TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

"Algoriths and Data Structures" Task: Biconnecting Graphs

Author: Sandra Örd
Student code: 223277IADB
Programme: IT-System Development
Supervisor: Jaanus Pöial

Tallinn 2023

Table of Contents

1 Task 5
2 Description of the Solution 6
2.1 Solution Description using an Example 6
2.1.1 Finding the Discovery Times of the Vertices 7
2.1.2 Finding the low-link values of vertices 7
2.1.3 Identifying Articulation Points 9
2.1.4 Adding edges to eliminate articulation points 11
3 Program Usage Guide 12
4 Test Plan 14
4.1 Small Graphs Without Articulation Points 14
4.2 Full Graph 15
4.3 Simple Cycle Graph 15
4.4 Simple Chain Graph 16
4.5 Graph With a Root Vertex Articulation Point 17
4.6 Articulation points in more complex graphs 18
4.7 Complex graph 20
References 21
Appendix 1 – Test Result Examples 22

List of Figures

Figure 1. The removal of articulation points increases the number of connected components. 5
Figure 2. Adding an arc ensures the biconnectivity of the graph. 5
Figure 3. Example graph. 6
Figure 4. Example graph with depth-first search path and discovery times. 7
Figure 5. Example graph with articulation points marked. 10
Figure 6. Example graph with added edges. 11
Figure 7. Graph with one vertex. 14
Figure 8. Graph with two vertices. 14
Figure 9. Full graph. 15
Figure 10. Simple cycle graph. 15
Figure 11. Simple chain graph. 16
Figure 12. Simple chain graph with added edges. 16
Figure 13. Graph, where the root vertex is an articulation point. 17
Figure 14. Graph, where the root vertex is an articulation point, with added edges. 17
Figure 15. Example graph with cycles. 18
Figure 16. Example graph with cycles with added edges. 18
Figure 17. Example graph without cycles. 19
Figure 18. Example graph without cycles with added edges. 19

List of Tables

Table 1. Finding the low-link values of vertices. 9
Table 2. Identifying articulation points. 10

1 Task

Computer networks should avoid nodes whose failure would break the network's connectivity. Representing a computer network as a graph, where nodes are vertices and connections between nodes are edges, the task is to identify all vertices whose removal would split the graph into two or more components. In other words, removing such a vertex would disconnect the graph (see Figure 1). Figure 1 illustrates, how the removal of vertex C (with the edges connected to it) causes the graph to separate into two separate graphs. These critical single point of failure vertices are called articulation points. [1]

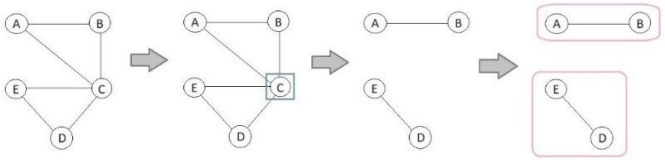

Figure 1. The removal of articulation points increases the number of connected components.

In the current task's context, the graph is a simple graph, meaning it is undirected and does not contain loops, also known as self-loops (edges that connect a vertex to itself). An undirected edge is represented by two arcs – one for each direction.

Let the number of vertices in a graph be $n \geq 3$, and the number of edges be $m \geq n - 1$. The value $n - 1$ is the minimum number of edges required for a graph with n vertices to remain connected. The minimum number of vertices for which a graph can have an AP is 3.

Using an algorithm with complexity $O(n + m)$, it is possible to add at most n edged to the graph to biconnect the graph (see Figure 2). Biconnecting is the process of eliminating articulation points so that the removal of any single vertex does not disconnect the graph. Figure 2 shows that adding an edge between the components dormed by the removal of vertex C ensures that the graph remains connected, even if the former AP is removed.

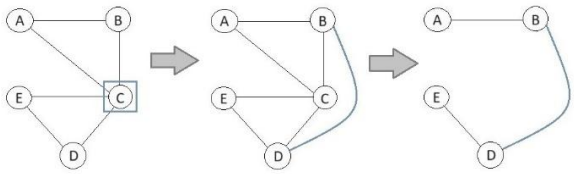

Figure 2. Adding an arc ensures the biconnectivity of the graph.

2 Description of the Solution

To solve the single point of failure problem of graphs, an algorithm for finding articulation points was used, which is based on depth-first search (hereinafter also referred to as DFS) and Tarjan's algorithm. This method compares the discovery time of a vertex (the sequence order number in which the vertex was found via DFS) and its low-link values of the vertex' descendants in the DFS tree. Due to the nature of depth-first search – each visited vertex is not revisited – every node in the DFS tree has exactly one direct parent, but may have multiple children. [2]

A failure at an articulation point (hereinafter also referred to as AP) can disrupt the connectivity of the graph. To mitigate this risk, additional edges are added from the parent of the AP to its vulnerable children nodes. This ensures there is an alternate path to the AP's children that does not pass through the articulation point itself.

2.1 Solution Description using an Example

An example graph with 17 vertices and 20 edges is used to aid the description of the solution (see Figure 3). The solution to the problem consists of two main steps. First, the articulation points must be identified. Second, new edges must be added in a way that after their introduction to the graph, no articulation points remain in the graph.

