting segments. This can be done efficiently by executing the following procedure for each segment S_i:

- Let k be the number of attachments of S_i. Traverse through C and assign a label to each vertex as follows:
  - Label attachments $a_0, \ldots, a_{k-1}$ with $0, 2, \ldots, 2(k-1)$. The index denotes the order in which the attachments are encountered while traversing the cycle.
  - Let $v \in C$ be a cycle vertex that is not an attachment of S_i. If v lies between a_j and a_{j+1}, v is labelled $2j + 1$.

  We will refer to this labelling procedure as *makeInterlacementGraph*.
- For each segment $S_j \neq S_i$, check whether the labels of their attachments are in the set $\{2i, 2i+1, 2i+2 \bmod 2k\}$ for a certain $i \in \{0, 1, \ldots, k-1\}$. If not, the attachments of S_i and S_j interleave, so they are conflicting segments.

The last step can be implemented as follows:

Make an array of zeroes of length $2k$, where k is the number of attachments of S_i. The elements of the array correspond to the numbers $0, 1, \ldots, 2k-1$, which are the labels assigned to the cycle vertices for segment S_i. For each attachment of S_j, look up the label assigned to this vertex and replace the corresponding array element with 1. Let s be the sum of the values in the array. To check whether the labels of the attachments are in the set $\{2i, 2i+1, 2i+2 \mod 2k\}$ for some i, we now loop over i and check whether the sum of the subarray $\{2i, 2i+1, 2i+2\}$ is equal to s. If this is not the case for any $i \in \{0, 1, \ldots, k-1\}$, we have found a conflicting pair of segments. Algorithm 4 shows the pseudo code of this conflict check.

Algorithm 4: checkConflict

input : Segment S_j; hashmap `cycleLabels(`*cycleLabels*`)` containing cycle vertex - label pairs from segment S_i; k, the number of attachments of S_i

output: A boolean indicating whether segment S_i and S_j conflict

```
checkConflict(S_j, cycleLabels, k)
    initialise array labels of size 2k
    fill labels with 0's
    sum = 0
    for each attachment a of S_j do
        labels[cycleLabels(a)] = 1
    for i from labels do
        sum = sum + labels[i]
    partSum = labels[0] + labels[1] + labels[2]
    for i in {0, 2, ..., 2k − 2} do
        if partSum == sum then
            return false
        end
        partSum = partSum + labels[(3 + i) mod 2k] + labels[(4 + i) mod 2k]
        partSum = partSum − labels[i] − labels[(1 + i) mod 2k]
    return true;
```

The found conflicts can then be stored in an interlacement graph.

ii. c) Testing for bipartiteness

Testing whether the interlacement graph of B's segments is bipartite can be done with a single graph traversal. The algorithm uses a BFS to traverse the graph and assigns a 0 or 1 to each vertex when it is first discovered. The root is marked with 0 and all other vertices are assigned a different label from their parent in the BFS tree. After assigning a label to a vertex, we check whether its neighbours all have a different label than this vertex. If not, an odd cycle has been found and the interlacement graph is not bipartite. If the algorithm terminates without finding any conflicting neighbours, then the vertices labelled 0 and the vertices labelled 1 form a bipartition of G.

Algorithm 5 shows the implementation of this method and Lemma 4.11 argues its correctness.

Lemma 4.11. *Algorithm 5 correctly outputs whether an arbitrary graph G is bipartite or not.*

Proof. Suppose G is bipartite. Algorithm 5 assigns 0 to the root, 1 to its neighbours, 0 to their neighbours, etc. By Theorem 2.2, G contains no odd cycles, so there cannot be neighbouring vertices with the same label. Therefore, the algorithm will return that the graph is bipartite.

Suppose that G is not bipartite. Then it follows from Theorem 2.2 that G contains an odd cycle. Since the algorithm assigns labels to each vertex, there must be two adjacent vertices in this odd cycle that have the same label. Note that a BFS visits all vertices and for each vertex the algorithm checks all adjacent vertices for conflicts, so if there is a conflicting label, it will always be found. Therefore the algorithm will correctly return that G is not bipartite. □

Algorithm 5: testBipartite

input : Interlacement graph $G = (V, E)$
output: A list of the segments attached to C

```
1  bfsBipartite(G, u, labels)
2      labels[u] = 0
3      add u to the queue
4      while the queue is not empty do
5          v is the first element from the queue
6          for each vertex w adjacent to v do
7              if labels[w] equals −1 then
8                  labels[w] = 1 − labels[v]
9                  add w to the queue
10             else if labels[w] == labels[v] then
11                 return false
12     return true
13 testBipartite(G)
14     fill labels with −1
15     for each vertex u in V do
16         if labels[u] == −1 then
17             bipartite = testBipartite(G, u, labels)
18             if bipartite is false then
19                 return false
20     return true
```

ii. d) Recursion

If biconnected component B is not a base case and its segments do not conflict we still do not know whether it is planar. It follows that the current cycle and its segments can be embedded in the plane if all segments are planar, but this might not be the case.

To find out, the algorithm recurses on each subgraph consisting of one segment and the cycle as illustrated in Figure 7.

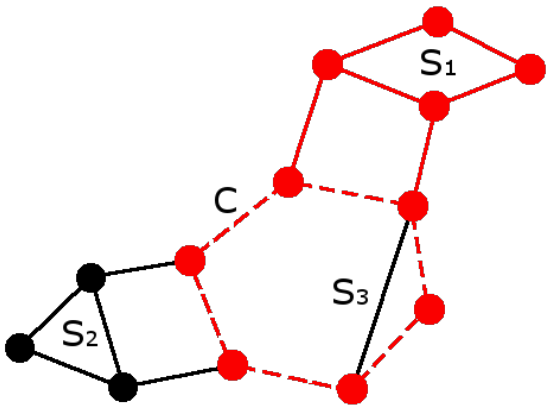


Figure 7: A biconnected graph with cycle C, indicated by dotted lines. Recursion on segment S_1 will use the subgraph marked in red.

Recall that in step ii.a) the algorithm computed an arbitrary cycle in the biconnected component. To ensure that the algorithm terminates after a finite number of recursions, each recursive invocation computes a specific cycle based on the cycle of its parent invocation. To this end, we will define a cyclic order on the vertices in a cycle. It does not matter which of the two possible orders is chosen, as long as it is used consistently throughout the algorithm. Below, the word 'positive' is used to refer to this order. Vertices u

and v are said to be positive consecutive in cycle C if after u one reaches v before reaching any other vertex when traversing C in positive direction.

Suppose that the algorithm recurses on cycle C and segment S, then the new cycle is constructed as follows.

- Choose two positive consecutive attachments a_1 and a_2 of S and find a path p from a_1 to a_2 in S. Because of biconnectivity these always exist
- Find a path in negative direction from a_2 to a_1 in C and remove it from C
- Add p to C.

Figure 8 illustrates this procedure. The algorithm then computes the segments of the new cycle and proceeds as usual.

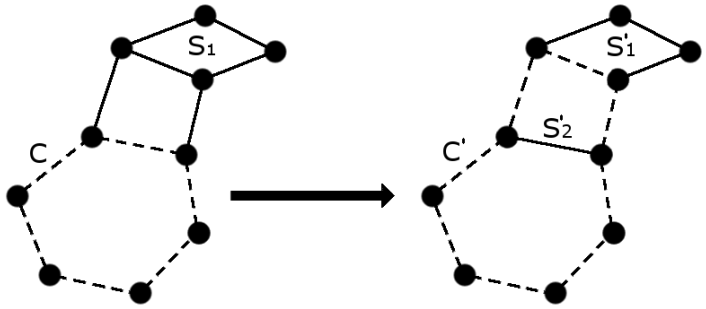


Figure 8: Finding a new cycle in the graph from Figure 7.

Note that Figure 8 concerns a segment with exactly two attachments, so a_1 and a_2 are positive consecutive by default. However, it should be stressed that if the segment has more than two attachments, the algorithm must always choose two consecutive ones to augment the circuit. The importance of this condition is illustrated by the following example.

Consider the biconnected graph pictured on the left in Figure 9. Its cycle has one segment with three attachments a_1, a_2, a_3. If we pick non-consecutive attachments a_1 and a_3 to augment the cycle, we obtain the graph on the right of Figure 9. This graph once again has one segment with attachments a_1, a_2' and a_3. If we replace the top part of the new cycle C' with a positive path between a_1 and a_3, we are back at the original graph and have made no progress. The algorithm could get stuck in such a loop of augmentations and never terminate.

From Lemma 4.13 below, it follows that the algorithm must end after a finite number of recursions if the above method is used to compute new cycles.

Lemma 4.12 ([Tamassia, 2013])**.** *Let C be a cycle with a single segment S which is not a path. If we use the method illustrated in Figure 8 to generate a new cycle from C and S, then:*

1. *we obtain a cycle with at least two segments*
2. *the part of C that is replaced, forms a path segment of C'.*

Proof. Let a_1 and a_2 be two positive consecutive attachments of S in C. Let p be a path in S from a_1 to a_2, which must exist due to biconnectivity and let q be a positive path in C from a_1 to a_2 such that q does not contain any other attachments of S. Suppose C' is the cycle obtained by replacing q with p, then q is a segment of C' with attachments a_1 and a_2. Since q contains no attachments of S apart from a_1 and a_2, this new segment must be a path (2).

If S is not a path, there must be an edge $e \in S$ that is not contained in p, and thus is part of a segment. Since q is a path and $e \notin q$, e and q must belong to different segments (1). □

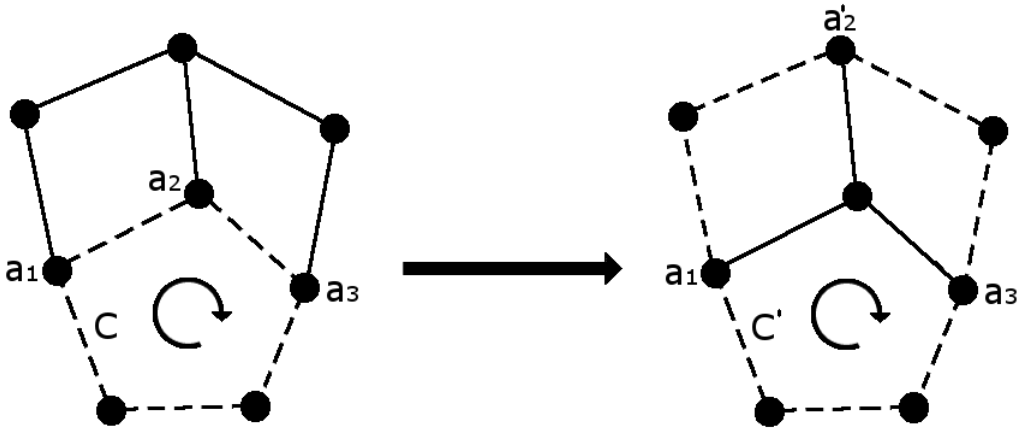


Figure 9: Augmenting the cycle of the left graph with a path between positive non-consecutive attachments a_1 and a_3 results in the graph on the right. Augmenting the cycle of the graph on the right with a path between a_1 and a_3 could result in the original graph on the left. This would cause the algorithm to get stuck in an infinite loop.

Lemma 4.13. *Let B be a biconnected component with m edges. Then the Auslander-Parter algorithm terminates after at most m recursive invocations when evaluating B.*

Proof. We will prove this lemma by showing that a unique edge can be assigned to each recursive invocation of the algorithm.

Let C and S be a cycle and segment that are recursed on, and C' be the cycle of the next recursive invocation. Find an edge e such that $e \in C'$ and $e \notin C$. Due to the way C' is constructed such an edge must always exist. Moreover, it is impossible to choose the same edge twice:

Recall that a new cycle C' is constructed by replacing a part of C, call this q, between two positive consecutive attachments of S with a path p in S between these attachments. From part 2 of Lemma 4.12 it follows that q is a segment of C' and that it is a path. This means that the recursive call on $C' + q$ finds the base case depicted in Figure 6b and halts. As a result, all edges contained in q cannot be added to a cycle at any later recursion. Then each edge from q can be chosen as e at most once.

If there are multiple choices for e it can be chosen at random.

We now have that each invocation of AP is associated with a unique edge, so there can be no more than m recursive invocations. □

Now that we have defined the recursive part of the algorithm, we can combine all subalgorithms from Section 4.2 into one main method. Algorithm 6 outlines the recursive part of the Auslander-Parter algorithm and shows when each subalgorithm is executed. It is started by running the method *compute* with a graph as a parameter and returns a boolean value representing whether the given graph is planar.

Algorithm 6: Auslander-Parter

input : A graph G
output: A boolean indicating whether G is planar

```
1  recurse(B, C)
2      if |C| == 0 then
3          return true
4      get list of segments segs = findSegments(B, C)
5      if |segs| == 0 then
6          return true
7      if |segs| == 1 and δv ≤ 2 for all vertices v in S ∈ segs then
8          return true
9      create interlacement graph G_I = makeInterlacementGraph(segs, C)
10     if testBipartite(G_I) == false then
11         return false
12     for each S ∈ segs do
13         create graph B' = cycle + B
14         if |E_B| > 3 · |V_B| − 6 and |V_B| > 2 then
15             return false
16         cycle C' = augmentCycle(B', C, S)
17         if recurse(B', C') == false then
18             return false
19     return true
20 compute(G)
21     get list of biconnected components biconnectedComps = findBiconnectedComponents(G)
22     for each B = (V_B, E_B) ∈ biconnectedComps do
23         if |E_B| > 3 · |V_B| − 6 and |V_B| > 2 then
24             return false
25         cycle C = getCycle(B)
26         if recurse(B, C) == false then
27             return false
```

4.3 Running time

The Auslander-Parter algorithm is claimed to run in $O(|V|^3)$. This section is dedicated to the proof of this running time. We will break this down into the same steps as the algorithm.

i. Biconnected components
In Algorithm 2, each vertex is marked as visited after processing and visited vertices are skipped, so each vertex is visited at most once. The loop starting on line 22 ensures that each vertex is visited exactly once. For each vertex u, the following operations are executed:

- Lines 2-5, 19-21 describe elementary operations that can be done in $O(1)$.
- Lines 14 pops all edges up and including (u, v) from the stack. Recall from Lemma 4.8 that each edge is put on the stack exactly once, so in total, this results in $O(E)$ operations.
- Lines 7-8, 10-13 and 15-18 describe elementary operations for each vertex adjacent to u, which takes $\sum_{u \in V} O(\delta_u) = O(|E|)$ time for all vertices in total.

In total this gives a running time of $|V| \cdot O(1) + O(|E|) + O(|E|) = O(|V|) + O(|E|) = O(|V| + |E|)$. Note that Algorithm 2 only finds the edges for each biconnected component. When an edge is added to a biconnected component, the two adjacent vertices must also be present in that component. Verifying whether these are

present and adding them if this is not the case can be done in constant time for a single edge. For all edges, this amounts to $O(|E|)$, so it does not influence the total running time.

The remaining part of the algorithm is applied to each biconnected component separately. If such a component is planar, we may apply Lemma 4.14 below. This allows us to express the running time of AP for each planar biconnected component in the number of vertices only.

Lemma 4.14. *$O(|V|) = O(|E|)$ for any connected planar graph $G = (V, E)$.*

Proof. From Lemma 2.4 it follows that $|E| \leq 3 \cdot |V| - 6$. Then $|E| = O(|V|)$.
Any connected graph must have at least $|V| - 1$ edges, so we also have $|V| = O(|E|)$.
Then for any connected planar graph $O(|V|) = O(|E|)$. □

When testing for planarity it is not known in advance whether the a biconnected component is planar. We can ensure that Lemma 4.14 still holds by adding a simple test to the algorithm: for each biconnected component $B = (V_B, E_B)$, check whether $|E_B| \leq 3 \cdot |V_B| - 6$. If not, the component cannot be planar according to Lemma 2.4, so the algorithm need not recurse on it. If it does satisfy this condition then $O(|V_B|) = O(|E_B|)$.

The following claims all apply to an invocation of the algorithm on a biconnected component B that satisfies the above condition. The number of vertices and edges in this component will be denoted as n and m respectively.

ii. a) Cycle and segments

A cycle in a biconnected component can be found with a DFS, so from Lemmas 2.6 and 4.14 it follows that this can be done in $O(n)$ time.

Algorithm 3 consists of a DFS-based function to find the vertices of each segment and a function to find all chords. Lines 1-17 and 26-32 describe a DFS with a few extra operations that take constant time. According to Lemma 2.6 this takes $O(n + m) = O(n)$ time. The function *findChords* loops over all edges of B and processes them in constant time, so it runs in $O(m) = O(n)$ time.

ii. b) Interlacement graph

To generate an interlacement graph from a cycle C and its segments $S_1, \ldots, S_{nc}$, we assign labels to each cycle vertex and check for conflicts between each pair of segments using Algorithm 4.