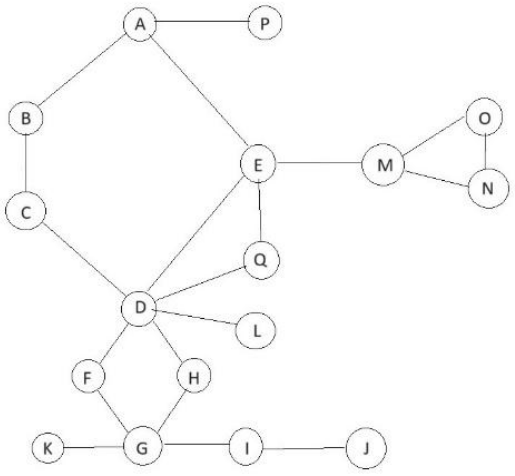

Figure 3. Example graph.

2.1.1 Finding the Discovery Times of the Vertices

In order to find articulation points, the graph is first traversed using depth-first search, and the discovery number (discovery time) of each vertex is recorded (see Figure 4). This order is stored in the *Integer discovery* field of each vertex. This step completes one part of the Tarjan's algorithm – recording the discovery times of the vertices.

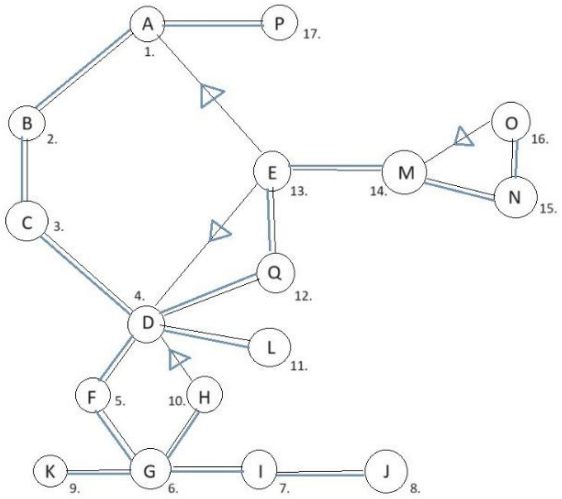

Figure 4. Example graph with depth-first search path and discovery times.

2.1.2 Finding the low-link values of vertices

To compute a vertex's low-link value – stored in the *Integer low* field of the vertex – the minimum value of three quantities must be found:

- The vertex's own discovery time (*discovery*)
- The smallest *low* value among the vertex's children in the DFS tree
- The smallest discovery time reached by any back edge that originated from the vertex

The *Vertex* class' *low()* method is used to calculate this value once all the necessary information has been collected.

In Figure 4, every edge traversed by depth-first search is shown in blue. Any other edge that is present in the original graph but missing from the DFS tree closes a cycle and is therefore a back edge. A back edge is an edge that points from the current vertex to an already discovered ancestor. The back edge discovery time or back edge dfn is the discovery number of its destination.

For example, the arc D → H is not a back edge, because the discovery time of H is higher than the discovery time of D, while H → D does meet the requirement to be a back edge. Due to this, Figure 4 also illustrates the direction of the back edge with an arrow. The back edge dfn of the arc H → D, is the discovery number of D = 4.

For every vertex, that has outgoing back edges, the smallest reachable destination number is recorded. In the example illustrated on Figure 4, the vertex E has two back edges with destination numbers 1 and 4, so it stores the back edge dfn is recorded with the value 1. Similarly vertex O stores the value 14 and vertex H stores 4. In the source code, each vertex keeps a *List<Integer> backEdgesDfn* to store the back edge dfn values. The method *minBackEdgeDfn()* is used to retrieve the minimum of that list when the *low* value is being calculated using the *low()* method.

A vertex's low value depends on the low values of its children as well, it is necessary to know the lowest low-link value of the children of the vertex in the DFS tree. In other words, to find out the low-link value of a vertex, the low-link value of all its children must be known. Therefore, the calculation must start at the leaves of the DFS tree – vertices with no children – whose low-link values equal their own discovery number (due to the lack of children the children's lowest low-link value does not exist). Once these values are recorded, it is possible to move upward, computing each parent's low-link value only after all outgoing edges from that vertex have been processed. Internally, *low()* method invoked *dfsChildLowValue()*, which returns the minimum low value among the vertex's children.

Table 1 is ordered in an order that is suitable for low-link value computing. The order corresponds to the sequence in which the recursive DFS unwinds. The column *id* is used to identify the vertex for which the table row's values correspond to. The table also states the children nodes of each vertex and all the back edges. The minimum value among the *discovery time, backEdgesDfns, childrenLowValues* is the low-link value of the vertex and is written into the *Low* column. The *childrenLowValues* are taken from the previous rows of the table, since the children's low-link values must be found before that of the vertex itself. The cells for missing values have been colored gray, to better emphasize these rows, since calculating the *low-link* value for these cases depends on fewer values.

Table 1. Finding the low-link values of vertices.