To assign labels to the cycle vertices, the cycle is traversed once per segment, which results in a running time of $O(|C|)$ per segment. Checking for a conflict between two distinct segments S_i and S_j requires a loop over all k_j attachments of S_j (Alg. 4, lines 5-6) and three loops over all label values induced by S_i (Alg. 4, lines 3, 7-8 and 10-15). All other operations are elementary. This results in a total running time of

$$\begin{aligned}
\sum_{i=1}^{n_c}(O(|C|) + \sum_{j=1}^{n_c}(O(k_j) + 3 \cdot O(2k_i))) &= \sum_{i=1}^{n_c} O(|C|) + \sum_{i=1}^{n_c}\sum_{j=1}^{n_c} O(k_j) + \sum_{i=1}^{n_c}\sum_{j=1}^{n_c} O(k_i) \\
&= n_c \cdot O(|C|) + n_c \cdot \sum_{j=1}^{n_c} O(k_j) + n_c \cdot \sum_{i=1}^{n_c} O(k_i) \\
&\overset{*}{=} n_c \cdot O(|C|) + n_c \cdot O(n_c + m) + n_c \cdot O(n_c + m) \\
&\overset{**}{=} m \cdot O(n) + m \cdot O(m) + m \cdot O(m) \\
&= O(n^2).
\end{aligned}$$

To see equality *, we use the fact that a segment with l edges can have at most $l + 1$ attachments. Since all segments are edge-disjoint, the sum of the number of attachments over all segments can be no larger than $n_s + m$. To see **, note that $|C| \leq n$ and $n_s \leq m$.

ii. c) Testing for bipartiteness
The algorithm to test for bipartiteness is based on a BFS. Colouring nodes and testing for conflicts can be done in constant time, so this amounts to a worst-case running time of $O(\#segments + \#conflicts) = O(\#segments^2)$. Note that the algorithm terminates if a conflict is found, so in practice its average running time might be better than that of a regular BFS. Initialising the colour array in Algorithm 5 takes $O(\#segments)$ time. Since each segment contains at least one edge, it follows that $O(\#segments^2) = O(m^2) = O(n^2)$.

ii. d) Recursion
If the biconnected component currently examined by the algorithm is not a base case, a new biconnected component is formed and a new cycle computed for each segment. This result in the following running times:

- Creating a new biconnected component requires all but one segments to be removed from the old component. This takes $O(n)$ time.
- Finding a new cycle requires finding a path in both the current cycle and a segment. The path in the cycle C can be found in $O(|C|) = O(n)$. Finding a path in the segment an be done with a DFS and hence takes $O(n + m) = O(n)$ time.

In total, this adds $O(n)$ to the running time.

So far, we have only considered the running time of all subalgorithms of a single AP invocation. Combining Section ii.a-d we obtain a total running time of $O(n) + O(n^2) + O(n^2) + O(n)$. To obtain the total running time for a single biconnected component, the number of recursive invocations needs to be taken into account. It follows directly from Lemma 4.13 and 4.14 that the algorithm terminates after at most $O(n)$ invocations. This means that in total the Auslander-Parter algorithm runs in $O(n) \cdot O(n^2) = O(n^3)$ for a biconnected component with n vertices.

Let graph G have k biconnected components $B_1, \dots B_k$ with $n_1, \dots, n_k$ vertices respectively. Then the running time of B_i equals $O(n_i^3)$. Therefore, the total running time for the entire graph is given by $O(|V| + |E|) + O(n_1^3 + \cdots + n_k^3) = O(|V| + |E|) + O((n_1 + \cdots + n_k)^3) = O(|V| + |E|) + O(|V|^3) = O(|V|^3)$.

4.4 Implementation

To test and visualise the Auslander-Parter algorithm, the pseudo code from Section 4.2 was implemented in Java. The algorithm uses an existing graph library that provides the basic data structures and operations for the underlying graph. This allows us to focus on the implementation of the algorithm itself and reduces the probability of errors in the core code describing the graph. GraphStream is an open-source Java library that provides a graph data structure, graph visualisation tools and basic graph algorithms. We will use the following functionalities from GraphStream in the implementation of AP:

- A basic data structure to store graphs. A graph object consists of a set of vertices and a set of edges. Vertices and edges can dynamically be added or removed from the graph and are referred to by a unique index. Other functions include returning the number of vertices/edges, looping over all vertices/edges and testing whether a certain vertex/edge exists in the graph.
- Attaching and removing attributes to vertices and edges.
- Displaying a graph on screen with an automatic layout.
- Specifying the colour and size of vertices and edges based on their attributes.

Figure 10 visualises the structure of the program. Each box represents a separate Java class. The upper section of each box contains the name of the class and the lower part contains all public methods of the class and methods that are called automatically when the class is created. Arrows indicate child-parent relations. If there is an arrow from class A to class B, then instances of class B are created from inside class A. The

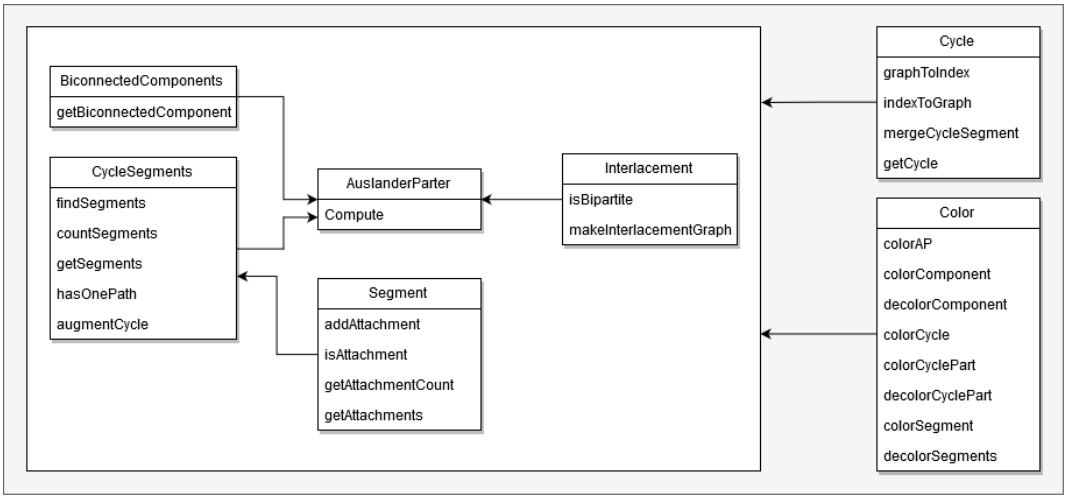


Figure 10: A diagram which shows the class structure of the Java implementation of AP. Each box represents a class. Arrows indicate a child-parent relation.

classes *Color* and *Cycle* are utility classes whose methods can be used directly without creating an object of that class. They are used by the other classes to perform operations on cycles and color graph elements.

Below a more detailed description of each class is provided. The full Java code can be found in Appendix A.

- **AuslanderParter** - The main class which, given a graph, executes the Auslander-Parter algorithm and returns whether the graph is planar or not.
- **BiconnectedComponents** - Class that computes and returns the biconnected components of a graph.
- **Color** - Utility class that can be used for the visualisation of the algorithm. It colours and de-colours paths, cycles, segments and biconnected components and marks articulation points.
- **Cycle** - Utility class that performs cycle algorithms. Given a biconnected component, it can compute a cycle, convert a cycle given by a list of vertex indices to a graph object or convert a cycle represented as a graph to a list of indices. Given a segment S and cycle C it computes graph $S + C$.
- **CycleSegments** - A class that is created for each biconnected component - cycle pair. It computes and returns the segments of the cycle. The class contains methods to deal with the base cases of the algorithm: it counts the number of segments and checks whether there is exactly one segment that is a path. Given a specific segment, it augments the cycle as described in Section 4.2 ii. d.
- **Interlacement** - A class that computes the interlacement graph of the segments of a cycle and tests whether it is bipartite.
- **Segment** - An extension of the GraphStream graph data structure. It adds a list of attachments to the graph. Attachments can be added, removed or counted.

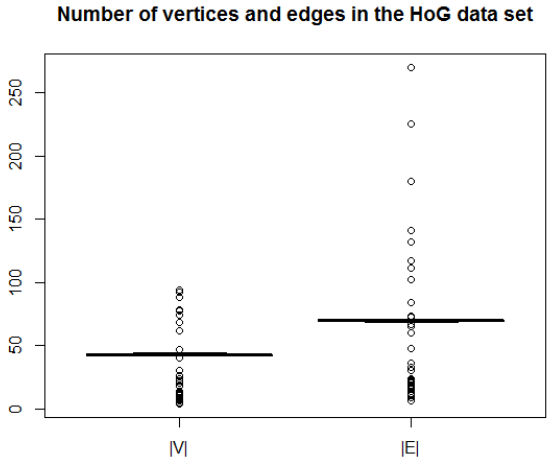
4.5 Testing

To test whether the implementation of the Auslander-Parter algorithm works as intended, a large set of test data is needed with graphs for which it is known whether they are planar or not. Given the running time of the algorithm, these graphs should not contain more than approximately 200 vertices to assure a feasible running time. House of graphs [Brinkmann et al., 2013] has a large collection of graphs that can be searched by many different characteristics such as number of vertices, connectivity and (average) degree. Search results can be exported to a text file as a list of adjacency matrices. Planarity is not one of the

built-in search options, but one can search the description of the graphs for specific words. Searching for 'planar' returns a list of 902 graphs. Figure 11 summarises the composition of this data set. In Table 11a we see that all graphs are connected and box plot 11b shows that the majority of graphs have about 43 vertices and 70 edges, so there is not much variety. Ideally, we would like to test the algorithm on a wider range of graphs.

Number of graphs	902
Number of connected graphs	902
$\overline{\lvert V \rvert}$	43.622
$\overline{\lvert E \rvert}$	70.481
$\sigma(\lvert V \rvert)$	13.366
$\sigma(\lvert E \rvert)$	21.239

(a) Composition of the HoG data set.



(b) A box plot of the number of vertices and edges of the graphs from the HoG data set.

Figure 11: Composition of the data set corresponding to the search term 'planar' from [Brinkmann et al., 2013].

For 901 of the planar test graphs the algorithm returns planar. However, the graph shown in Figure 12 is labelled as non-planar. As it turns out, this is not a mistake by the algorithm. The description of graph 12 contains the phrase "dropping the requirement of planarity", so the graph is not planar. This means that we cannot be certain that the other graphs from this set are planar either, so a different test set is needed to verify the correctness of the algorithm. Section 4.5.1 introduces an algorithm to generate planar graphs, which can be used to create reliable planar test sets. Section 4.5.2 details a method to generate non-planar graphs based on Kuratowski subgraphs.

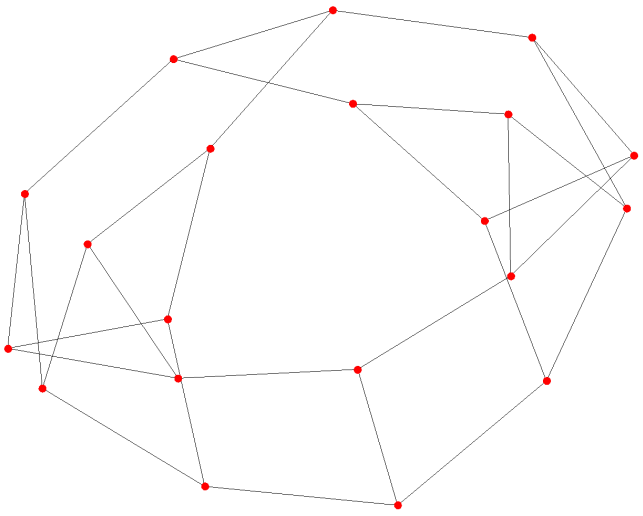


Figure 12: The only graph from the planar test set that is classified by the Auslander-Parter algorithm as not planar.

4.5.1 Generating planar graphs

Recall from Section 2 that a planar graph can be embedded in the plane. We will use this property to reverse-engineer a planar graph. An algorithm given by [Tantalo and Radcliffe, 2007] generates a drawing of lines in the plane and reconstructs a corresponding planar graph. It can be summarised by the following steps:

- Generate a set of random non-parallel lines in the plane.
- Calculate the intersection point of each pair of lines.
- Create a graph with a vertex for each intersection point.
- Add an edge to the graph for each line segment between two intersections.

Since the intersection of any two lines is substituted by a vertex, no intersections remain in the drawing so the corresponding graph is planar.

Lemma 4.15. *Let G be a graph generated generated as described above with n lines. Then G has $\binom{n}{2}$ vertices and $n(n-2)$ edges.*

Proof. Because the lines are not parallel, each line intersects with all of the other lines. These $n-1$ intersections form $n-1$ vertices and each line segment between two consecutive vertices on the line generates an edge. This results in $(n-1)-1 = n-2$ edges on each line, so in total there are $n(n-2)$ edges. Each intersection is created by two distinct lines, so there are $\frac{1}{2}n(n-1) = \binom{n}{2}$ intersections, each of which corresponds to a distinct vertex. □

Each intersection is uniquely identified by two lines. It is easier to refer to a vertex by a single index than by a pair of line indices, so we will use mapping

$$f : (p,q) \to \{1,2,...,n(n-1)/2\} : (p,q) \mapsto f(p,q) = \frac{(p-1)(2n-p)}{2} + q - p$$

to compute a unique index for each vertex. n is the number of lines and $1 \leq p < q \leq n$ are the unique indices of two distinct lines.

Theorem 4.16. *f is a bijection for a fixed n [Tantalo and Radcliffe, 2007].*

Proof. Note that when p is fixed, f is linearly increasing in q. This means the maximum value of f for a fixed p is attained for $q = n$ and the minimum value is attained for $q = p+1$. We can derive that

$$\begin{aligned}
f(p,n) &= \frac{(p-1)(2n-p)}{2} + n - p \\
&= \frac{2np - p^2 - 2n + p + 2n - 2p}{2} \\
&= \frac{2np - p^2 - p}{2} \\
&= \frac{2np - p^2 - p}{2} - 1 + 1 \\
&= \frac{2np - p^2 - p}{2} + (p+2) - (p+1) - 1 \\
&= \frac{p(2n - p - 1)}{2} + (p+2) - (p+1) - 1 \\
&= \frac{((p+1)-1)(2n-(p+1))}{2} + (p+2) - (p+1) - 1 \\
&= f(p+1, p+2) - 1
\end{aligned}$$

so the maximum value of f for a fixed p is one lower than the minimum value of f for $p+1$.

When $p = 1$, f takes on the values $\{1, 2, \ldots n-1\}$. When $p = n-1$, $q > p$ must be n, hence the value of f equals

$$\begin{aligned} f(p,q) &= f(n-1,n) \\ &= \frac{((n-1)-1)(2n-(n-1))}{2} + n - (n-1) \\ &= \frac{(n-2)(n+1)}{2} + 1 \\ &= \frac{n^2-n-2+2}{2} \\ &= \frac{n(n-1)}{2}. \end{aligned}$$

From Lemma 4.15 we know that a graph generated with n lines has exactly $\frac{n(n-1)}{2}$ vertices. Therefore any pair (p, q) with $p < q$ corresponds to exactly one value in $\{1, 2, \ldots n(n-1)/2\}$. □

Algorithm 7 shows the planar graph generation algorithm in more detail.

Algorithm 7: makePlanarGraph

input : Integer n
output: A planar graph on $\binom{n}{2}$ vertices and $n(n-2)$ lines

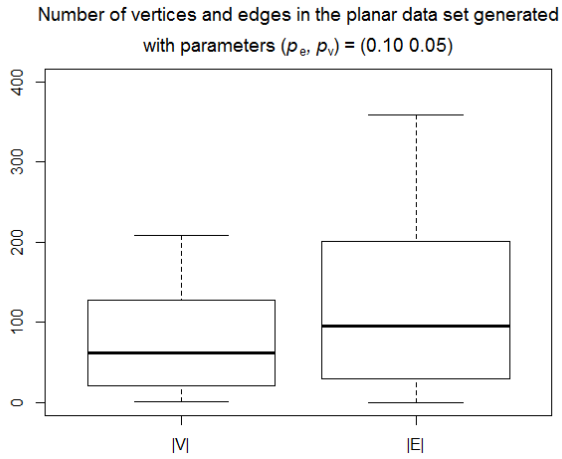
```
1  f(p, q)
2  |  return (p - 1)(2n - p) / 2 + q - p
3  makePlanarGraph(n)
4  |  for i = 1, ..., n do
5  |  |  generate two random points p_1 and p_2 in the plane
6  |  |  calculate slope s_i of the line through p_1 and p_2
7  |  |  while s_i = s_j for j ∈ {1, 2, ... i − 1} do
8  |  |  |  generate two random points p_1 and p_2 in the plane
9  |  |  |  calculate slope s_j of the line through p_1 and p_2
10 |  |  store the line through p_1 and p_2
11 |  create graph G with n(n − 1)/2 vertices
12 |  for each line p do
13 |  |  for each line q do
14 |  |  |  if p equals q then
15 |  |  |  |  skip
16 |  |  |  calculate intersection p and q
17 |  |  sort intersections in ascending order by x coordinate
18 |  |  for each intersection I do
19 |  |  |  if I is the last intersection then
20 |  |  |  |  skip
21 |  |  |  add edge (f(p, I), f(p, I + 1)) to G
```

A limitation of Algorithm 7 is that it only generates graphs with $\binom{n}{2}$ vertices and $n(n-2)$ edges for $n = 2, 3, ...$ lines. This severely limits the types of graphs that can be generated. We will create more variety by changing the number of vertices and edges of the generated graph. Note that planarity may not be preserved when edges are added, so we will restrict ourselves to deleting vertices and edges. After a graph is created with Algorithm 7, we do:

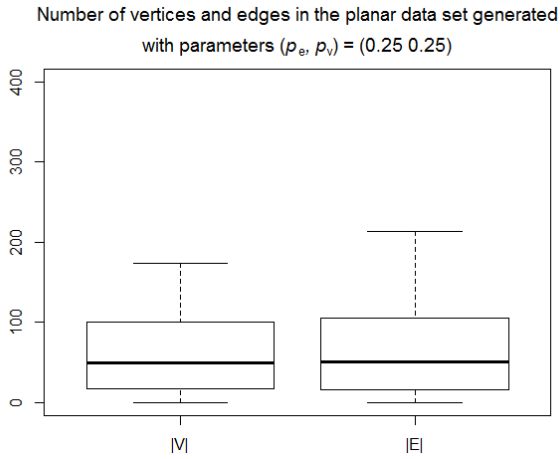
- Remove each edge with probability p_e.
- Remove each vertex with probability p_v.