			Low = min(discovery, backEdgesDfns, childrenLowValues)				
seq	id	Low	discovery time	backEdgesDfns	childrenLowValues	backEdges	children
1	J	8	8				
2	I	7	7		8		J
3	K	9	9				
4	H	4	10	4		H→D	
5	G	4	6		4, 7, 9		I, K, H
6	F	4	5		4		G
7	L	11	11				
8	O	14	16	14		O→M	
9	N	14	15		14		O
10	M	14	14		14		N
11	E	1	13	1, 4	14	E→A, E→D	M
12	Q	1	12		1		E
13	D	1	4		4, 11, 1		F, L, Q
14	C	1	3		1		D
15	B	1	2		1		C
16	P	17	17				
17	A	1	1		1, 17		B, P

2.1.3 Identifying Articulation Points

The root vertex of a graph is an articulation point if and only if it has more than one child in the DFS tree. This indicates that, starting from one child, it is not possible to reach another child by any path that does not pass through the root. If the root has only one child, then it is not an articulation point.

Additionally, leaf vertices in the DFS tree, with no children of their own, are not articulation points. Removing them decreases the number of vertices in a graph, but does not affect connectivity of the rest of the graph.

A vertex is an articulation point if its own discovery value is less than or equal to the low-link value of at least one of its children in the DFS tree. The removal of such a vertex, makes the children nodes, for which this condition was true, unreachable from the starting point of the traversal.

The order used in Table 1, which is based on the exit sequence of the DFS recursion, is also used in Table 2. Table 2 shows each vertex's discovery time value, the vertex's children and their low-link values along, the relevant comparison results and conlusions about whether the vertex is an articulation point. If at least one of the children of the vertex has a lower or equal low-link value than the discovery number of the vertex itself, then the vertex is an AP. Vertices with no children are not APs and the root vertex A is an AP only if it has more than one child.

Table 2. Identifying articulation points.

jrk	id	discovery		child.low	children	Articulation point?		Reason
1	J	8				no		J is leaf
2	I	7	≤	J.low = 8	J	yes	AP	
3	K	9				no		K is leaf
4	H	10				no		H is leaf
5	G	6	≤	I.low = 7 K.low = 9 H.low = 4	I K H	yes yes no	AP	
6	F	5	≤	G.low = 4	G	no		
7	L	11				no		L is leaf
8	O	16				no		O is leaf
9	N	15	≤	O.low = 14	O	no		
10	M	14	≤	N.low = 14	N	yes	AP	
11	E	13	≤	M.low = 14	M	yes	AP	
12	Q	12	≤	E.low = 1	E	no		
13	D	4	≤	F.low = 4 L.low = 11 Q.low = 1	F L Q	yes yes no	AP	
14	C	3	≤	D.low = 1	D	no		
15	B	2	≤	C.low = 1	C	no		
16	P	17				no		P is leaf
17	A	1			B P	yes	AP	A is root with more than one child

Based on the previously stated condition for identifying articulation points, a vertex is considered an articulation point if at least one of its children satisifed the inequality *discovery time of the vertex* ≤ *low-link value of the child*, and according to the results in Table 2, the articulation points in the graph are vertices D, E, G, I, M. The root vertex of the graph A has more than one child and is therefore also an articulation point. Figure 5 demonstrates the articulation points – surrounded with a green rectangle – and the paths to the children, for whom the inequality condition was true – with a green edge – which demostrates the DFS tree children who are endangered by the single point of failure.

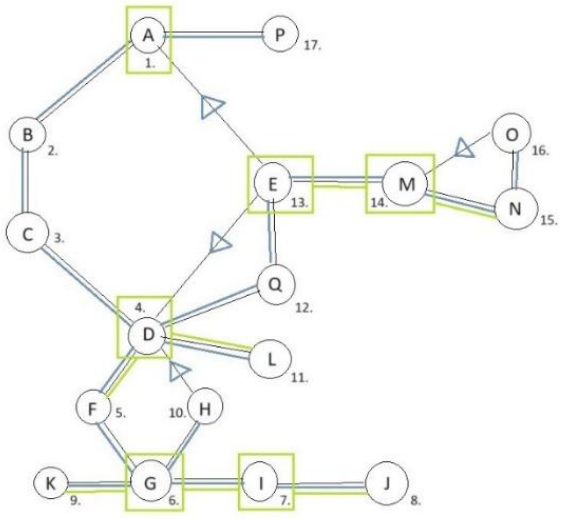

Figure 5. Example graph with articulation points marked.

2.1.4 Adding edges to eliminate articulation points

A maximum of *n* edges can be added to a graph with *n* vertices in order to eliminate all articulation points. Since in a depth-first search each vertex is processed only once and has only one parent, at most one new edge per vertex is needed – giving an upper limit of *n* added edged.

If the root vertex is an articulation point, new edges can be added between its children. For example, if the root has more than two children in the DFS tree, edges can be added between the first and second child, the second and third child, and so on.

For non-root articulation points, it is important to identify the parent vertex of the AP in the DFS tree. New edges are added from the parent of the AP to the AP's children in the DFS true that satisfy the condition *discovery time of AP ≥ low-link value of the child*.

For this task, new edges are added between the parent of each articulation point to the children who meet the mentioned condition, which are demonstrated by using the green line from the AP to the child vertex (see Figure 6). Additionally, an edge is added to connect two of the root's DFS tree children. In total, 8 new edges were added to the example graph.