If a vertex is removed, its edges are also removed from the graph.

To obtain a balanced test set, we set parameters p_e and p_v to 0.10 and 0.05. These values create a large spread in the amount of vertices and edges in the generated graphs. If we choose higher probabilities, for example 0.25 for both p_e and p_v, the number of edges and vertices in the test set decreases significantly. As a result, many sparse graphs consisting of small connected components and isolated points are created. Figure 13 shows the comparison between these parameter values.



(a) Composition of the planar data set generated with parameters $\{p_e, p_v\} = \{0.10, 0.05\}$.



(b) Composition of the planar data set generated with parameters $\{p_e, p_v\} = \{0.25, 0.25\}$.

Figure 13: A box plot of the number of vertices and edges in a planar dataset generated with different parameters.

4.5.2 Generating non-planar graphs

Recall that a graph is planar if and only if it does not contain a Kuratowski subdivision. This means that if we start with a Kuratowski graph and randomly subdivide edgs, the graph remains non-planar. Adding vertices and edges also does not compromise non-planarity. This can be combined into the following graph generation algorithm:

- With probability p_{K5} create Kuratowski graph K_5, otherwise create $K_{3,3}$.
- Generate a random integer *steps* between *minSteps* and *maxSteps*.
- Repeat *steps* times:
  - Generate a random number r between 0 and 1.
  - If $r < p_{addE}$, pick two random vertices u, v in the graph and add edge (u, v) if it doesn't already exist.
  - If $p_{addE} \leq r < p_{splitE}$, remove a random edge $e = (u, v)$ from the graph and replace it with a new vertex w and edges (u, w), (w, v).
  - Otherwise, add a new vertex u to the graph and generate a random number p_E between $p_{min} >= 0$ and $p_{max} <= 1$. For each vertex $v \neq u$, add edge (u, v) to the graph with probability p_E.

The choice of parameters $p_{K5}, minSteps, maxSteps, p_{addE}, p_{splitE}, p_{min}$ and p_{max} greatly influences the structure of the graphs that are generated. To thoroughly test the algorithm we will need to tune them

such that we get a varied data set that tests all aspects of the algorithm. We need to take into account the following aspects.

Recall that the AP algorithm terminates if for any biconnected component we have $|E| > 3|V| - 6$. This means the rest of the algorithm is not tested in such a case. Therefore we should limit the amount of graphs in the generated data set for which $|E| > 3|V| - 6$. These graphs will be referred to as *dense graphs*. Note that even if a graph is not dense, one of its biconnected components might be, so the number of dense graphs is only a lower bound on the number of times early termination occurs.

The method used to generate non-planar graphs only generates a second connected component if a new vertex is created, but no edges are added between this vertex and the rest of the graph. The component may grow in size if another vertex is added and connected to this second component, but not to the rest of the graph. This is not likely, so most connected components will be isolated points. Recall that these points are ignored by the algorithm, so we would like to minimise the number of unconnected graphs.

The parameter $maxSteps$ of steps may not exceed 200, because this could result in graphs with more than 200 vertices. To ensure variety in the number of vertices and edges in the test set we choose $minSteps = 0$.

The operation that influences the density of a graph the most is adding an extra node. Depending on p_E, a number of edges is added, while the other two operations only introduce one new edge. If p_E is low, it will also introduce isolated points and increase the number of unconnected graphs in the data set. Figure 14 illustrates this influence of p_E on the number of dense and connected graphs in a test set of 10000 graphs. All other parameters are fixed. As p_E increases, the set contains more connected graphs, but the number of dense graphs increases as well. p_{min} and p_{max}, should be chosen such that there is a balance between these two criteria.

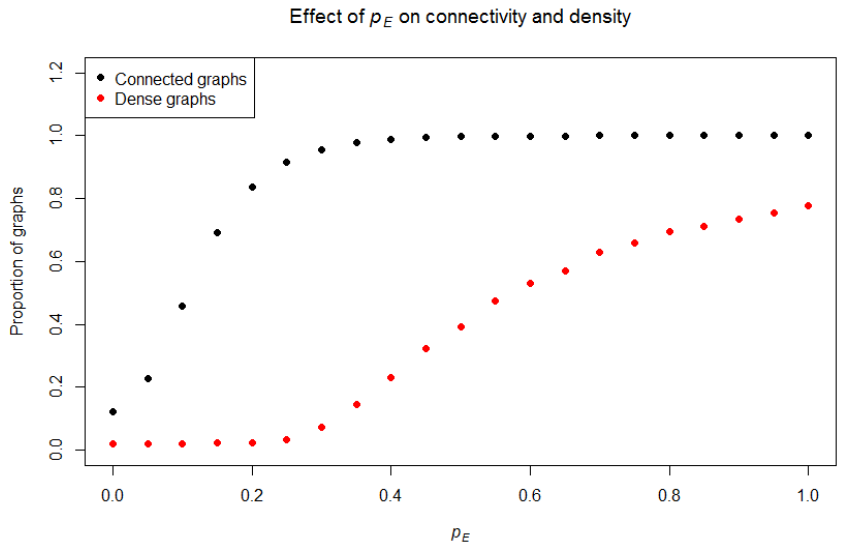


Figure 14: The influence of probability p_E on the number of connected and dense graphs in the non-planar data set.

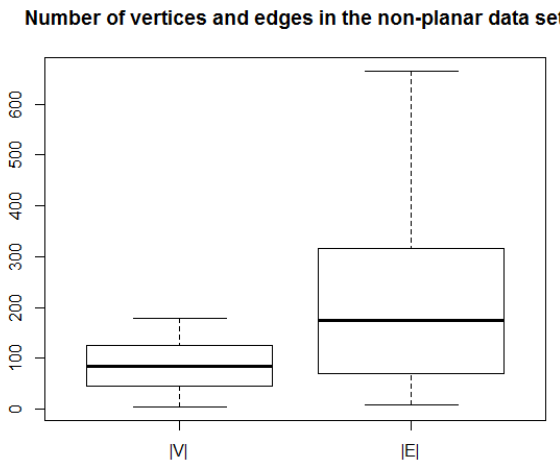
Using the parameters

$$\{minSteps, maxSteps, p_{min}, p_{max}, p_{addE}, p_{splitE}, p_{K5}\} = \{0, 200, 0.05, 0.15, 0.2, 0.8, 0.5\}$$

we obtain a test set that complies with the above criteria. Figure 15 summarises its composition.

Number of graphs	10000
Number of connected graphs	7968
Number of dense graphs	922
$\overline{\lvert V \rvert}$	86.244
$\overline{\lvert E \rvert}$	206.285
$\sigma(\lvert V \rvert)$	46.911
$\sigma(\lvert E \rvert)$	150.493

(a) Composition of the non-planar data set.



(b) A box plot of the number of vertices and edges of the graphs from the non-planar data set.

Figure 15: Composition of the non-planar data set generated with parameters $\{minSteps, maxSteps, p_{min}, p_{max}, p_{addE}, p_{splitE}, p_{K5}\} = \{0, 200, 0.05, 0.15, 0.2, 0.8, 0.5\}$.

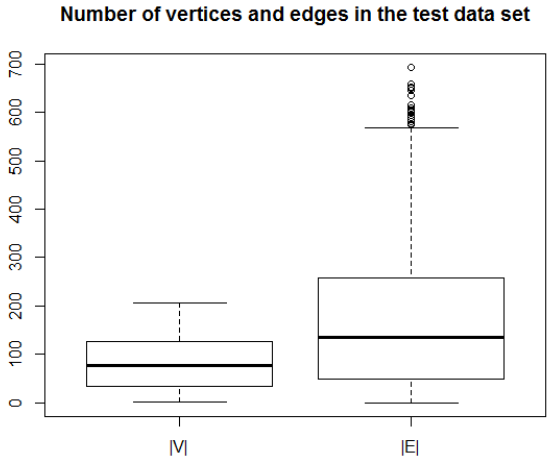
4.5.3 Results

To test the AP algorithm, the graph generation algorithms from Section 4.5.1 and 4.5.2 are combined into a single script. It generates a data set of 10000 graphs, each of which is planar with probability 0.5. The script writes the adjacency matrices of all graphs to a single text file and creates an answer file with a single boolean for each graph, indicating whether it is planar. Figure 16 describes the resulting data set that was used for testing the implementation of the Auslander-Parter algorithm.

The testing procedure is automated by a Java class that uses the output files from the generation script. For every graph in the test set, it reads the adjacency matrix, converts it to a graph object, runs the Auslander-Parter algorithm and verifies the correctness of the result with the answer key. If the result is incorrect, it outputs the adjacency matrix of the graph that caused the mistake and whether it was planar or not.

Number of graphs	10000
Number of connected graphs	7934
Number of dense graphs	452
Number of planar graphs	4949
$\overline{\lvert V \rvert}$	81.942
$\overline{\lvert E \rvert}$	165.032
$\sigma(\lvert V \rvert)$	54.396
$\sigma(\lvert E \rvert)$	133.976

(a) Composition of the test set for the Auslander-Parter algorithm.

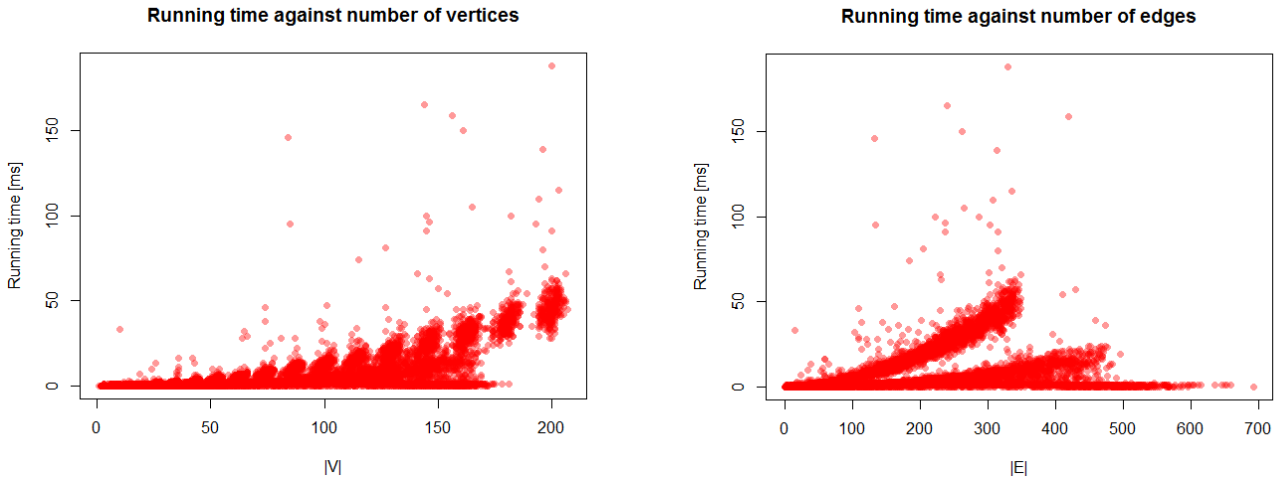


(b) A box plot of the number of vertices and edges of the graphs from the test set.

Figure 16: Composition of the test set generated with the parameters from Section 4.5.1 and 4.5.2.

For each of the graphs in the test set, the AP algorithm gives the correct answer. Given the variety of graphs in the data set, this strongly suggests that the implementation works correctly.

Besides the correctness of the Java implementation, we are also interested in its running time. In Section 4.3 we proved the asymptotic running time of the Auslander-Parter algorithm to be cubic in the number of vertices of the input graph. However, in practice the average running time of the algorithm might be better than this theoretical upper bound. Counting the number of elementary steps cannot be done with any standard function in Java, but we can easily count the number of milliseconds it takes to complete the algorithm with a timer. Figure 17 shows the running time in milliseconds plot against the number of vertices and edges respectively of the graphs in the test set. We see two distinct clusters of points corresponding to the planar and non-planar graphs in the test data. This is due to the fact that non-planarity always leads to early termination, either because the graph is dense and rejected immediately or because a conflict is found before the end of the algorithm. If the input graph is planar, the algorithm is always fully executed. While the running time is asymptotically cubic for both types of graphs, this difference is still visible for the graphs in our test set, because they are relatively small. Based on this difference, it makes sense to investigate the running time of the algorithm on planar and non-planar graphs separately. We will only focus on the worst-case scenario and examine the running time for a data set of planar graphs.



(a) *(b)*

Figure 17: The running time in milliseconds of the AP algorithm plotted against the number of vertices (a) and edges (b) of the test graphs.

Using the planar graph generation algorithm from Section 4.5.1, we generated a second data set consisting of planar graphs only. The running times of this data set are plot in Figure 18. Since we expect a cubic running time, a cubic regression line was added to both plots. These lines are described by the polynomials

$$P_V(|V|) = -4.291217 \cdot |V|^3 + 351.519486 \cdot |V|^2 + 1407.304141 \cdot |V| + 12.287000 \tag{2}$$

and

$$P_E(|E|) = -5.381189 \cdot |E|^3 + 345.568027 \cdot |E|^2 + 1416.999870 \cdot |E| + 12.287000 \tag{3}$$

for the number of vertices and edges respectively with an R-squared value of 0.8908 and 0.9007.

To compare these results to the theoretical running time, we need a conversion from milliseconds to number of elementary operations. By running an empty for loop of one million iterations a million times, we obtain an average running time of 0.301294 per million elementary operations. Converting equation 2 and 3 to the number of elementary operations per vertex or edge yields

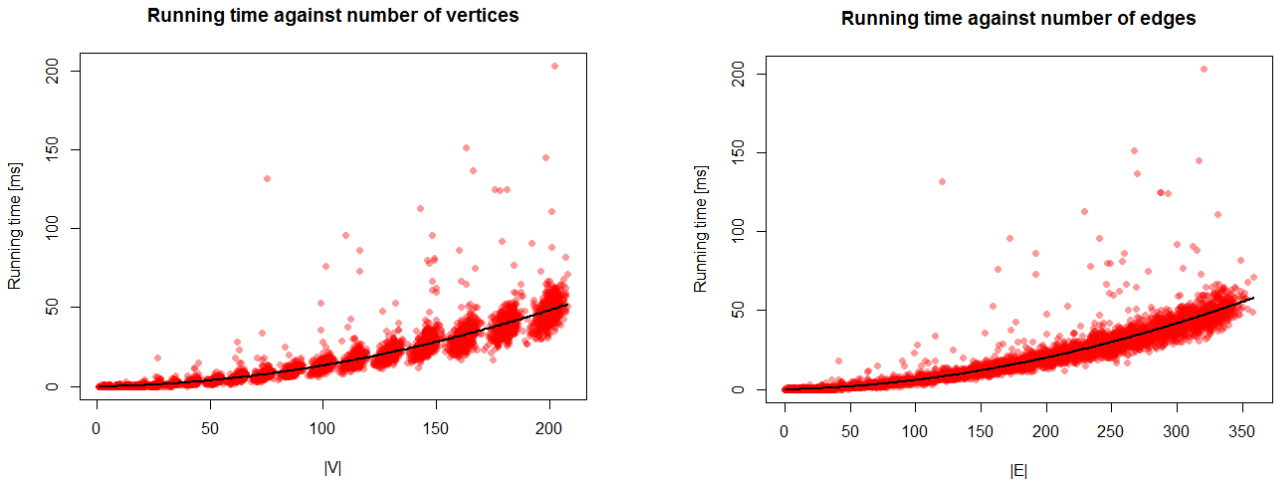
$$\hat{P}_V(|V|) = -14242625 \cdot |V|^3 + 1166699257 \cdot |V|^2 + 4670866796 \cdot |V| + 40780766. \tag{4}$$

and

$$\hat{P}_E(|E|) = -17860259 \cdot |E|^3 + 1146946262 \cdot |E|^2 + 4703047090 \cdot |E| + 40780766. \quad (5)$$

The coefficients are much larger than the maximum number of vertices of any input graph, so for our data set they will dominate the input size and mostly determine the running time. This means that in practice our algorithm's running time is much worse than cubic for small graphs. We also see that while asymptotically it holds that $O(|V|) = O(|E|)$, this equation does does not scale to small graphs, as there is a distinct difference between equation 4 and 5.

It should be noted that although we used the same power settings for each test and limited background activity as much as possible, it is impossible to guarantee a constant performance. Nevertheless, the coefficients in equation 4 are of such a magnitude, that it is safe to assume that the real coefficients in the running time function of the AP algorithm have a significant influence on the running time for small graphs.



(a) *(b)*

Figure 18: The running time in milliseconds of the AP algorithm plotted against the number of vertices (a) and edges (b) of 10000 planar test graphs.