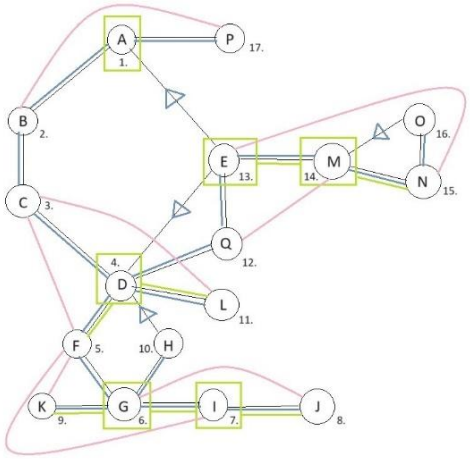

Figure 6. Example graph with added edges.

It is important to note that while the articulation points in a graph are always uniquely determined, then the possible edges to add in order to eliminate them are not. This solution aims to eliminate the articulation points by adding edges, but does not determine what is the best set of edges to add.

3 Program Usage Guide

The task is solved by the *biConnectGraph()* method of the *Graph* class. This method returns the set of articulation points found in the graph. The AP set is always the same regardless of the order in which the graph is traversed. However, the specific edges added during the method may differ depending on the order in which edges are explored from each vertex.

To use the program, first a new graph instance must be created. Then the internal structure of the graph has to be built. There are 2 two options to achieve this: create a random simple graph, using the *Graph* class method *createRandomSimpleGraph(int n, int m)*, to create a graph with *n* vertices and *m* edges or create a graph from a specified structure using the *createGraphFromStructure(LinkedHashMap<String, List<String>> structure)* method, where the structure maps each vertex name (key) to a list of vertex names (values) representing the outgoing vertices the edges outgoing from the key vertex must connect to.

For an existing graph the *biConnectGraph()* method can be called on its instance. This method finds the articulation points of the graph and adds edges as needed to ensure the graph remains connected even if any of the points were removed. The method returns the set of articulation points as *Vertex* objects. For the best comparison, it is recommended to display the graph structure on the screen before and after the method is applied to highlight the added new edges.

It is also possible to control the output's verbosity by using the overload *biConnectGraph(boolean print, boolean detailed)*. This allows to display how many articulation points were found and which vertices were identified as APs. The method has two boolean type parameters. If the first value *print* is set to false, the method call is equivalent to calling the method without parameters. If the *print* value is true but *detailed* value is false, then only the number of articulation points is displayed – this is especially useful for large graphs where the full details may be overwhelming and not very informative by themselves. If both parameters are set to true, then the number of found

APs is displayed along with which vertices are articulation points and the names of the edges that were added to the graph.

The class *GraphBiConnector* has a helper method *printGraphBeforeAndAfter(String graphName, LinkedHashMap<String, List<String>> graphStructure)*. This method creates a graph with the provided name and structure, prints the intial graphs structure, applies the *biConnectGraph()* and finally prints the modified graph after edges are added.

Additionally, the *GraphBiConnector* class has a *bigGraphTimeMeasuring(String title, String graphName, int i, int m)* helper method. This method requires a title to specify the information about the graph for which time is being measured. As previously, this method also needs a name for the graph and furthermore the number of vertices *n* and the number of edges to add *m*. This method creates a random graph, which has *n* vertices and *m* edges, applies the *biConnectGraph()* method on it and measures how long the program takes to complete. In the end, the number of articulation points found is displayed on the screen and how many milliseconds the process took.

4 Test Plan

During the testing of the program, it is important to verify that articulation points are correctly identified in edge cases – specifically, when the graph has either the minimum or maximum allowed number of edges. The test scenarios should include cases where no articulation points exist in the graph, to make sure the method does not find them where they do not exist, but also cases with articulation points to verify the ability of the method to find them, simple examples to test the method on regular graph forms, but also on more complex graphs to ensure it handles the DFS and calculations correctly.

In the following examples, vertex A is the root vertex in each graph.

4.1 Small Graphs Without Articulation Points

An articulation point can exist in a graph only if it has at least three vertices. In graphs with fewer vertices, all nodes are either leaves, which can't be articulation points or roots with less than 2 children, which also do no meet the criteria to be an AP.

A graph with only one vertex (see Figure 7) can't have any articulation points. Removing the single vertex leaves no remaining nodes, which means the graph is not divided into multiple connected components.

Figure 7. Graph with one vertex.

Articulation points are also not possible in a graph with only two vertices (see Figure 8). Removing one of the vertices leaves a single remaining vertex. The graph can't be split into multiple connected components, so no articulation points can exists.

Figure 8. Graph with two vertices.

Both graphs depicted on Figure 7 and Figure 8 have their structure printouts in the table *"Test Examples Results"* under the names *"Small Graph with one vertex"* and *"Small Graph with two vertices"*, respectively (see Appendix 1). The graph creation code is in the *run()* method of the *GraphBiConnector* class. The test case that verifies the absence of articulation points in these graphs is called *biConnectVerySmallGraph()*.