Since the theoretical running time we proved in section 4.3 is asymptotic, we should test the algorithm on a data set with larger graphs to get a more accurate comparison. If the number of vertices of the input graphs is sufficiently large, it should start to dominate the coefficients of the running time polynomial. Due to the slow running time we are limited in our choice of $|V|$, but it should be possible to test a data set of 10000 graphs with approximately 10000 vertices overnight. However, if we try to run the algorithm on such a test set, we run into the practical limitations of Java. Graphs with more vertices generally require the algorithm to go deeper into recursion, which causes stack overflow errors. There is no straightforward workaround for this problem that does not directly affect the running time of the algorithm, so we are forced to restrict our analysis to small graphs only.

4.6 Visualisation

Besides an efficient graph data structure, GraphStream offers a method to visualise graphs. We can link the Auslander-Parter algorithm to this visual component to show the steps that the algorithm takes on a graph. The result can be used as a tool to explain the algorithm and for debugging purposes. To control the algorithm and input, a simple graphic user interface was built around GraphStream's visualisation window, as shown in Figure 19. This interface offers the following functionalities:

- **Different input modes** - Buttons 'Adjacency matrix' and 'Edge list' allow the user to enter a graph in two different ways. One can either enter an adjacency matrix or a list of all edges in the graph. In each case, the input should start with a line with a single integer, specifying the number of input lines

to follow. The input can be typed or copied in the text field underneath the buttons. When 'Confirm' is pressed, the input is evaluated. If it describes a valid graph, this graph is displayed in the blank area to the right of the control panel. If not, an error message is displayed in the bottom text area.

- **Graph generation** - The 'Random graph' button allows the user to generate a random graph of the type selected in the drop-down menu. One can choose between a planar graph, a non-planar graph and a graph of random type. The parameters for graph generation are chosen such that a random graph rarely has more than 30 vertices.

- **Run the AP algorithm** - The 'Start' button starts the Auslander-Parter algorithm on the graph currently displayed. 'Stop' halts the algorithm. If a new graph is entered during the execution of the algorithm, it stops immediately. The bottom text area displays information about the current recursive invocation of the algorithm, such as the level of recursion, the number of segments and whether a base case was encountered. When the algorithm is finished, it outputs whether the graph is planar.

- **Visualisation** - The check box labelled 'Visualisation' allows the user to choose whether the algorithm should give visual output. If it is checked, the steps of the algorithm are visualised on the graph in the display area on the right. Details about the visualisation are specified below. If the number of vertices of the graph exceeds 30, visualisation is disabled by default, because it would take several minutes for the algorithm to complete. If a graph with more than 200 vertices is entered, the algorithm is not executed at all and an error message is displayed.

- **Help function** - The blue help button causes an explanation of both input modes to be displayed in the bottom text area.

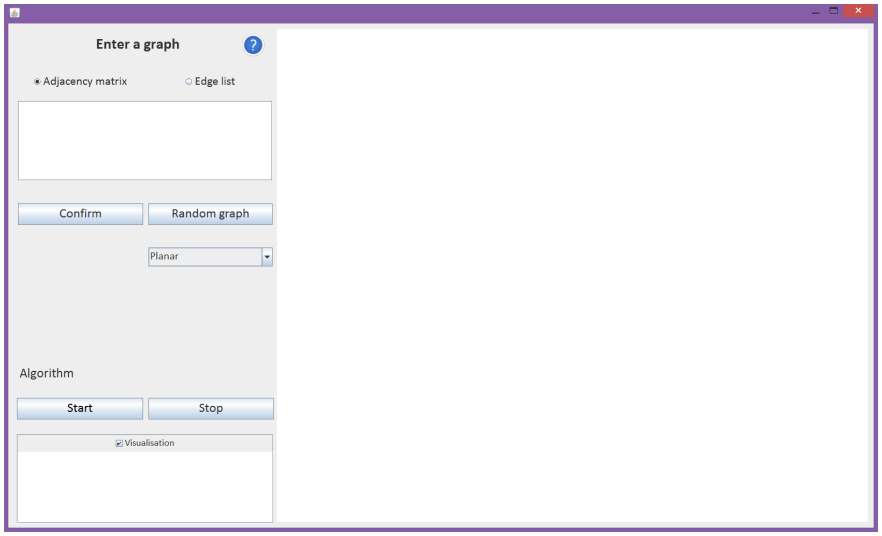
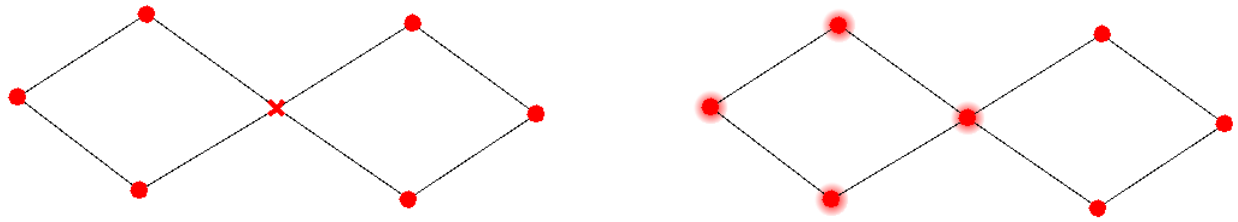


Figure 19: The graphical user interface to operate the Auslander-Parter algorithm.

If the visualisation option is enabled, several steps of the algorithm will be marked on the graph using different colours and line types. Below, the visual aspects of each phase of the algorithm are specified.

i. Biconnected components

At the start of the algorithm, the shape of all articulation points briefly changes to a cross to show which biconnected components have been found (Figure 20a). During the rest of the algorithm, the vertices of the biconnected component that is being processed have a red glow to indicate that they are currently being examined (Figure 20b).



(a) (b)

Figure 20: Visualisation of an articulation point (a) and biconnected component (b) in a graph.

ii.a) Cycle and segments
A cycle is indicated by black vertices and dashed lines. The algorithm colours the vertices and edges of the cycle in a cyclic order and pauses for a short period of time between colouring each vertex or edge. Vertices belonging to a segment are already distinguished by their red glow, so they do not need to be recoloured.

ii.b) Interlacement graph
No visual output is generated during this phase of the algorithm.

ii.c) Testing for bipartiteness
Each segment is alternately coloured orange or blue during the bipartiteness testing algorithm, as shown in Figure 21. There is currently no visual output to indicate that a conflict has been found.

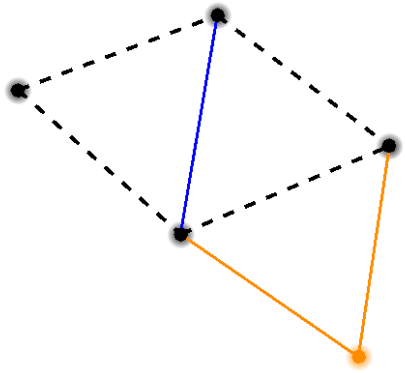


Figure 21: Visualisation of a cycle and two segments.

ii.d) Recursion
When a new recursive invocation of the algorithm starts, the layout of the graph is reset and the new biconnected component and cycle are coloured as specified above.

5 Conclusion

Given an arbitrary graph $G = (V, E)$, we have proved that the Auslander-Parter algorithm correctly establishes whether it is planar or not and that this test can be done in $O(|V|^3)$ time. Experimental results show that our implementation of the algorithm works as intended on a randomly generated data set. Given the diversity of the test data, the algorithm will likely produce correct results for any given input graph. Due to shortcomings of our method of implementation, we restricted our test data to graphs of at most two hundred vertices. For these graphs, the running time is worse than cubic in practice.

References

[Auslander and Parter, 1961] Auslander, L. and Parter, S. V. (1961). On imbedding graphs in the sphere. *Journal of Mathematics and Mechanics*, 10(3):517–523.

[Battista et al., 1999] Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G. (1999). *Graph drawing: algorithms for the visualization of graphs*. Prentice Hall PTR.

[Bondy and Murty, 2008] Bondy, J. A. and Murty, U. S. (2008). Graph theory. *Graduate texts in mathematics*, 1(244):120–121.

[Boyer and Myrvold, 2004] Boyer, J. M. and Myrvold, W. J. (2004). On the cutting edge: Simplified o (n) planarity by edge addition. *J. Graph Algorithms Appl.*, 8(2):241–273.

[Brinkmann et al., 2013] Brinkmann, G., Coolsaet, K., Goedgebeur, J., and Mélot, H. (2013). House of Graphs: a database of interesting graphs. *Discrete Applied Mathematics*, 161(1-2):311–314.

[Cormen, 2009] Cormen, T. H. (2009). *Introduction to algorithms*, pages 517–523. MIT press.

[De Fraysseix et al., 2006] De Fraysseix, H., De Mendez, P. O., and Rosenstiehl, P. (2006). Trémaux trees and planarity. *International Journal of Foundations of Computer Science*, 17(05):1017–1029.

[Goldstein, 1963] Goldstein, A. (1963). An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Graph and Combinatorics Conference, Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dept. of Mathematics, Princeton University, May 16-18*.

[Harary, 1969] Harary, F. (1969). *Graph theory*, chapter 2, page 18. Addison-Wesley, Reading, MA.

[Hopcroft and Tarjan, 1973] Hopcroft, J. E. and Tarjan, R. E. (1973). Efficient algorithms for graph manipulation (Algorithm 447). *Commun. ACM*, 16(6):372–378.

[Jordan, 1893] Jordan, C. (1893). *Cours d'analyse de l'École polytechnique*, volume 1. Gauthier-Villars et fils.

[Kuratowski, 1930] Kuratowski, C. (1930). Sur le probleme des courbes gauches en topologie. *Fundamenta mathematicae*, 15(1):271–283.

[Shirey, 1969] Shirey, R. W. (1969). Implementation and analysis of efficient graph planarity testing algorithms.

[Tamassia, 2013] Tamassia, R. (2013). *Handbook of graph drawing and visualization*. CRC press.

[Tantalo and Radcliffe, 2007] Tantalo, J. and Radcliffe, M. (2007). Planarity.

A Java implementation

Auslander-Parter

```
1  public class AuslanderParter {
2    private Graph theGraph;
3    private Set<Graph> biconnectedComps;
4    private boolean planar;
5    private boolean vis;
6    private boolean text;
7    private JTextArea textArea;
8    private SwingWorker<String, String> sw;
9
10   public AuslanderParter(Graph graph, boolean vis) {
11     theGraph = graph;
12     planar = true;
13     text = false;
14     this.vis = vis;
15   }
16
17   public AuslanderParter(Graph graph, boolean vis, JTextArea textArea,
        SwingWorker<String, String> sw) {
18     theGraph = graph;
19     planar = true;
20     text = true;
21     this.vis = vis;
22     this.textArea = textArea;
23     this.sw = sw;
24   }
25
26   /*
27    * updates text area in GUI
28    */
29   private void update(String s) {
30     textArea.append(s);
31     textArea.setCaretPosition(textArea.getDocument().getLength());
32   }
33
34   /*
35    * executes the Auslander-Parter algorithm
36    */
37   public boolean compute() {
38     if (vis) Color.decolorComponent(theGraph, theGraph, true, false, sw);
39     if (text && !sw.isCancelled()) update("Computing biconnected
          components...\n");
40     biconnectedComps = new BiconnectedComponents(theGraph, vis, sw).
          getBiconnectedComponents();
41     Iterator<Graph> it = biconnectedComps.iterator();
42     while (it.hasNext()) {
43       Graph comp = it.next();
44       if (text && !sw.isCancelled()) update("\nNew biconnected component\
           nRecursion level 0\n");
45       if (comp.getEdgeCount() > 3 * comp.getNodeCount() - 6 && comp.
```

```
          getNodeCount() > 2) {
46          planar = false;
47          if (text && !sw.isCancelled()) update(" This component has too
              many edges\n\n");
48        }
49        if (!planar) break;
50        if (vis) Color.colorComponent(theGraph, comp, true, false, sw);
51        ArrayList<Integer> cycle = Cycle.getCycle(comp);
52        if (text && !sw.isCancelled()) update(" Computing cycle\n");
53        if (cycle.size() > 0 && vis) Color.colorCycle(theGraph, comp, cycle,
            true, sw);
54        recurse(comp, cycle, 1);
55        if (!planar) break;
56      }
57      return planar;
58    }
59
60    /*
61     * recursively applies the Auslander-Parter algorithm to each
          biconnected component
62     */
63    private void recurse(Graph comp, ArrayList<Integer> cycle, int level) {
64      if (cycle.size() == 0) {
65        if (text && !sw.isCancelled()) update(" No cycle found\n");
66        if (vis) Color.decolorComponent(theGraph, comp, true, false, sw);
67        return;
68      }
69      CycleSegments su = new CycleSegments(theGraph, comp, cycle, vis, sw);
70      ArrayList<Segment> segs = su.getSegments();
71      if (su.countSegments() == 0) {
72        if (vis) Color.decolorComponent(theGraph, comp, true, false, sw);
73        if (text && !sw.isCancelled()) update(" Base case: cycle has no
            segments\n");
74        return;
75      }
76      if (su.hasOnePath()) {
77        if (vis) {
78          Color.colorSegment(theGraph, comp, segs.get(0), true, sw);
79          Color.decolorComponent(theGraph, comp, true, false, sw);
80        }
81        if (text && !sw.isCancelled()) update(" Base case: cycle has one
            segment, which is a path\n");
82        return;
83      }
84      if (text && !sw.isCancelled()) update(" " + segs.size() + " segments
          found\n");
85      if (!(new Interlacement(segs, theGraph, comp, cycle, vis, sw).
          isBipartite())) {
86        planar = false;
87        if (text && !sw.isCancelled()) update(" Conflicting segments found\n
            ");
88        return;
89      }
90      boolean first = true;
```

```
91      for (Segment s: segs) {
92        if (!planar) return;
93        if (text && !sw.isCancelled()) update("Recursion level " + level + "
              \n");
94        if (!first && vis)
95          Color.colorCycle(theGraph, comp, cycle, false, sw);
96        first = false;
97        Graph newComp = Cycle.mergeCycleSegment(s, cycle, comp);
98        if (vis) {
99          Color.decolorComponent(theGraph, comp, true, true, sw);
100         Color.colorComponent(theGraph, newComp, false, true, sw);
101       }
102       if (newComp.getEdgeCount() > 3 * newComp.getNodeCount() - 6 &&
              newComp.getNodeCount() > 2) {
103         planar = false;
104         if (text && !sw.isCancelled()) update("This component has too many
                edges\n");
105         return;
106       }
107       ArrayList<Integer> newCycle = Cycle.graphToIndex(newComp, Cycle.
              indexToGraph(comp, su.augmentCycle(s)));
108       recurse(newComp, newCycle, level + 1);
109     }
110     if (planar && vis) Color.decolorComponent(theGraph, comp, true, false,
            sw);
111   }
112 }
113
114 public class BiconnectedComponents {
115   private Graph theGraph;
116   private Set<Graph> biconnectedComps;
117   private int time;
118   private int[] tin;
119   private int[] lowlink;
120   private boolean[] visited;
121   private boolean vis;
122   private Stack<Integer> stack;
123   private List<Integer> articulationPoints;
124   private SwingWorker<String, String> sw;
125
126   public BiconnectedComponents(Graph graph, boolean vis, SwingWorker<
        String, String> sw) {
127     this.vis = vis;
128     this.sw = sw;
129     theGraph = graph;
130   }
131
132   /*
133    * finds all biconnected components with a DFS
134    */
135   private void dfsBiComp(int u, int p) {
136     visited[u] = true;
137     lowlink[u] = tin[u] = time++;
138     int children = 0;
```

```
139       boolean articulationPoint = false;
140       for (Edge e: theGraph.getNode(u).getEachEdge()) {
141         int v = e.getOpposite(theGraph.getNode(u)).getIndex();
142         if (!visited[v]) {
143           stack.add(e.getIndex());
144           dfsBiComp(v, u);
145           lowlink[u] = Math.min(lowlink[u], lowlink[v]);
146           articulationPoint |= lowlink[v] >= tin[u];
147           ++children;
148           if (lowlink[v] >= tin[u]) {
149             Graph component = new SingleGraph("comp", false, true);
150             while (!stack.isEmpty() && stack.peek() != e.getIndex()) {
151               Edge x = theGraph.getEdge(stack.pop());
152               component.addEdge(x.getId(), x.getNode0().getId(), x.getNode1
      ().getId());
153             }
154             Edge x = theGraph.getEdge(stack.pop());
155             component.addEdge(x.getId(), x.getNode0().getId(), x.getNode1().
        getId());
156             biconnectedComps.add(component);
157           }
158         }
159         else if (v != p && tin[v] < tin[u]) {
160           stack.add(e.getIndex());
161           lowlink[u] = Math.min(lowlink[u], tin[v]);
162         }
163       }
164       if (p == -1) {
165         articulationPoint = children >= 2;
166         if (children == 0) {
167           Graph component = new SingleGraph("comp", false, true);
168           component.addNode(theGraph.getNode(u).getId());
169           biconnectedComps.add(component);
170         }
171       }
172       if (articulationPoint)
173         articulationPoints.add(u);
174     }
175
176     /*
177      * returns all biconnected components
178      */
179     public Set<Graph> getBiconnectedComponents(){
180       int n = theGraph.getNodeCount();
181       visited = new boolean[n];
182       stack = new Stack<>();
183       time = 0;
184       tin = new int[n];
185       lowlink = new int[n];
186       articulationPoints = new ArrayList<>();
187       biconnectedComps = new HashSet<>();
188       for (int u = 0; u < n; u++)
189         if (!visited[u])
190           dfsBiComp(u, -1);
```

```
191       if (vis) {
192         Color.colorAP(theGraph, articulationPoints, sw);
193         Color.decolorComponent(theGraph, theGraph, true, false, sw);
194       }
195       return biconnectedComps;
196     }
197   }
198
199   public class Color {
200
201     /*
202      * pauses the program for x milliseconds
203      */
204     private static void sleep(int x, SwingWorker<String, String> sw) {
205       long startTime = System.currentTimeMillis();
206       long elapsedTime = 0L;
207       while (elapsedTime < x){
208         if (sw.isCancelled()) break;
209         elapsedTime = (new Date()).getTime() - startTime;
210       }
211     }
212
213     /*
214      * marks all articulation points
215      */
216     public static void colorAP(Graph graph, List<Integer> aps, SwingWorker<
          String, String> sw) {
217       for (int i: aps)
218         graph.getNode(i).setAttribute("ui.class", "cut");
219       sleep(1800, sw);
220     }
221
222     /*
223      * colors a biconnected component
224      */
225     public static void colorComponent(Graph graph, Graph comp, boolean slow,
          boolean cycle, SwingWorker<String, String> sw) {
226       if (slow) sleep(900, sw);
227       if (cycle) {
228         for (Node node: comp.getEachNode()) {
229           if (graph.getNode(node.getId()).getAttribute("ui.class") == null
              || !graph.getNode(node.getId()).getAttribute("ui.class").equals
              ("cycle"))
230             graph.getNode(node.getId()).setAttribute("ui.class", "bicomp");
231         }
232         for (Edge edge: comp.getEachEdge()) {
233           if (graph.getEdge(edge.getId()).getAttribute("ui.class") == null
              || !graph.getEdge(edge.getId()).getAttribute("ui.class").equals
              ("cycle"))
234             graph.getEdge(edge.getId()).setAttribute("ui.class", "bicomp");
235         }
236       }
237       else {
238         for (Node node: comp.getEachNode()) graph.getNode(node.getId()).
```

```
            setAttribute("ui.class", "bicomp");
239       for (Edge edge: comp.getEachEdge()) graph.getEdge(edge.getId()).
            setAttribute("ui.class", "bicomp");
240     }
241   }
242
243   /*
244    * decolors a biconnected component
245    */
246   public static void decolorComponent(Graph graph, Graph comp, boolean
        slow, boolean cycle, SwingWorker<String, String> sw) {
247     if (slow) sleep(800, sw);
248     if (cycle) {
249       for (Node node: comp.getEachNode()) {
250         if (graph.getNode(node.getId()).getAttribute("ui.class") == null
              || !graph.getNode(node.getId()).getAttribute("ui.class").equals
              ("cycle"))
251           graph.getNode(node.getId()).removeAttribute("ui.class");
252       }
253     }
254     else {
255       for (Node node: comp.getEachNode()) graph.getNode(node.getId()).
            removeAttribute("ui.class");
256       for (Edge edge: comp.getEachEdge()) graph.getEdge(edge.getId()).
            removeAttribute("ui.class");
257     }
258   }
259
260   /*
261    * colors a node and the edge to its parent
262    */
263   private static void color(Graph graph, Graph comp, int prev, int cur,
        boolean slow, SwingWorker<String, String> sw) {
264     Node a = graph.getNode(comp.getNode(cur).getId());
265     Node b = graph.getNode(comp.getNode(prev).getId());
266     Edge c = a.getEdgeBetween(b);
267     c.addAttribute("ui.class", "cycle");
268     if (slow) sleep(700, sw);
269     a.addAttribute("ui.class", "cycle");
270     if (slow) sleep(700, sw);
271   }
272
273   /*
274    * Colors the nodes and edges of a cycle
275    */
276   public static void colorCycle(Graph graph, Graph comp, ArrayList<Integer
        > cycle, boolean slow, SwingWorker<String, String> sw) {
277     if (slow) sleep(1000, sw);
278     else sleep(500, sw);
279     int prev = cycle.get(cycle.size() - 1);
280     int cur = -1;
281     for (int i = 0; i < cycle.size(); i++) {
282       cur = cycle.get(i);
283       color(graph, comp, prev, cur, slow, sw);
```

```
284         prev = cur;
285       }
286     }
287
288     /*
289      * decolors the part of the cycle between a and b, excluding end points
290      */
291     public static void decolorCyclePart(Graph graph, Graph comp, ArrayList<
            Integer> cycle, int a, int b, SwingWorker<String, String> sw) {
292       sleep(900, sw);
293       for (int i = 0; i < cycle.size(); i++) {
294         int node1 = cycle.get((a + i) % cycle.size());
295         int node2 = cycle.get((a + i + 1) % cycle.size());
296         if ((a + i) % cycle.size() == b) break;
297         Node cur = graph.getNode(comp.getNode(node1).getId());
298         cur.getEdgeBetween(comp.getNode(node2).getId()).setAttribute("ui.
              class", "bicomp");
299         if (i != 0)
300           cur.setAttribute("ui.class", "bicomp");
301       }
302     }
303
304     /*
305      * colors the part of the cycle between a and b, including end points
306      */
307     public static void colorCyclePart(Graph graph, Graph comp, ArrayList<
            Integer> cycle, int a, int b, SwingWorker<String, String> sw) {
308       sleep(900, sw);
309       for (int i = 0; i < cycle.size(); i++) {
310         int node1 = cycle.get((a + i) % cycle.size());
311         int node2 = cycle.get((a + i + 1) % cycle.size());
312         Node cur = graph.getNode(comp.getNode(node1).getId());
313         cur.addAttribute("ui.class", "cycle");
314         sleep(700, sw);
315         if ((a + i) % cycle.size() != b) {
316           cur.getEdgeBetween(comp.getNode(node2).getId()).addAttribute("ui.
                class", "cycle");
317           sleep(700, sw);
318         }
319         if ((a + i) % cycle.size() == b) break;
320       }
321     }
322
323     /*
324      * decolors the segments of a cycle without decoloring the attachments
325      */
326     public static void decolorSegments(Graph graph, Graph comp, ArrayList<
            Segment> segs, SwingWorker<String, String> sw) {
327       sleep(900, sw);
328       for (Segment seg: segs) {
329         for (Node n: seg.getEachNode())
330           if (!seg.isAttachment(comp.getNode(n.getId()).getIndex()))
331             graph.getNode(n.getId()).removeAttribute("ui.class");
332         for (Edge e: seg.getEachEdge())
```

```
333           graph.getEdge(e.getId()).removeAttribute("ui.class");
334       }
335     }
336
337     /*
338      * colors the segments of a cycle without coloring the attachments
339      */
340     public static void colorSegment(Graph graph, Graph comp, Segment seg,
          boolean col, SwingWorker<String, String> sw) {
341       for (Node n: seg.getEachNode())
342         if (!seg.isAttachment(comp.getNode(n.getId()).getIndex())) {
343           if (col) graph.getNode(n.getId()).addAttribute("ui.class", "
              bipartite1");
344           else graph.getNode(n.getId()).addAttribute("ui.class", "bipartite2
              ");
345         }
346       for (Edge e: seg.getEachEdge()) {
347         if (col) graph.getEdge(e.getId()).addAttribute("ui.class", "
             bipartite1");
348         else graph.getEdge(e.getId()).addAttribute("ui.class", "bipartite2")
              ;
349       }
350       sleep(800, sw);
351     }
352   }
353
354   public class Cycle {
355     private static Stack<Integer> stk;
356     private static ArrayList<Integer> theCycle;
357     private static boolean cycleFound;
358
359     /*
360      * converts a cycle of nodes represented by vertex ids to a cycle of
          vertex indices
361      */
362     public static ArrayList<Integer> graphToIndex(Graph graph, ArrayList<
          String> cycle) {
363       ArrayList<Integer> indexCycle = new ArrayList<>();
364       for (int i = 0; i < cycle.size(); i++)
365         indexCycle.add(graph.getNode(cycle.get(i)).getIndex());
366       return indexCycle;
367     }
368
369     /*
370      * converts a cycle of nodes represented by vertex indices to a cycle of
           vertex ids
371      */
372     public static ArrayList<String> indexToGraph(Graph graph, ArrayList<
          Integer> cycle) {
373       ArrayList<String> idCycle = new ArrayList<>();
374       for (int i = 0; i < cycle.size(); i++)
375         idCycle.add(graph.getNode(cycle.get(i)).getId());
376       return idCycle;
377     }
```

```
378
379   /*
380    * sorts vertex ids
381    */
382   private static boolean compareNode(String n1, String n2) {
383     int a = Integer.parseInt(n1);
384     int b = Integer.parseInt(n2);
385     return a < b;
386   }
387
388   /*
389    * Converts the cycle into a graph
390    */
391   private static Graph getCycleGraph(Graph comp, ArrayList<Integer> cycle)
          {
392     Graph cycleGraph = new SingleGraph("cycle");
393     int prev = cycle.get(cycle.size() - 1);
394     int cur = cycle.get(0);
395     cycleGraph.addNode(comp.getNode(prev).getId());
396     for (int i = 0; i < cycle.size(); i++) {
397       cur = cycle.get(i);
398       String n1 = comp.getNode(prev).getId();
399       String n2 = comp.getNode(cur).getId();
400       if (i != cycle.size() - 1)
401         cycleGraph.addNode(n2);
402       if (compareNode(n1, n2)) cycleGraph.addEdge(n1 + " " + n2, n1, n2);
403       else cycleGraph.addEdge(n2 + " " + n1, n1, n2);
404       prev = cur;
405     }
406     return cycleGraph;
407   }
408
409   /*
410    * merges a cycle and a segment
411    */
412   public static Graph mergeCycleSegment(Segment seg, ArrayList<Integer>
          cycle, Graph comp) {
413     Graph mergeGraph = getCycleGraph(comp, cycle);
414     for (Node v: seg.getEachNode())
415       if (mergeGraph.getNode(v.getId()) == null)
416         mergeGraph.addNode(v.getId());
417     for (Edge e: seg.getEachEdge())
418       mergeGraph.addEdge(e.getId(), e.getNode0().getId(), e.getNode1().
             getId());
419     return mergeGraph;
420   }
421
422   /*
423    * finds a cycle in graph
424    */
425   private static void findCycle(Graph graph, int v, boolean[] discovered,
          int parent) {
426     discovered[v] = true;
427     stk.add(v);
```

```
428       Node node = graph.getNode(v);
429       for (Edge e: node.getEachEdge()) {
430         int target = e.getOpposite(node).getIndex();
431         if (!discovered[target])
432           findCycle(graph, target, discovered, v);
433         else if (target != parent && cycleFound == false) {
434           while (stk.peek() != target) {
435             int w = stk.pop();
436             theCycle.add(w);
437           }
438           int w = stk.pop();
439           theCycle.add(w);
440           cycleFound = true;
441           return;
442         }
443       }
444       if (!cycleFound) stk.pop();
445       return;
446     }
447
448     /*
449      * returns a list of the node ids of a cycle in biComp
450      */
451     public static ArrayList<Integer> getCycle(Graph biComp) {
452       stk = new Stack<>();
453       theCycle = new ArrayList<>();
454       cycleFound = false;
455       boolean[] discovered = new boolean[biComp.getNodeCount()];
456       findCycle(biComp, 0, discovered, -1);
457       return theCycle;
458     }
459   }
460
461   public class CycleSegments {
462     private Graph theComp, theGraph;
463     private ArrayList<Segment> segments;
464     private ArrayList<Integer> theCycle;
465     private HashMap<Integer, Integer> inCycle;
466     private Segment curSegment;
467     private Stack<Integer> stk;
468     private boolean found;
469     private boolean vis;
470     private SwingWorker<String, String> sw;
471
472     CycleSegments(Graph graph, Graph biComp, ArrayList<Integer> cycle,
          boolean vis, SwingWorker<String, String> sw) {
473       theComp = biComp;
474       theCycle = cycle;
475       theGraph = graph;
476       inCycle = new HashMap<>();
477       for (int i = 0; i < theCycle.size(); i++)
478         inCycle.put(theCycle.get(i), i);
479       segments = new ArrayList<>();
480       this.vis = vis;
```

```
481       this.sw = sw;
482       findSegments();
483     }
484
485     /*
486      * finds path to node goal in segment seg
487      */
488     private void DFS(Segment seg, int v, boolean[] discovered, int goal, int
          start) {
489       if (inCycle.containsKey(theComp.getNode(seg.getNode(v).getId()).
           getIndex()) && v != goal && v != start) {
490         discovered[v] = true;
491         return;
492       }
493       if (!found) stk.add(theComp.getNode(seg.getNode(v).getId()).getIndex()
           );
494       else return;
495       discovered[v] = true;
496       if (v == goal) found = true;
497       Node node = seg.getNode(v);
498       for (Edge e: node.getEachEdge()) {
499         int target = e.getOpposite(node).getIndex();
500         if (!discovered[target])
501           DFS(seg, target, discovered, goal, start);
502       }
503       if (!found) stk.pop();
504       return;
505     }
506
507     /*
508      * returns two consecutive segments of the cycle
509      */
510     private int[] get2Attachments(Segment seg) {
511       int[] ats = new int[2];
512       int start = -1;
513       for (int i = 0; i < theCycle.size(); i++)
514         if (seg.isAttachment(theCycle.get(i))) {
515           ats[0] = theCycle.get(i);
516           start = inCycle.get(theCycle.get(i));
517           break;
518         }
519       for (int i = 1; i < theCycle.size(); i++) {
520         if (seg.isAttachment(theCycle.get((i + start) % theCycle.size()))) {
521           ats[1] = theCycle.get((i + start) % theCycle.size());
522           break;
523         }
524       }
525       return ats;
526     }
527
528     /*
529      * returns a separating cycle based from a non-separating cycle
530      */
531     public ArrayList<Integer> augmentCycle(Segment seg) {
```

```
532       found = false;
533       stk = new Stack<>();
534       int[] ats = get2Attachments(seg), cons = new int[2];
535       cons[0] = seg.getNode(theComp.getNode(ats[0]).getId()).getIndex();
536       cons[1] = seg.getNode(theComp.getNode(ats[1]).getId()).getIndex();
537       boolean[] discovered = new boolean[seg.getNodeCount()];
538       DFS(seg, cons[1], discovered, cons[0], cons[1]);
539       ArrayList<Integer> newCycle = new ArrayList<>();
540       int startInd = inCycle.get(ats[1]);
541       for (int i = 0; i < theCycle.size(); i++) {
542         int node = theCycle.get((startInd + i) % theCycle.size());
543         newCycle.add(node);
544         if (node == ats[0]) break;
545       }
546       if (vis) Color.decolorCyclePart(theGraph, theComp, theCycle, inCycle.
              get(ats[0]), inCycle.get(ats[1]), sw);
547       stk.pop();
548       while (stk.size() > 1) newCycle.add(stk.pop());
549       HashMap<Integer, Integer> inNewCycle = new HashMap<>();
550       for (int i = 0; i < newCycle.size(); i++)
551         inNewCycle.put(newCycle.get(i), i);
552       if (vis) Color.colorCyclePart(theGraph, theComp, newCycle, inNewCycle.
              get(ats[0]), inNewCycle.get(ats[1]), sw);
553       return newCycle;
554     }
555
556     /*
557      * returns whether the first segment is a path
558      * should only be called when the cycle has only one segment
559      */
560     private boolean checkPath() {
561       Segment segment = segments.iterator().next();
562       for (Node node: segment.getEachNode())
563         if (node.getDegree() > 2) return false;
564       return true;
565     }
566
567     /*
568      * returns whether the only segment is a path
569      */
570     public boolean hasOnePath() {
571       return countSegments() == 1 && checkPath();
572     }
573
574     /*
575      * finds the segments attached to theCycle
576      */
577     private void DFSSegments(Graph graph, int v, boolean[] discovered, int
          parent) {
578       discovered[v] = true;
579       Node node = graph.getNode(v);
580       if (inCycle.containsKey(node.getIndex())) {
581         if (parent != -1) {
582           if (!curSegment.isAttachment(v)) curSegment.addAttachment(v);
```

```
583           Node b = graph.getNode(parent);
584           if (curSegment.getNode(node.getId()) == null) curSegment.addNode(
                  node.getId());
585           curSegment.addEdge(node.getEdgeBetween(b).getId(), node.getId(),
586               graph.getNode(parent).getId());
587         }
588         return;
589       }
590       curSegment.addNode(node.getId());
591       if (parent != -1) {
592         Node b = graph.getNode(parent);
593         curSegment.addEdge(node.getEdgeBetween(b).getId(), node.getId(),
594             graph.getNode(parent).getId());
595       }
596       for (Edge e: node.getEachEdge()) {
597         int target = e.getOpposite(node).getIndex();
598         if (!discovered[target] || inCycle.containsKey(target))
599           DFSSegments(graph, target, discovered, v);
600         else
601           if (curSegment.getEdge(e.getId()) == null)
602             curSegment.addEdge(e.getId(), e.getNode0().getId(), e.getNode1()
                    .getId());
603       }
604       return;
605     }
606
607     /*
608      * finds all chords attached to cycle theCycle
609      */
610     private void findChords(Graph graph) {
611       for (Edge e: graph.getEachEdge()) {
612         if (inCycle.containsKey(e.getNode0().getIndex()) && inCycle.
                containsKey(e.getNode1().getIndex())) {
613           int a1 = Math.min(inCycle.get(e.getNode0().getIndex()), inCycle.
                  get(e.getNode1().getIndex()));
614           int a2 = Math.max(inCycle.get(e.getNode0().getIndex()), inCycle.
                  get(e.getNode1().getIndex()));
615           if (a2 - a1 != 1 && a1 + theCycle.size() - a2 != 1) {
616             curSegment = new Segment("segment");
617             curSegment.addNode(e.getNode0().getId());
618             curSegment.addNode(e.getNode1().getId());
619             curSegment.addEdge(e.getId(), e.getNode0().getId(), e.getNode1()
                    .getId());
620             curSegment.addAttachment(e.getNode0().getIndex());
621             curSegment.addAttachment(e.getNode1().getIndex());
622             segments.add(curSegment);
623           }
624         }
625       }
626     }
627
628     /*
629      * finds the segments attached to theCycle
630      */
```

```
631   private void findSegments() {
632     boolean[] discovered = new boolean[theComp.getNodeCount()];
633     for (int i = 0; i < theComp.getNodeCount(); i++) {
634       if (discovered[i]) continue;
635       curSegment = new Segment("segment");
636       DFSSegments(theComp, i, discovered, -1);
637       if (curSegment.getNodeCount() != 0)
638         segments.add(curSegment);
639     }
640     findChords(theComp);
641   }
642
643   /*
644    * returns the number of segments attached to theCycle
645    */
646   public int countSegments() {
647     return segments.size();
648   }
649
650   /*
651    * returns a set of all segments attached to theCycle
652    */
653   public ArrayList<Segment> getSegments() {
654     return segments;
655   }
656 }
657
658 public class GraphReader {
659   private SingleGraph graph;
660
661   private String error = "";
662
663   public GraphReader() {
664      graph = new SingleGraph("Graph", false, true);
665      System.setProperty("org.graphstream.ui.renderer", "org.graphstream.ui
         .j2dviewer.J2DGraphRenderer");
666      System.setProperty("gs.ui.renderer", "org.graphstream.ui.j2dviewer.
         J2DGraphRenderer");
667   }
668
669   /*
670    * sorts the nodes of an input edge
671    */
672   private boolean compareNode(String[] newEdge) {
673     int a = Integer.parseInt(newEdge[0]);
674     int b = Integer.parseInt(newEdge[1]);
675     return a < b;
676   }
677
678   /*
679    * extracts a graph from an input string (edge mode)
680    */
681   private void readEdgeIO(String str) {
682     BufferedReader br = new BufferedReader(new StringReader(str));
```

```
683       try {
684         int nE = Integer.parseInt(br.readLine());
685         for (int i = 0; i < nE; i++) {
686           String[] newEdge = br.readLine().split("\\s+");
687           if (compareNode(newEdge))
688             graph.addEdge(newEdge[0] + " " + newEdge[1], newEdge[0] + "",
                  newEdge[1] + "");
689           else graph.addEdge(newEdge[1] + " " + newEdge[0], newEdge[0] + "",
                  newEdge[1] + "");
690         }
691       } catch (Exception e) {
692         error = "There seems to be a problem with the input format. Please
                try again.";
693       }
694     }
695
696     /*
697      * extracts a graph from an input string (matrix mode)
698      */
699     private void readMatrixIO(String str) {
700       BufferedReader br = new BufferedReader(new StringReader(str));
701       try {
702         int nE = Integer.parseInt(br.readLine());
703         for (int i = 0; i < nE; i++) {
704           Node n = graph.addNode("" + i);
705           n.addAttribute("ui.label", i);
706         }
707         for (int i = 0; i < nE; i++) {
708           char[] newEdge = br.readLine().toCharArray();
709           for (int j = i + 1; j < nE; j++)
710           if (newEdge[j] == '1') graph.addEdge(i + " " + j, "" + i, "" + j);
711         }
712       } catch (Exception e) {
713         error = "There seems to be a problem with the input format. Please
                try again.";
714       }
715     }
716
717     /*
718      * reads a graph from an input string
719      */
720     public SingleGraph read(int mode, String str) {
721       switch (mode) {
722       case 0: readEdgeIO(str);
723         break;
724       case 1: readMatrixIO(str);
725         break;
726       }
727       return graph;
728     }
729
730     /*
731      * returns the error (if any) encountered during graph reading
732      */
```

```
733   public String getError() {
734     return error;
735   }
736
737   /*
738    * returns an empty graph object
739    */
740   public Graph getEmptyGraph() {
741     return graph;
742   }
743 }
744
745 public class Interlacement {
746   private Graph theGraph, intGraph, theComp;
747   private HashMap<Integer, Integer> cycleLabel;
748   private ArrayList<Integer> theCycle;
749   private ArrayList<Segment> theSegments;
750   private int s, c;
751   private boolean vis;
752   private SwingWorker<String, String> sw;
753
754   Interlacement(ArrayList<Segment> segments, Graph graph, Graph comp,
        ArrayList<Integer> cycle, boolean vis, SwingWorker<String, String> sw
        ) {
755     theGraph = graph;
756     theComp = comp;
757     theCycle = cycle;
758     theSegments = segments;
759     cycleLabel = new HashMap<>();
760     intGraph = new SingleGraph("Interlacement graph");
761     s = theSegments.size();
762     c = theCycle.size();
763     this.vis = vis;
764     this.sw = sw;
765   }
766
767   /*
768    * BFS to test for bipartiteness
769    */
770   private boolean testBipartite(int src, int[] colors) {
771     colors[src] = (int) Math.round(Math.random());
772     if (vis) Color.colorSegment(theGraph, theComp, theSegments.get(src),
          colors[src] == 1, sw);
773     LinkedList<Integer> queue = new LinkedList<Integer>();
774     queue.add(src);
775     while (!queue.isEmpty()) {
776       int v = queue.pop();
777       if (vis) {
778         if (colors[v] == 1) Color.colorSegment(theGraph, theComp,
      theSegments.get(v), true, sw);
779         else Color.colorSegment(theGraph, theComp, theSegments.get(v),
      false, sw);
780       }
781       for (Edge e: intGraph.getNode(v).getEachEdge()) {
```

```
782           int target = e.getOpposite(intGraph.getNode(v)).getIndex();
783           if (colors[target] == -1) {
784             colors[target] = 1 - colors[v];
785             queue.push(target);
786           }
787           else if (colors[target] == colors[v])
788             return false;
789         }
790       }
791       return true;
792     }
793
794     /*
795      * initialises test for bipartiteness
796      */
797     private boolean testBipartiteMain() {
798       int colors[] = new int[s];
799       for (int i = 0; i < s; ++i)
800         colors[i] = -1;
801       for (int i = 0; i < s; i++)
802        if (colors[i] == -1)
803         if (testBipartite(i, colors) == false)
804           return false;
805        return true;
806     }
807
808     /*
809      * checks for a conflict between segment s2 and the segment
          corresponding to the current cycle labels
810      */
811     private boolean checkConflict(Segment s2, int max) {
812       int[] labels = new int[max + 1];
813       int sum = 0;
814       ArrayList<Integer> attachments = s2.getAttachments();
815       for (int a: attachments)
816         labels[cycleLabel.get(a)] = 1;
817       for (int i = 0; i < labels.length; i++)
818         sum += labels[i];
819       int partSum = 0;
820       for (int i = 0; i < 3; i++) partSum += labels[i];
821       for (int i = 0; i < max + 1; i++) {
822         if (partSum == sum && i % 2 == 0)
823           return false;
824         partSum += labels[(3 + i) % (max + 1)];
825         partSum -= labels[i];
826       }
827       return true;
828     }
829
830     /*
831      * computes the interlacement graph of the cycle segments
832      */
833     private void makeInterlacementGraph() {
834       for (int i = 0; i < s; i++)
```

```
835         intGraph.addNode(i + "");
836       for (int i = 0; i < s; i++) {
837         cycleLabel.clear();
838         Segment s1 = theSegments.get(i);
839         int label = 0, start = -1;
840         for (int j = 0; j < c; j++) {
841           if (s1.isAttachment(theCycle.get(j))) {
842             start = j;
843             break;
844           }
845         }
846         for (int j = 0; j < c; j++) {
847           if (s1.isAttachment(theCycle.get((j + start) % c))) {
848             if (j != 0) label++;
849             cycleLabel.put(theCycle.get((j + start) % c), label * 2);
850           }
851           else
852             cycleLabel.put(theCycle.get((j + start) % c), label * 2 + 1);
853         }
854         for (int j = i + 1; j < s; j++)
855           if (checkConflict(theSegments.get(j), 2 * label + 1))
856             intGraph.addEdge(i + " " + j, i, j);
857       }
858     }
859
860     /*
861      * returns whether the interlacement graph of this cycle's segments is
           bipartite
862      */
863     public boolean isBipartite() {
864       makeInterlacementGraph();
865       boolean planar = testBipartiteMain();
866       if (vis) {
867         Color.decolorSegments(theGraph, theComp, theSegments, sw);
868       }
869       return planar;
870     }
871   }
872
873   public class Segment extends SingleGraph {
874     private ArrayList<Integer> attachments;
875     private HashSet<Integer> isAttachment;
876
877     Segment(String id) {
878       super(id);
879       attachments = new ArrayList<>();
880       isAttachment = new HashSet<>();
881     }
882
883     /*
884      * adds an attachment at node c to the segment
885      */
886     public void addAttachment(int c) {
887       attachments.add(c);
```

```
888       isAttachment.add(c);
889     }
890
891     /*
892      * returns whether node a is an attachment of this segment
893      */
894     public boolean isAttachment(int a) {
895       return isAttachment.contains(a);
896     }
897
898     /*
899      * returns the number of attachments
900      */
901     public int getAttachmentCount() {
902       return attachments.size();
903     }
904
905     /*
906      * returns all attachments
907      */
908     public ArrayList<Integer> getAttachments() {
909       return attachments;
910     }
911 }
```