4.2 Full Graph

The test case where the graph is a full graph – a graph in which all vertices have edges connecting it to every other vertex in the graph (see Figure 9). In such a graph, no articulation poionts can exist, because the removal of any single vertex does not break the connectivity of the graph – all possible connections between vertices already exist.

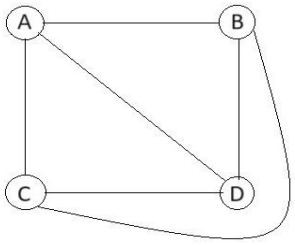

Figure 9. Full graph.

The creation of the full graph is also demonstrated in the *run()* method under the name *"Full Graph"* and the output of the graph structure before and after biconnecting can be seen in the table *"Test Examples Results"* (see Appendix 1). Additionally, the test *biConnectFullGraph()* is used to verify that no articulation points are found in the complete graph by checking that the resulting set is empty.

4.3 Simple Cycle Graph

The algorithm implementation could also be tested on a graph forming a simple cycle – a closed chain where no vertices are repeated upon traversal and the chain starts and ends at the same vertex (see Figure 10). Such a graph does not have any APs since removing any vertex still leaves a simple chain graph that allows for reaching all other vertices.

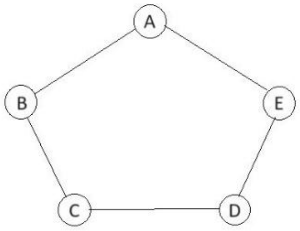

Figure 10. Simple cycle graph.

The creation of the simple cycle graph is also included in the *run()* method, under the name *"Simple Cycle Graph"* for which the printout can be seen in the table *"Test Examples Results"* (see Appendix 1). Additionally, the test *biConnectCycleGraph()* verifies that no articulation points are found in a simple cycle graph.

4.4 Simple Chain Graph

So far the algorithm has been tested on graphs with no articulation points. The algorithm also has to be tested on graphs that do contain APs to validate that it works correctly – rather than doing nothing. A simple example is a simple chain – a chain where no vertices are repeated upon traversal (see Figure 11). When any vertex, other than the endpoints, is removed, the chain splits into two disconnected subgraphs: one containing all the vertices before the removed vertex, and the other containing all vertices after it.

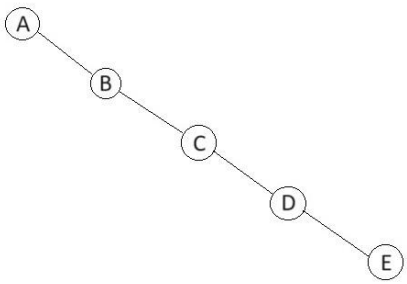

Figure 11. Simple chain graph.

In a simple graph, all vertices except the root and the last leaf are articulation points (see Figure 12). To prevent the graph from separating into multiple subgraphs when such a point fails, edges are added between the vertices immediately preceding and following each AP. This ensures that the graph's connectivity no longer depends solely on individual vertices.

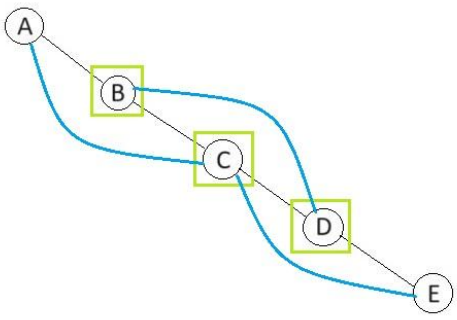

Figure 12. Simple chain graph with added edges.

The source code for creating the simple chain graph is available in the *run()* method under the name *"Simple Chain Graph"*. The simple graph scenario is tested by the *biConnectChainGraph()* test. The original graph and the resulting graph after identifying all the APs and adding new edges is displayed in the table *"Test Examples Results"* (see Appendix 1).

4.5 Graph With a Root Vertex Articulation Point

The condition for the root vertex to be an articulation point differs from that of other vertices. For the root vertex, the condition is that it must have at least two direct descendants in the DFS tree, since these descendants can't be reached by any other path except through the root (see Figure 13).

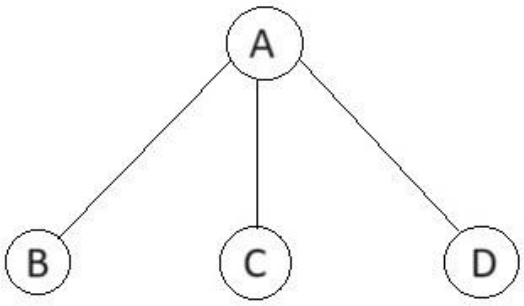

Figure 13. Graph, where the root vertex is an articulation point.

If the root vertex is an AP, its children must be connected to each other to maintain connectivity after the removal of the root vertex (see Figure 14). Removing the root vertex A from the graph in Figure 14 results in a simple chain, ensuring that all vertices remain connected.