Interface

```
1  public class APGUI extends JFrame {
2    private Graph theGraph;
3    private String style;
4    private SwingWorker<String, String> sw;
5
6    private JTextArea textArea;
7    private JPanel contentPane;
8
9    private int inputMode = 1;
10   private int graphType = 0;
11   private String[] types = {"Planar", "Non-planar", "Random"};
12   private boolean vis = true;
13   private boolean checked = true;
14
15   private void setStyle() {
16     style = "node {"
17     + "fill-color: red, #FF0000;"
18     + "text-mode: normal;"
19     + "size: 15px;"
20     + "text-background-mode: plain;"
21     + "text-style: bold;"
22     + "text-color: white;"
23     + "text-alignment: center;"
24     + "text-background-mode: none;"
25     + "fill-mode: dyn-plain;"
26     + "}"
27     + "node.cycle {"
```

```
28      +   "fill-color: black, #000000;"
29      +  "shadow-mode: gradient-radial;"
30      +  "shadow-color: black, white;"
31      +  "shadow-width: 7px;"
32      +  "shadow-offset: 0, 0;"
33      + "}"
34      + "node.bicomp {"
35      +  "shadow-mode: gradient-radial;"
36      +  "shadow-color: red, white;"
37      +  "shadow-width: 7px;"
38      +  "shadow-offset: 0, 0;"
39      + "}"
40      + "node.bipartite1 {"
41      +   "fill-color: blue, #0000FF;"
42      +  "shadow-mode: gradient-radial;"
43      +  "shadow-color: blue, white;"
44      +  "shadow-width: 7px;"
45      +  "shadow-offset: 0, 0;"
46      + "}"
47      + "node.bipartite2 {"
48      + "fill-color: #FF8C00;"
49      +  "shadow-mode: gradient-radial;"
50      +  "shadow-color: #FF8C00, white;"
51      +  "shadow-width: 7px;"
52      +  "shadow-offset: 0, 0;"
53      + "}"
54      + "node.cut {"
55      +  "shape: cross;"
56      + "}"
57      + "edge {"
58      + "}"
59      + "edge.cycle {"
60      + "fill-mode: none;"
61      + "stroke-mode: dashes;"
62      + "stroke-width: 3px;"
63      + "}"
64      + "edge.bipartite1 {"
65      +  "fill-color: blue, #0000FF;"
66      +   "size: 3px;"
67      + "}"
68      + "edge.bipartite2 {"
69      +   "fill-color: #FF8C00;"
70      +   "size: 3px;"
71      + "}";
72    }
73
74    /*
75     * cancels Auslander-Parter algorithm if currently running
76     */
77    private void cancelSW() {
78      if (sw != null && !sw.isDone()) {
79        sw.cancel(true);
80        textArea.append("\n\nStopping...");
81        textArea.setCaretPosition(textArea.getDocument().getLength());
```

```
82        long startTime = System.currentTimeMillis();
83        long elapsedTime = 0L;
84        while (elapsedTime < 100)
85          elapsedTime = (new Date()).getTime() - startTime;
86        textArea.append("\nDone\n");
87        textArea.setCaretPosition(textArea.getDocument().getLength());
88      }
89    }
90
91    /*
92     * copies graph from graph reader to the interface
93     */
94    private void setGraph(Graph graph) {
95      theGraph.clear();
96      theGraph.addAttribute("ui.stylesheet", style);
97      for (Node n: graph.getEachNode())
98        theGraph.addNode(n.getId());
99      for (Edge e: graph.getEachEdge())
100       theGraph.addEdge(e.getId(), e.getNode0().getId(), e.getNode1().getId
            ());
101   }
102
103   /*
104    * creates the frame
105    */
106   public APGUI() {
107     setMinimumSize(new Dimension(640, 480));
108     System.setProperty("org.graphstream.ui.renderer", "org.graphstream.ui.
          j2dviewer.J2DGraphRenderer");
109     theGraph = new SingleGraph("graph");
110     setStyle();
111     theGraph.addAttribute("ui.stylesheet", style);
112     Viewer viewer = new Viewer(theGraph, Viewer.ThreadingModel.
          GRAPH_IN_ANOTHER_THREAD);
113     viewer.enableAutoLayout();
114     ViewPanel viewPanel = viewer.addDefaultView(false);
115     setExtendedState(Frame.MAXIMIZED_BOTH);
116     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
117     setBounds(100, 100, 450, 300);
118
119     contentPane = new JPanel();
120     contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
121     setContentPane(contentPane);
122     GridBagLayout gbl_contentPane = new GridBagLayout();
123     gbl_contentPane.columnWidths = new int[]{0, 0, 0};
124     gbl_contentPane.rowHeights = new int[]{0, 0, 0};
125     gbl_contentPane.columnWeights = new double[]{0.2, 1.0, 1.0};
126     gbl_contentPane.rowWeights = new double[]{0.6, 0.4, Double.MIN_VALUE};
127     contentPane.setLayout(gbl_contentPane);
128
129     JPanel panel = new JPanel();
130     GridBagConstraints gbc_panel = new GridBagConstraints();
131     gbc_panel.insets = new Insets(0, 5, 5, 5);
132     gbc_panel.fill = GridBagConstraints.BOTH;
```

```
133     gbc_panel.gridx = 0;
134     gbc_panel.gridy = 0;
135     contentPane.add(panel, gbc_panel);
136     GridBagLayout gbl_panel = new GridBagLayout();
137     gbl_panel.columnWidths = new int[]{0, 0, 0, 0, 0, 0, 0, 0};
138     gbl_panel.rowHeights = new int[]{0, 0, 0, 0, 0, 0};
139     gbl_panel.columnWeights = new double[]{1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
          1.0, 1.0};
140     gbl_panel.rowWeights = new double[]{0.2, 0.2, 1.0, 0.5, 0.5, 0.5};
141     panel.setLayout(gbl_panel);
142
143     JLabel lblTitle = new JLabel("Enter a graph");
144     lblTitle.setFont(new Font("Calibri", Font.BOLD, 28));
145     GridBagConstraints gbc_lblTitle = new GridBagConstraints();
146     gbc_lblTitle.gridwidth = 4;
147     gbc_lblTitle.insets = new Insets(0, 0, 5, 5);
148     gbc_lblTitle.gridx = 2;
149     gbc_lblTitle.gridy = 0;
150     panel.add(lblTitle, gbc_lblTitle);
151
152
153     JButton buttonHelp = new JButton("");
154     buttonHelp.setToolTipText("Help");
155     buttonHelp.setBorder(BorderFactory.createEmptyBorder());
156     buttonHelp.setContentAreaFilled(false);
157     buttonHelp.setIcon(new ImageIcon(APGUI.class.getResource("/com/sun/
          javafx/scene/control/skin/caspian/dialog-confirm.png")));
158     buttonHelp.addActionListener(new ActionListener() {
159       public void actionPerformed(ActionEvent arg0) {
160         if (sw == null || sw.isDone()) textArea.setText("Enter a graph in
              one of the following forms:\n* Adjacency matrix:\n"
161             + " 4 <- number of nodes\n 0111 <- adjacency matrix\n 1011\n
                  1101\n 1110"
162             + "\n\n* List of edges:\n 5 <- number of edges\n 0 1 <- edge
                  between node 0 and 1\n"
163             + " 1 2\n 2 3\n 1 3\n 0 2\n\nPress 'confirm' to see your graph
                   and 'random' to generate a random graph."
164             + "\nPress 'start' to start the Auslander-Parter algorithm and
                   'stop' to stop it.\n........\n\n");
165       }
166     });
167     buttonHelp.setPreferredSize(new Dimension(42, 42));
168     GridBagConstraints gbc_buttonHelp = new GridBagConstraints();
169     gbc_buttonHelp.insets = new Insets(0, 0, 5, 0);
170     gbc_buttonHelp.gridx = 7;
171     gbc_buttonHelp.gridy = 0;
172     panel.add(buttonHelp, gbc_buttonHelp);
173
174     JRadioButton rdbtnMatrix = new JRadioButton("Adjacency matrix");
175     rdbtnMatrix.setFont(new Font("Calibri", Font.PLAIN, 22));
176     rdbtnMatrix.addActionListener(new ActionListener() {
177       public void actionPerformed(ActionEvent e) {
178         inputMode = 1;
179       }
```

```
180     });
181     rdbtnMatrix.setSelected(true);
182     GridBagConstraints gbc_rdbtnMatrix = new GridBagConstraints();
183     gbc_rdbtnMatrix.gridwidth = 4;
184     gbc_rdbtnMatrix.insets = new Insets(0, 5, 5, 5);
185     gbc_rdbtnMatrix.gridx = 0;
186     gbc_rdbtnMatrix.gridy = 1;
187     panel.add(rdbtnMatrix, gbc_rdbtnMatrix);
188
189     JRadioButton rdbtnEdges = new JRadioButton("Edge list");
190     rdbtnEdges.setFont(new Font("Calibri", Font.PLAIN, 22));
191     rdbtnEdges.addActionListener(new ActionListener() {
192       public void actionPerformed(ActionEvent e) {
193         inputMode = 0;
194       }
195     });
196     GridBagConstraints gbc_rdbtnEdges = new GridBagConstraints();
197     gbc_rdbtnEdges.gridwidth = 4;
198     gbc_rdbtnEdges.insets = new Insets(0, 5, 5, 0);
199     gbc_rdbtnEdges.gridx = 4;
200     gbc_rdbtnEdges.gridy = 1;
201     panel.add(rdbtnEdges, gbc_rdbtnEdges);
202
203     ButtonGroup group = new ButtonGroup();
204     group.add(rdbtnMatrix);
205     group.add(rdbtnEdges);
206
207     JScrollPane scrollPane_1 = new JScrollPane();
208     GridBagConstraints gbc_scrollPane_1 = new GridBagConstraints();
209     gbc_scrollPane_1.gridwidth = 8;
210     gbc_scrollPane_1.insets = new Insets(0, 5, 5, 0);
211     gbc_scrollPane_1.fill = GridBagConstraints.BOTH;
212     gbc_scrollPane_1.gridx = 0;
213     gbc_scrollPane_1.gridy = 2;
214     panel.add(scrollPane_1, gbc_scrollPane_1);
215
216     JTextArea txtrInput = new JTextArea();
217     scrollPane_1.setViewportView(txtrInput);
218
219     JButton btnConfirm = new JButton("Confirm");
220     btnConfirm.setFont(new Font("Calibri", Font.PLAIN, 24));
221     GridBagConstraints gbc_btnConfirm = new GridBagConstraints();
222     btnConfirm.addActionListener(new ActionListener() {
223       public void actionPerformed(ActionEvent arg0) {
224         cancelSW();
225         GraphReader reader = new GraphReader();
226         Graph graph = reader.read(inputMode, txtrInput.getText());
227         textArea.setText(reader.getError());
228         if (textArea.getText().length() == 0 && graph.getNodeCount() > 30
               && graph.getNodeCount() <= 200) {
229           textArea.setText("This graph is too large to visualise the
                 Auslander-Parter algorithm. Press start to test for planarity
                  without visualisation.");
230           setGraph(graph);
```

```
231             vis = false;
232           }
233           else if (textArea.getText().length() == 0 && graph.getNodeCount()
                 > 200)
234             textArea.setText("This graph is too large to test for planarity
                   with the Auslander-Parter algorithm. Please enter a smaller
                   graph.");
235           else {
236             setGraph(graph);
237             vis = true;
238           }
239         }
240       });
241       gbc_btnConfirm.fill = GridBagConstraints.HORIZONTAL;
242       gbc_btnConfirm.gridwidth = 4;
243       gbc_btnConfirm.insets = new Insets(5, 5, 5, 5);
244       gbc_btnConfirm.gridx = 0;
245       gbc_btnConfirm.gridy = 3;
246       panel.add(btnConfirm, gbc_btnConfirm);
247
248       JButton btnRandom = new JButton("Random graph");
249       btnRandom.setFont(new Font("Calibri", Font.PLAIN, 24));
250       btnRandom.addActionListener(new ActionListener() {
251         public void actionPerformed(ActionEvent arg0) {
252           cancelSW();
253           textArea.setText("");
254           Graph graph = new GraphGenaratorMain(graphType).getGraph();
255           setGraph(graph);
256         }
257       });
258       GridBagConstraints gbc_btnRandom = new GridBagConstraints();
259       gbc_btnRandom.fill = GridBagConstraints.HORIZONTAL;
260       gbc_btnRandom.gridwidth = 4;
261       gbc_btnRandom.insets = new Insets(5, 5, 5, 0);
262       gbc_btnRandom.gridx = 4;
263       gbc_btnRandom.gridy = 3;
264       panel.add(btnRandom, gbc_btnRandom);
265
266       JComboBox comboBox = new JComboBox();
267       comboBox.setModel(new DefaultComboBoxModel(types));
268       comboBox.setSelectedIndex(0);
269       comboBox.addActionListener(new ActionListener() {
270         public void actionPerformed(ActionEvent e) {
271           JComboBox<String> combo = (JComboBox<String>) e.getSource();
272           String selected = (String) combo.getSelectedItem();
273           if (selected.equals("Planar"))
274             graphType = 0;
275           else if (selected.equals("Non-planar"))
276             graphType = 1;
277           else graphType = 2;
278         }
279       });
280       comboBox.setFont(new Font("Calibri", Font.PLAIN, 20));
281       GridBagConstraints gbc_comboBox = new GridBagConstraints();
```

```
282     gbc_comboBox.anchor = GridBagConstraints.NORTH;
283     gbc_comboBox.gridwidth = 4;
284     gbc_comboBox.insets = new Insets(5, 5, 5, 0);
285     gbc_comboBox.fill = GridBagConstraints.HORIZONTAL;
286     gbc_comboBox.gridx = 4;
287     gbc_comboBox.gridy = 4;
288     panel.add(comboBox, gbc_comboBox);
289
290     Component glue = Box.createGlue();
291     GridBagConstraints gbc_glue = new GridBagConstraints();
292     gbc_glue.gridwidth = 8;
293     gbc_glue.gridx = 0;
294     gbc_glue.gridy = 5;
295     panel.add(glue, gbc_glue);
296
297     JPanel panel_1 = new JPanel();
298     panel_1.setLayout(new BorderLayout(0, 0));
299     GridBagConstraints gbc_panel_1 = new GridBagConstraints();
300     gbc_panel_1.insets = new Insets(5, 0, 5, 5);
301     gbc_panel_1.gridheight = 2;
302     gbc_panel_1.fill = GridBagConstraints.BOTH;
303     gbc_panel_1.gridwidth = 2;
304     gbc_panel_1.gridx = 1;
305     gbc_panel_1.gridy = 0;
306     panel_1.add(viewPanel);
307     contentPane.add(panel_1, gbc_panel_1);
308
309     JPanel panel_2 = new JPanel();
310     GridBagConstraints gbc_panel_2 = new GridBagConstraints();
311     gbc_panel_2.insets = new Insets(0, 5, 0, 5);
312     gbc_panel_2.fill = GridBagConstraints.BOTH;
313     gbc_panel_2.gridx = 0;
314     gbc_panel_2.gridy = 1;
315     contentPane.add(panel_2, gbc_panel_2);
316     GridBagLayout gbl_panel_2 = new GridBagLayout();
317     gbl_panel_2.columnWidths = new int[]{0, 0};
318     gbl_panel_2.rowHeights = new int[]{0, 0, 0, 0, 0, 0};
319     gbl_panel_2.columnWeights = new double[]{1.0, 1.0};
320     gbl_panel_2.rowWeights = new double[]{1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
321     panel_2.setLayout(gbl_panel_2);
322
323     JLabel lblAlgorithm = new JLabel("Algorithm");
324     lblAlgorithm.setFont(new Font("Calibri", Font.PLAIN, 25));
325     GridBagConstraints gbc_lblAlgorithm = new GridBagConstraints();
326     gbc_lblAlgorithm.anchor = GridBagConstraints.SOUTHWEST;
327     gbc_lblAlgorithm.insets = new Insets(0, 10, 7, 5);
328     gbc_lblAlgorithm.gridwidth = 2;
329     gbc_lblAlgorithm.gridx = 0;
330     gbc_lblAlgorithm.gridy = 0;
331     panel_2.add(lblAlgorithm, gbc_lblAlgorithm);
332
333     JButton btnStart = new JButton("Start");
334     btnStart.setForeground(new Color(0, 0, 0));
335     btnStart.addActionListener(new ActionListener() {
```

```
336       public void actionPerformed(ActionEvent arg0) {
337         if (theGraph.getNodeCount() != 0) {
338           cancelSW();
339           textArea.setText("");
340           sw = new SwingWorker<String, String>() {
341
342             @Override
343             protected String doInBackground() throws Exception {
344               AuslanderParter algo = new AuslanderParter(theGraph, vis,
                      textArea, sw);
345               boolean planar = algo.compute();
346               return planar? "planar":"not planar";
347             }
348
349             @Override
350             protected void done() {
351               try {
352                 String out = "\nThis graph is " + get();
353                 textArea.append(out);
354               } catch (Exception e) {
355               }
356             }
357
358           };
359           sw.execute();
360         }
361       }
362     });
363     btnStart.setFont(new Font("Calibri", Font.PLAIN, 24));
364     GridBagConstraints gbc_btnNewButton1 = new GridBagConstraints();
365     gbc_btnNewButton1.fill = GridBagConstraints.HORIZONTAL;
366     gbc_btnNewButton1.insets = new Insets(2, 5, 5, 5);
367     gbc_btnNewButton1.gridx = 0;
368     gbc_btnNewButton1.gridy = 1;
369     panel_2.add(btnStart, gbc_btnNewButton1);
370
371     JButton btnStop = new JButton("Stop");
372     btnStop.addActionListener(new ActionListener() {
373       public void actionPerformed(ActionEvent arg0) {
374         cancelSW();
375       }
376     });
377     btnStop.setFont(new Font("Calibri", Font.PLAIN, 24));
378     GridBagConstraints gbc_btnStop = new GridBagConstraints();
379     gbc_btnStop.fill = GridBagConstraints.HORIZONTAL;
380     gbc_btnStop.insets = new Insets(2, 5, 5, 0);
381     gbc_btnStop.gridx = 1;
382     gbc_btnStop.gridy = 1;
383     panel_2.add(btnStop, gbc_btnStop);
384
385     JScrollPane scrollPane = new JScrollPane();
386     scrollPane.setPreferredSize(new Dimension(1, 1));
387     GridBagConstraints gbc_scrollPane = new GridBagConstraints();
388     gbc_scrollPane.insets = new Insets(2, 5, 0, 0);
```

```
389     gbc_scrollPane.fill = GridBagConstraints.BOTH;
390     gbc_scrollPane.gridheight = 4;
391     gbc_scrollPane.gridwidth = 2;
392     gbc_scrollPane.gridx = 0;
393     gbc_scrollPane.gridy = 2;
394     panel_2.add(scrollPane, gbc_scrollPane);
395
396     textArea = new JTextArea();
397     textArea.setAutoscrolls(false);
398     textArea.setLineWrap(true);
399     textArea.setWrapStyleWord(true);
400     textArea.setFont(new Font("Calibri", Font.PLAIN, 20));
401     scrollPane.setViewportView(textArea);
402
403     JCheckBox chckbxNewCheckBox = new JCheckBox("Visualisation");
404     chckbxNewCheckBox.addActionListener(new ActionListener() {
405       public void actionPerformed(ActionEvent arg0) {
406         checked = checked? false:true;
407         vis = checked? true:false;
408       }
409     });
410     scrollPane.setColumnHeaderView(chckbxNewCheckBox);
411     chckbxNewCheckBox.setSelected(true);
412     chckbxNewCheckBox.setFont(new Font("Calibri", Font.PLAIN, 18));
413     chckbxNewCheckBox.setVerticalAlignment(SwingConstants.BOTTOM);
414     chckbxNewCheckBox.setHorizontalAlignment(SwingConstants.CENTER);
415   }
416
417   /*
418    * launches the application.
419    */
420   public static void main(String[] args) {
421     EventQueue.invokeLater(new Runnable() {
422       public void run() {
423         try {
424           APGUI frame = new APGUI();
425           frame.setVisible(true);
426         } catch (Exception e) {
427         }
428       }
429     });
430   }
431 }
```

Graph generation

```
1  public class GraphGenaratorMain {
2    private Graph graph;
3
4    public GraphGenaratorMain(int type) {
5      switch(type) {
6      case 0: makePlanarGraph();
7      break;
8      case 1: makeNonPlanarGraph();
```

```
9       break;
10      default: if (Math.random() < 0.5) makePlanarGraph();
11      else makeNonPlanarGraph();
12      }
13    }
14
15    public GraphGenaratorMain(int type, double[] pars) {
16      switch(type) {
17      case 0: makePlanarGraph(pars);
18      break;
19      case 1: makeNonPlanarGraph(pars);
20      break;
21      default: if (Math.random() < 0.5) makePlanarGraph();
22      else makeNonPlanarGraph();
23      }
24    }
25
26    private void makePlanarGraph() {
27      graph = new PlanarGraphGenerator().getGraph();
28    }
29
30    private void makePlanarGraph(double[] pars) {
31      graph = new PlanarGraphGenerator(pars).getGraph();
32    }
33
34    private void makeNonPlanarGraph() {
35      graph = new NonPlanarGraphGenerator().getGraph();
36    }
37
38    private void makeNonPlanarGraph(double[] pars) {
39      graph = new NonPlanarGraphGenerator(pars).getGraph();
40    }
41
42    public Graph getGraph() {
43      return graph;
44    }
45  }
46
47  public class NonPlanarGraphGenerator {
48    private Graph graph;
49    private int steps;
50
51    private int minSteps = 0;
52    private int maxSteps = 30;
53    private double minAddProb = 0.05;
54    private double maxAddProb = 0.5;
55    private double p0 = 0.2;
56    private double p1 = 0.7;
57    private double pK5 = 0.5;
58
59    NonPlanarGraphGenerator() {
60      start();
61    }
62
```

```
63    NonPlanarGraphGenerator(double[] pars) {
64      minSteps = (int) pars[0];
65      maxSteps = (int) pars[1];
66      minAddProb = pars[2];
67      maxAddProb = pars[3];
68      p0 = pars[4];
69      p1 = pars[5];
70      pK5 = pars[6];
71      start();
72    }
73
74    private boolean compareNode(String node0, String node1) {
75        int a = Integer.parseInt(node0);
76        int b = Integer.parseInt(node1);
77        return a < b;
78    }
79
80    private void makeK33() {
81      for (int i = 0; i < 6; i++)
82        graph.addNode(i + "");
83      for (int i = 0; i < 3; i++)
84        for (int j = 3; j < 6; j++)
85          graph.addEdge(i + " " + j, i + "", j + "");
86    }
87
88    private void makeK5() {
89      for (int i = 0; i < 5; i++)
90        graph.addNode(i + "");
91      for (int i = 0; i < 5; i++)
92        for (int j = i + 1; j < 5; j++)
93          graph.addEdge(i + " " + j, i + "", j + "");
94    }
95
96    private void addEdge() {
97      Node node0 = Toolkit.randomNode(graph);
98      Node node1 = Toolkit.randomNode(graph);
99      if (node0.getIndex() != node1.getIndex()) {
100       String id0 = node0.getId();
101       String id1 = node1.getId();
102       if (!compareNode(id0, id1)) {
103         String temp = id0;
104         id0 = id1;
105         id1 = temp;
106       }
107       if (graph.getEdge(id0 + " " + id1) == null)
108         graph.addEdge(id0 + " " + id1, id0, id1);
109     }
110   }
111
112   private void splitEdge() {
113     Edge edge = Toolkit.randomEdge(graph);
114     Node node0 = edge.getNode0();
115     Node node1 = edge.getNode1();
116     Node node = graph.addNode(graph.getNodeCount() + "");
```

```
117       graph.removeEdge(edge);
118       graph.addEdge(node0.getId() + " " + node.getId(), node0.getId(), node.
              getId());
119       graph.addEdge(node1.getId() + " " + node.getId(), node1.getId(), node.
              getId());
120     }
121
122     private void addNode() {
123       Node node0 = graph.addNode(graph.getNodeCount() + "");
124       double addProb = minAddProb + (maxAddProb - minAddProb) * Math.random
              ();
125       for (int i = 0; i < graph.getNodeCount() - 1; i++) {
126         Node node1 = graph.getNode(i);
127         if (Math.random() < addProb)
128           graph.addEdge(node1.getId() + " " + node0.getId(), node1.getId(),
                  node0.getId());
129       }
130     }
131
132     private void start() {
133       graph = new GraphReader().getEmptyGraph();
134       if (Math.random() < pK5) makeK5();
135       else makeK33();
136       steps = minSteps + (int)(Math.random() * ((maxSteps - minSteps) + 1));
137       for (int i = 0; i < steps; i++) {
138         double rand = Math.random();
139         if (rand < p0) addEdge();
140         else if (rand >= p0 && rand < p1) splitEdge();
141         else addNode();
142       }
143     }
144
145     public Graph getGraph() {
146       return graph;
147     }
148   }
149   public class PlanarGraphGenerator {
150     private Graph graph;
151     private int nLines;
152     private int V;
153     private int E;
154     private Line[] lines;
155
156     private int minL = 3;
157     private int maxL = 9;
158     private double delFactorE = 0.15;
159     private double delFactorV = 0.05;
160     private double eps = 0.000000001;
161
162     PlanarGraphGenerator() {
163       start();
164     }
165
166     PlanarGraphGenerator(double[] pars) {
```

```
167       maxL = (int) pars[0];
168       delFactorE = pars[1];
169       delFactorV = pars[2];
170
171       start();
172     }
173
174     class Pair implements Comparable<Pair> {
175       double x;
176       double y;
177       int ind;
178
179       Pair(double inters[], int ind) {
180         x = inters[0];
181         y = inters[1];
182         this.ind = ind;
183       }
184
185       @Override
186       public int compareTo(Pair other) {
187         return other.x <= x? 1:-1;
188       }
189     }
190
191     class Line {
192       double slope;
193       double yInt;
194       int ind;
195
196       Line(double slope, double yInt, int ind) {
197         this.slope = slope;
198         this.yInt = yInt;
199         this.ind = ind;
200       }
201     }
202
203     private void deleteNodes() {
204       ArrayList<String> toRemove = new ArrayList<>();
205       for (int i = 0; i < V; i++) {
206         if (Math.random() < delFactorV)
207           toRemove.add(graph.getNode(i).getId());
208       }
209       for (int i = 0; i < toRemove.size(); i++)
210         graph.removeNode(toRemove.get(i));
211     }
212
213     private void deleteEdges() {
214       E = graph.getEdgeCount();
215       ArrayList<String> toRemove = new ArrayList<>();
216       for (int i = 0; i < E; i++) {
217         if (Math.random() < delFactorE)
218           toRemove.add(graph.getEdge(i).getId());
219       }
220       for (int i = 0; i < toRemove.size(); i++)
```

```
221         graph.removeEdge(toRemove.get(i));
222     }
223
224     private int getNodeIndex(int p, int q) {
225       p++; q++;
226       return (p - 1) * (2 * nLines - p) / 2 + q - p - 1;
227     }
228
229     private double[] findIntersection(Line l1, Line l2) {
230       double x = (l2.yInt - l1.yInt) / (l1.slope - l2.slope);
231       double y = l1.slope * x + l1.yInt;
232       double[] inters = {x, y};
233       return inters;
234     }
235
236     private void makeGraph() {
237       for (int i = 0; i < V; i++)
238         graph.addNode(i + "");
239       for (int i = 0; i < nLines; i++) {
240         ArrayList<Pair> ordLines = new ArrayList<>();
241         for (int j = 0; j < nLines; j++) {
242           if (i == j) continue;
243           ordLines.add(new Pair(findIntersection(lines[i], lines[j]), lines[
                j].ind));
244         }
245         Collections.sort(ordLines);
246         for (int j = 0; j < nLines - 2; j++) {
247           int n1 = getNodeIndex(Math.min(lines[i].ind, ordLines.get(j).ind),
248               Math.max(lines[i].ind, ordLines.get(j).ind));
249           int n2 = getNodeIndex(Math.min(lines[i].ind, ordLines.get(j + 1).
                ind),
250               Math.max(lines[i].ind, ordLines.get(j + 1).ind));
251           if (n1 > n2) {
252             int temp = n1;
253             n1 = n2;
254             n2 = temp;
255           }
256           graph.addEdge(n1 + " " + n2, n1 + "", n2 + "");
257         }
258       }
259     }
260
261     private double getSlope(double x1, double y1, double x2, double y2) {
262       return (y2 - y1) / (x2 - x1 + eps);
263     }
264
265     private double[] getRandomFour() {
266       double x1 = Math.random(), y1 = Math.random();
267       double x2 = Math.random(), y2 = Math.random();
268       double[] ar = {x1, y1, x2, y2};
269       return ar;
270     }
271
272     private void makeLines() {
```

```
273     ArrayList<Double> slopes = new ArrayList<>();
274     for (int i = 0; i < nLines; i++) {
275       double[] r = getRandomFour();
276       double slope = getSlope(r[0], r[1], r[2], r[3]);
277       boolean par = true;
278       while (par == true) {
279         par = false;
280         for (int j = 0; j < slopes.size(); j++)
281           if (Math.abs(slope - slopes.get(j)) < eps) {
282             par = true;
283             break;
284           }
285         if (par) {
286           r = getRandomFour();
287           slope = getSlope(r[0], r[1], r[2], r[3]);
288         }
289       }
290       lines[i] = new Line(slope, r[1] - slope * r[0], i);
291     }
292   }
293
294   private void start() {
295     graph = new GraphReader().getEmptyGraph();
296     nLines = minL + (int)(Math.random() * ((maxL - minL) + 1));
297     V = nLines * (nLines - 1) / 2;
298     lines = new Line[nLines];
299     makeLines();
300     makeGraph();
301     deleteEdges();
302     deleteNodes();
303   }
304
305   public Graph getGraph() {
306     return graph;
307   }
308 }
```