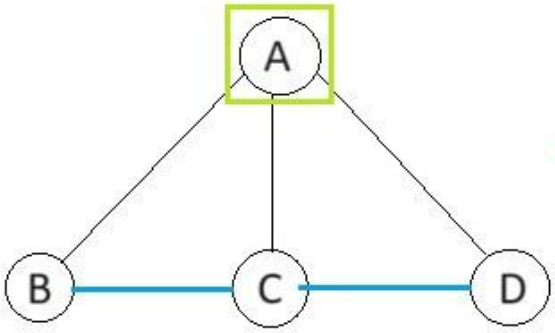

Figure 14. Graph, where the root vertex is an articulation point, with added edges.

The graph depicted in Figure 13 is also created in *GraphBiConnector* class' *run()* method, under the name *"Root AP"* and the output can be seen in the *"Test Examples Results"* table (see Appendix 1). This test case scenario is tested by *biConnectRootAp()* test, which verifies that exactly one articulation point is found for this graph, and that it is the root vertex A.

4.6 Articulation points in more complex graphs

To ensure that the previous results were not correct merely by chance, the method must also be tested on irregular graphs. One such example is the graph shown in Figure 15. In this graph, two vertices C and D act as bottlenecks – any path that connects parts of the graph must pass through them. Consequently, C and D are APs. Removing C disconnects vertices G and H from the rest, while removing D prevents reaching E or F. This example is also useful because the root is adjacent to several vertices, yet each of them can be reached by another route – in the DFS tree the root only has one child and is not an AP.

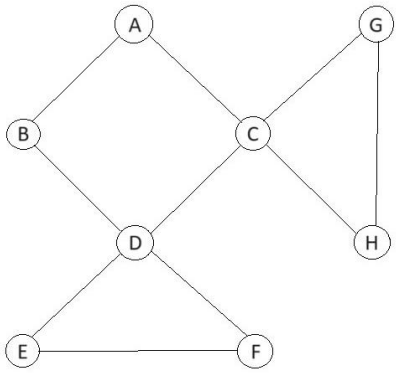

Figure 15. Example graph with cycles.

The source code for creating this graph is in the *run()* method under the name *"Complex Graph With Cycles"* and the graph structure before and after biconnecting is presented in the *"Test Examples Results"* table (see Appendix 1). Test method *biConnectComplexGraphWithCycles()* verifies the vertices that are identified as APs.

Applying the *biConnectGraph()* method on this graph identifies the 2 APs and adds edges to eliminate articulation points from the graph (see Figure 16). Added edges are shown in blue, while the articulation points are oulined in green.

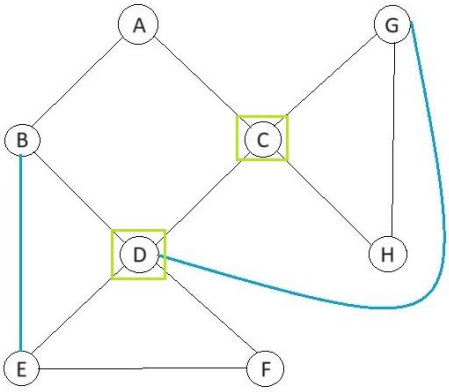

Figure 16. Example graph with cycles with added edges.

The graph mentioned above has multiple cycles, which means that not all edges are represented in the graph's DFS tree. For instance, after reaching vertex C by traversing through A and B, then there is no need to revisit vertex A. For contrast, it would be beneficial to test a more complex graph without cycles, so every edge is traversed during DFS (see Figure 17).

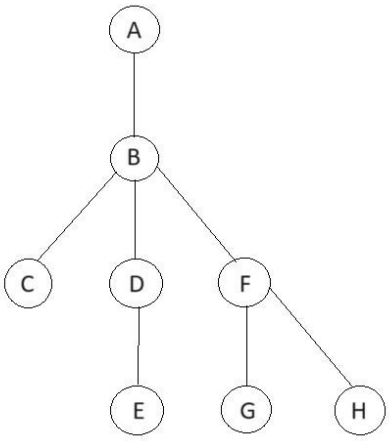

Figure 17. Example graph without cycles.

The articulation points in the graph on Figure 17 are highlighted in green in Figure 18. The edges that were added as the result of the *biConnectGraph()* method are illustrated in blue.

The code for this acyclic graph is available in the *run()* method in the *GraphBiConnector* class under the name *"Complex Graph Without Cycles"* and the console printouts are displayed in the *"Test Examples Results"* table (see Appendix 1). Additionally, this scenario is tested by *biConnectComplexGraphWithoutCycles()* test method.

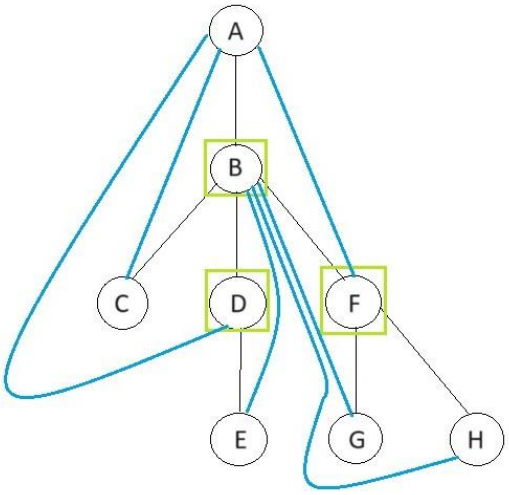

Figure 18. Example graph without cycles with added edges.

4.7 Complex graph

The example graph displayed on Figure 3 is also included in the test plan. The test method *biConnectComplexGraph()* verifies the results of the *biConnectGraph()* method. The example graph used in Chapter 2.1 is useful due to combining many different previously mentioned elements.

The source code for creating the graph is in the *run()* method of the *GraphBiConnector()* class and the console printout results are available in the table *"Test Examples Results"* under the name *"Complex graph"* (see Appendix 1).

References

[1] V. Khyade, „Articulation Point in Graph Algorithm Detail | Graph Theory #20,“ 3 5 2019. [Online]. Available: https://www.youtube.com/watch?v=EvShMOf5CRQ. [Accessed 20 11 2023].

[2] T. Tennisberg, „Graafiteooria,“ [Online]. Available: http://www.targotennisberg.com/eio/VP/vp_ptk10_graafiteooria.pdf. [Accessed 27 11 2023].

Appendix 1 – Test Result Examples

```
Complex example graph
a --> aa_b (a->b) aa_e (a->e) aa_p (a->p)
b --> ab_a (b->a) ab_c (b->c)
c --> ac_b (c->b) ac_d (c->d)
d --> ad_c (d->c) ad_f (d->f) ad_h (d->h) ad_l (d->l) ad_q (d->q) ad_e (d->e)
e --> ae_q (e->q) ae_d (e->d) ae_m (e->m) ae_a (e->a)
f --> af_d (f->d) af_g (f->g)
g --> ag_f (g->f) ag_i (g->i) ag_k (g->k) ag_h (g->h)
h --> ah_g (h->g) ah_d (h->d)
i --> ai_g (i->g) ai_j (i->j)
j --> aj_i (j->i)
k --> ak_g (k->g)
l --> al_d (l->d)
m --> am_e (m->e) am_n (m->n) am_o (m->o)
n --> an_m (n->m) an_o (n->o)
o --> ao_m (o->m) ao_n (o->n)
p --> ap_a (p->a)
q --> aq_d (q->d) aq_e (q->e)

-------------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (6)
Articulation points found: i, g, m, a, e, d
Edges that were added: ab_p, ai_f, aj_g, ac_f, af_c, ap_b, ae_n, an_e, am_q, aq_m, ac_l,
af_i, ak_f, al_c, ag_j, af_k

Complex example graph
a --> aa_b (a->b) aa_e (a->e) aa_p (a->p)
b --> ab_a (b->a) ab_c (b->c) ab_p (b->p)
c --> ac_b (c->b) ac_d (c->d) ac_f (c->f) ac_l (c->l)
d --> ad_c (d->c) ad_f (d->f) ad_h (d->h) ad_l (d->l) ad_q (d->q) ad_e (d->e)
e --> ae_q (e->q) ae_d (e->d) ae_m (e->m) ae_a (e->a) ae_n (e->n)
f --> af_d (f->d) af_g (f->g) af_i (f->i) af_k (f->k) af_c (f->c)
g --> ag_f (g->f) ag_i (g->i) ag_k (g->k) ag_h (g->h) ag_j (g->j)
h --> ah_g (h->g) ah_d (h->d)
i --> ai_g (i->g) ai_j (i->j) ai_f (i->f)
j --> aj_i (j->i) aj_g (j->g)
k --> ak_g (k->g) ak_f (k->f)
l --> al_d (l->d) al_c (l->c)
m --> am_e (m->e) am_n (m->n) am_o (m->o) am_q (m->q)
n --> an_m (n->m) an_o (n->o) an_e (n->e)
o --> ao_m (o->m) ao_n (o->n)
p --> ap_a (p->a) ap_b (p->b)
q --> aq_d (q->d) aq_e (q->e) aq_m (q->m)
```

```
Small Graph with one vertex
a -->

-------------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (0)
Articulation points found:
Edges that were added:

Small Graph with one vertex
a -->
```

```
Small Graph with two vertices
```

```
a --> aa_b (a->b)
b --> ab_a (b->a)

-------------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (0)
Articulation points found:
Edges that were added:

Small Graph with two vertices
a --> aa_b (a->b)
b --> ab_a (b->a)
```

```
Full Graph
a --> aa_b (a->b) aa_c (a->c) aa_d (a->d)
b --> ab_a (b->a) ab_c (b->c) ab_d (b->d)
c --> ac_a (c->a) ac_b (c->b) ac_d (c->d)
d --> ad_a (d->a) ad_b (d->b) ad_c (d->c)

-------------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (0)
Articulation points found:
Edges that were added:

Full Graph
a --> aa_b (a->b) aa_c (a->c) aa_d (a->d)
b --> ab_a (b->a) ab_c (b->c) ab_d (b->d)
c --> ac_a (c->a) ac_b (c->b) ac_d (c->d)
d --> ad_a (d->a) ad_b (d->b) ad_c (d->c)
```

```
Simple Cycle Graph
a --> aa_b (a->b) aa_e (a->e)
b --> ab_a (b->a) ab_c (b->c)
c --> ac_b (c->b) ac_d (c->d)
d --> ad_c (d->c) ad_e (d->e)
e --> ae_d (e->d) ae_a (e->a)

-------------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (0)
Articulation points found:
Edges that were added:

Simple Cycle Graph
a --> aa_b (a->b) aa_e (a->e)
b --> ab_a (b->a) ab_c (b->c)
c --> ac_b (c->b) ac_d (c->d)
d --> ad_c (d->c) ad_e (d->e)
e --> ae_d (e->d) ae_a (e->a)
```

```
Simple Chain Graph
a --> aa_b (a->b)
b --> ab_a (b->a) ab_c (b->c)
c --> ac_b (c->b) ac_d (c->d)
d --> ad_c (d->c) ad_e (d->e)
e --> ae_d (e->d)

-------------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (3)
Articulation points found: c, b, d
Edges that were added: ad_b, ac_e, aa_c, ab_d, ac_a, ae_c

Simple Chain Graph
a --> aa_b (a->b) aa_c (a->c)
b --> ab_a (b->a) ab_c (b->c) ab_d (b->d)
c --> ac_b (c->b) ac_d (c->d) ac_e (c->e) ac_a (c->a)
d --> ad_c (d->c) ad_e (d->e) ad_b (d->b)
```

```
e --> ae_d (e->d) ae_c (e->c)
```

```
Root AP
a --> aa_b (a->b) aa_c (a->c) aa_d (a->d)
b --> ab_a (b->a)
c --> ac_a (c->a)
d --> ad_a (d->a)

--------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (1)
Articulation points found: a
Edges that were added: ac_b, ac_d, ab_c, ad_c

Root AP
a --> aa_b (a->b) aa_c (a->c) aa_d (a->d)
b --> ab_a (b->a) ab_c (b->c)
c --> ac_a (c->a) ac_b (c->b) ac_d (c->d)
d --> ad_a (d->a) ad_c (d->c)
```

```
Complex Graph With Cycles
a --> aa_b (a->b) aa_c (a->c)
b --> ab_a (b->a) ab_d (b->d)
c --> ac_d (c->d) ac_a (c->a) ac_g (c->g) ac_h (c->h)
d --> ad_b (d->b) ad_e (d->e) ad_f (d->f) ad_c (d->c)
e --> ae_f (e->f) ae_d (e->d)
f --> af_d (f->d) af_e (f->e)
g --> ag_c (g->c) ag_h (g->h)
h --> ah_c (h->c) ah_g (h->g)

--------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (2)
Articulation points found: c, d
Edges that were added: ag_d, ad_g, ab_e, ae_b

Complex Graph With Cycles
a --> aa_b (a->b) aa_c (a->c)
b --> ab_a (b->a) ab_d (b->d) ab_e (b->e)
c --> ac_d (c->d) ac_a (c->a) ac_g (c->g) ac_h (c->h)
d --> ad_b (d->b) ad_e (d->e) ad_f (d->f) ad_c (d->c) ad_g (d->g)
e --> ae_f (e->f) ae_d (e->d) ae_b (e->b)
f --> af_d (f->d) af_e (f->e)
g --> ag_c (g->c) ag_h (g->h) ag_d (g->d)
h --> ah_c (h->c) ah_g (h->g)
```

```
Complex Graph Without Cycles
a --> aa_b (a->b)
b --> ab_a (b->a) ab_c (b->c) ab_d (b->d) ab_f (b->f)
c --> ac_b (c->b)
d --> ad_b (d->b) ad_e (d->e)
e --> ae_d (e->d)
f --> af_b (f->b) af_g (f->g) af_h (f->h)
g --> ag_f (g->f)
h --> ah_f (h->f)

--------------------------------------------------------------------------------
Graph after biConnecting:
Articulation point count: (3)
Articulation points found: f, b, d
Edges that were added: aa_f, aa_c, ab_h, ac_a, ab_e, af_a, aa_d, ag_b, ab_g, ah_b, ae_b,
ad_a

Complex Graph Without Cycles
a --> aa_b (a->b) aa_c (a->c) aa_d (a->d) aa_f (a->f)
b --> ab_a (b->a) ab_c (b->c) ab_d (b->d) ab_f (b->f) ab_e (b->e) ab_g (b->g) ab_h (b-
>h)
c --> ac_b (c->b) ac_a (c->a)
d --> ad_b (d->b) ad_e (d->e) ad_a (d->a)
```

```
e --> ae_d (e->d) ae_b (e->b)
f --> af_b (f->b) af_g (f->g) af_h (f->h) af_a (f->a)
g --> ag_f (g->f) ag_b (g->b)
h --> ah_f (h->f) ah_b (h->b)
```

```
-------------------------------------------------------------------------------------------
Big graph example with the most edges (the least articulation points):
Articulation point count: (1)
For a graph with 2222 vertices and 10000 edges, finding and eliminating articulation
points took: 21 ms
-------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------
Big graph example with more edges (some articulation points):
Articulation point count: (413)
For a graph with 2222 vertices and 3333 edges, finding and eliminating articulation
points took: 8 ms
-------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------
Big graph example with the least edges (the most articulation points):
Articulation point count: (1115)
For a graph with 2222 vertices and 2221 edges, finding and eliminating articulation
points took: 10 ms
-------------------------------------------------------------------------------------------
```