Testing

```
1  public class AnalyseDataSet {
2    private double[] V;
3    private double[] E;
4    private int nGraphs;
5    private BufferedReader reader;
6
7    private long totalCon = 0;
8    private String dataTitle = "planarHOG.txt";
9    private String statsTitle = "HOGStats.txt";
10
11   private boolean isConnected(Graph graph) {
12     if (graph.getNodeCount() == 0) return true;
13     int[] dist = new int[graph.getNodeCount()];
14     for (int i = 0; i < dist.length; i++) dist[i] = -1;
15     ArrayList<Integer> queue = new ArrayList<Integer>();
```

```
16      dist[0] = 0;
17      queue.add(0);
18      for (int i = 0; i < queue.size(); i++) {
19        int k = queue.get(i);
20        for (Edge e: graph.getNode(k).getEachEdge()) {
21          int j = e.getOpposite(graph.getNode(k)).getIndex();
22          if (dist[j] >= 0) continue ;
23          dist[j] = 0;
24          queue.add(j);
25        }
26      }
27      int sum = 0;
28      for (int i = 0; i < dist.length; i++) sum += dist[i];
29      return sum == 0;
30    }
31
32    private void exportStats() throws IOException {
33      StringBuilder str = new StringBuilder();
34      for (int i = 0; i < V.length; i++)
35        str.append(V[i] + " " + E[i] + "\n");
36      BufferedWriter writer = new BufferedWriter(new FileWriter(statsTitle))
          ;
37      writer.write(str.toString());
38      writer.close();
39    }
40
41    private void read() throws IOException {
42      String line = null;
43      StringBuilder sb;
44      try {
45        for (int i = 0; i < nGraphs; i++) {
46          sb = new StringBuilder();
47          int n = Integer.parseInt(reader.readLine());
48          sb.append(n + "\n");
49          for (int j = 0; j < n; j++) {
50            line = reader.readLine();
51            sb.append(line + "\n");
52          }
53          Graph graph = new GraphReader().read(1, sb.toString());
54          V[i] = graph.getNodeCount();
55          E[i] = graph.getEdgeCount();
56          if (isConnected(graph)) totalCon++;
57        }
58      } finally {
59        reader.close();
60      }
61    }
62
63    private void endStats() {
64      double totalV = 0, totalE = 0, totalV2 = 0, totalE2 = 0;
65      for (int i = 0; i < nGraphs; i++) {
66        totalV += V[i];
67        totalE += E[i];
68      }
```

```
69      double meanV = totalV / nGraphs;
70      double meanE = totalE / nGraphs;
71      for (int i = 0; i < nGraphs; i++) {
72        totalV2 += (V[i] - meanV) * (V[i] - meanV);
73        totalE2 += (E[i] - meanE) * (E[i] - meanE);
74      }
75      double stdV = Math.sqrt(totalV2 / (nGraphs - 1));
76      double stdE = Math.sqrt(totalE2 / (nGraphs - 1));
77      System.out.println(nGraphs + " graphs");
78      System.out.println(totalCon + " connected graphs");
79      System.out.println("Mean #nodes: " + meanV);
80      System.out.println("Mean #edges: " + meanE);
81      System.out.println("Std #nodes: " + stdV);
82      System.out.println("Std #edges: " + stdE);
83    }
84
85    private void start() throws IOException {
86      reader = new BufferedReader(new FileReader(dataTitle));
87      nGraphs = Integer.parseInt(reader.readLine());
88      V = new double[nGraphs];
89      E = new double[nGraphs];
90      read();
91      endStats();
92      exportStats();
93    }
94
95    public static void main(String[] args) throws IOException {
96      new AnalyseDataSet().start();
97    }
98  }
99
100 public class HOGConverter {
101   private BufferedReader br = new BufferedReader(new InputStreamReader(
          System.in));
102   private static StringBuilder str = new StringBuilder();
103
104   private String fileName = "planarHOG";
105
106   private void start() throws IOException {
107     String s = br.readLine().replaceAll("\\s+","");
108     int l = s.length();
109     str.append(l);
110     str.append("\n");
111     str.append(s);
112     str.append("\n");
113     for (int i = 1; i < l; i++) {
114       str.append(br.readLine().replaceAll("\\s+",""));
115       str.append("\n");
116     }
117     br.readLine();
118   }
119
120   private void main() throws IOException {
121     for (int i = 0; i < 902; i++) {
```

```
122         start();
123         System.out.println(i);
124       }
125       BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));
126       writer.write(str.toString());
127       writer.close();
128     }
129
130     public static void main(String[] args) throws IOException {
131       new HOGConverter().main();
132     }
133   }
134
135   public class MakeDataSet {
136     private double[] V;
137     private double[] E;
138
139     private long totalCon = 0;
140     private long totalDense = 0;
141     private long totalPlanar = 0;
142     private StringBuilder sbGraph = new StringBuilder();
143     private StringBuilder sbSol = new StringBuilder();
144
145     private int n = 10000;
146     private int graphType = 1; //0 for planar, non-planar otherwise
147
148     private boolean exportDataSet = false;
149     private boolean exportStats = true;
150     private boolean random = false;
151     private String dataTitle = "testSet.txt";
152     private String statsTitle = "stats.txt";
153     private String solTitle = "solution.txt";
154
155     //planar graph generation
156     private double[] pars1 = {21, 0.10, 0.05};
157
158     //non-planar graph generation
159     private double[] pars2 = {0, 200, 0.05, 0.15, 0.2, 0.8, 0.5};
160
161     private String toMatrix(Graph graph) {
162       StringBuilder str = new StringBuilder();
163       str.append(graph.getNodeCount() + "\n");
164       for (int i = 0; i < graph.getNodeCount(); i++) {
165         for (int j = 0; j < graph.getNodeCount(); j++) {
166           if (graph.getNode(i).hasEdgeBetween(j))
167             str.append("1");
168           else str.append("0");
169         }
170         str.append("\n");
171       }
172       return str.toString();
173     }
174
175     private void exportGraphs() throws IOException {
```

```
176       BufferedWriter writer = new BufferedWriter(new FileWriter(dataTitle));
177       writer.write(sbGraph.toString());
178       writer.close();
179     }
180
181     private void exportSols() throws IOException {
182       BufferedWriter writer = new BufferedWriter(new FileWriter(solTitle));
183       writer.write(sbSol.toString());
184       writer.close();
185     }
186
187     private boolean isConnected(Graph graph) {
188       if (graph.getNodeCount() == 0) return true;
189       int[] dist = new int[graph.getNodeCount()];
190       for (int i = 0; i < dist.length; i++) dist[i] = -1;
191       ArrayList<Integer> queue = new ArrayList<Integer>();
192       dist[0] = 0;
193       queue.add(0);
194       for (int i = 0; i < queue.size(); i++) {
195         int k = queue.get(i);
196         for (Edge e: graph.getNode(k).getEachEdge()) {
197           int j = e.getOpposite(graph.getNode(k)).getIndex();
198           if (dist[j] >= 0) continue ;
199           dist[j] = 0;
200           queue.add(j);
201         }
202       }
203       int sum = 0;
204       for (int i = 0; i < dist.length; i++) sum += dist[i];
205       return sum == 0;
206     }
207
208     private void exportStats() throws IOException {
209       StringBuilder str = new StringBuilder();
210       for (int i = 0; i < n; i++)
211         str.append(V[i] + " " + E[i] + "\n");
212       BufferedWriter writer = new BufferedWriter(new FileWriter(statsTitle))
          ;
213       writer.write(str.toString());
214       writer.close();
215     }
216
217     private void endStats() {
218       double totalV = 0, totalE = 0, totalV2 = 0, totalE2 = 0;
219       for (int i = 0; i < n; i++) {
220         totalV += V[i];
221         totalE += E[i];
222       }
223       double meanV = totalV / n;
224       double meanE = totalE / n;
225       for (int i = 0; i < n; i++) {
226         totalV2 += (V[i] - meanV) * (V[i] - meanV);
227         totalE2 += (E[i] - meanE) * (E[i] - meanE);
228       }
```

```
229       double stdV = Math.sqrt(totalV2 / (n - 1));
230       double stdE = Math.sqrt(totalE2 / (n - 1));
231       System.out.println(n + " graphs");
232       System.out.println(totalCon + " connected graphs");
233       System.out.println(totalDense + " dense graphs");
234       System.out.println(totalPlanar + " planar graphs");
235       System.out.println("Mean #nodes: " + meanV);
236       System.out.println("Mean #edges: " + meanE);
237       System.out.println("Std #nodes: " + stdV);
238       System.out.println("Std #edges: " + stdE);
239     }
240
241     private void getData() throws IOException {
242       V = new double[n];
243       E = new double[n];
244       sbGraph.append(n + "\n");
245       for (int i = 0; i < n; i++) {
246         if (random) {
247           graphType = (int) Math.round(Math.random());
248           sbSol.append((graphType == 0) + "\n");
249         }
250         Graph graph = new GraphGenaratorMain(graphType, graphType == 0?
              pars1:pars2).getGraph();
251         V[i] = graph.getNodeCount();
252         E[i] = graph.getEdgeCount();
253         if (isConnected(graph)) totalCon++;
254         if (E[i] > V[i]* 3 - 6 && V[i] > 2) totalDense++;
255         if (graphType == 0) totalPlanar++;
256         if (exportDataSet) sbGraph.append(toMatrix(graph));
257       }
258       if (exportStats) exportStats();
259       if (exportDataSet) exportGraphs();
260       if (random && exportDataSet) exportSols();
261       endStats();
262     }
263
264     public static void main(String[] args) throws IOException {
265       new MakeDataSet().getData();
266     }
267   }
268
269   public class Test {
270     private String dataTitle = "testSet.txt";
271     private String solTitle = "solution.txt";
272     private String timeTitle = "times.txt";
273     private boolean[] planarSol;
274     private boolean correct = true;
275     private StringBuilder sbTime = new StringBuilder();
276
277     private void exportTimes() throws IOException {
278       BufferedWriter writer = new BufferedWriter(new FileWriter(timeTitle));
279       writer.write(sbTime.toString());
280       writer.close();
281     }
```

```
282
283   private String readFile(String file) throws IOException {
284     BufferedReader reader = new BufferedReader(new FileReader (file));
285     String line = null;
286     StringBuilder stringBuilder = new StringBuilder();
287     try {
288       while((line = reader.readLine()) != null) {
289         stringBuilder.append(line);
290         stringBuilder.append(" ");
291       }
292       return stringBuilder.toString();
293     } finally {
294       reader.close();
295     }
296   }
297
298   private void test(int nGraphs) throws IOException {
299     BufferedReader reader = new BufferedReader(new FileReader (dataTitle))
          ;
300     String line = null;
301     StringBuilder sb;
302     try {
303       reader.readLine();
304       for (int i = 0; i < nGraphs; i++) {
305         sb = new StringBuilder();
306         int n = Integer.parseInt(reader.readLine());
307         sb.append(n + "\n");
308         for (int j = 0; j < n; j++) {
309           line = reader.readLine();
310           sb.append(line + "\n");
311         }
312         Graph graph = new GraphReader().read(1, sb.toString());
313         AuslanderParter algo = new AuslanderParter(graph, false);
314         long time = System.currentTimeMillis();
315         boolean planar = algo.compute();
316         time = System.currentTimeMillis() - time;
317         sbTime.append((double) time + " " + graph.getNodeCount() + " " +
              graph.getEdgeCount() + "\n");
318         if (planar ^ planarSol[i]) {
319           correct = false;
320           System.out.println("Error at graph: " +i);
321           System.out.println("Graph is " + (planarSol[i]? "planar":"not
                planar"));
322           System.out.println(sb.toString());
323           System.out.println();
324         }
325       }
326     } finally {
327       reader.close();
328     }
329   }
330
331   private void start() throws IOException {
332     String[] solution = readFile(solTitle).split("\\s+");
```

```
333       planarSol = new boolean[solution.length];
334       for (int i = 0; i < solution.length; i++)
335         planarSol[i] = Boolean.parseBoolean(solution[i]);
336       test(solution.length);
337       exportTimes();
338       System.out.println(correct? "no errors\ndone":"done");
339     }
340
341     public static void main(String[] args) throws IOException {
342       new Test().start();
343     }
344   }
345
346   public class Timer {
347     BufferedReader br = new BufferedReader(new InputStreamReader(System.in))
          ;
348     long n = 1000000;
349     long m = 1000000;
350
351     void compute() {
352       for (long i = 0; i < m; i++) ;
353     }
354
355     void start() throws IOException {
356       long total = 0;
357       for (long i = 0; i < n; i++) {
358         long time = System.currentTimeMillis();
359         compute();
360         time = System.currentTimeMillis() - time;
361         total += time;
362       }
363       System.out.println((double) total / n);
364     }
365
366     public static void main(String[] args) throws IOException{
367       new Timer().start();
368     }
369   }
```