TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

"Algoritmid ja andmestruktuurid" individuaaltöö: Graafi SPOF likvideerimine

Autor: Sandra Örd
Üliõpilaskood: 223277IADB
Eriala: IT-süsteemide arendus
Juhendaja: Jaanus Pöial

Tallinn 2023

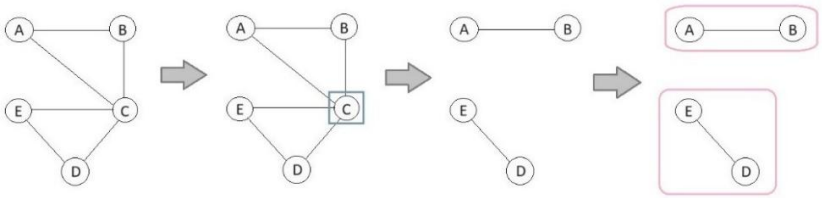
Sisukord

Jooniste loetelu

Tabelite loetelu

1 Ülesande püstitus

Arvutivõrgud peaksid vältima sõlmesid, mille tõrke korral ühendus arvutivõrgus katkeb. Kujutades arvutivõrku graafina, kus sõlmed on tipud ja ühendused sõlmede vahel on kaared, on ülesandeks leida üles kõik tipud, mille eemaldamisel graaf jaguneks kaheks või enamaks sidususkomponendiks. Teisisõnu, tipu eemaldamise tagajärjena ei ole graaf enam sidus (vt Joonis 1). Joonis 1 on näidatud kuidas tipu C (ja temast väljuvate ning temasse suubuvate kaarte) eemaldamise tagajärjel on graaf jagunenud kaheks omavahel ühendamata graafiks. Selliseid tippe nimetatakse artikulatsioonipunktideks. [1]

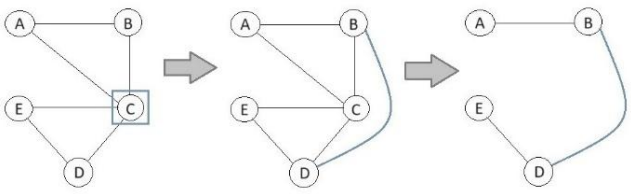


Joonis 1. Artikulatsioonipunkti eemaldamine suurendab sidususkomponentide arvu.

Antud ülesande kontekstis on graafiks lihtgraaf ehk graaf, mis on suunamata ja milles ei leidu silmuseid ehk ühendusi punktist punkti endasse tagasi. Iga serva kujutab kaks kaart – üks mõlema suuna jaoks.

Olgu graafi tippude arv $n \geq 3$ ja servade arv $m \geq n - 1$. Servade arv $n - 1$ on vähim võimalik servade arv, mille korral graaf n tipuga on veel sidus. Tippude arv 3 on vähim võimalik tippude arv, mille puhul võib graafis leiduda artikulatsioonipunkt.

Kasutades algoritmi keerukusega $O(n + m)$ on vaja lisada graafile kõige rohkem n serva, et ühegi üksiku tipu eemaldamisel ei muutuks graaf mittesidusaks (vt Joonis 2). Joonis 2 on näidatud, et serva lisamine tipu C eemaldamisel tekkinud sidususkomponentide vahele, tagab, et enam tipu C eemaldamisel graaf oma sidusust ei kaota.



Joonis 2. Kaare lisamine tagab graafi sidususe ka artikulatsioonipunkti eemaldamisel.

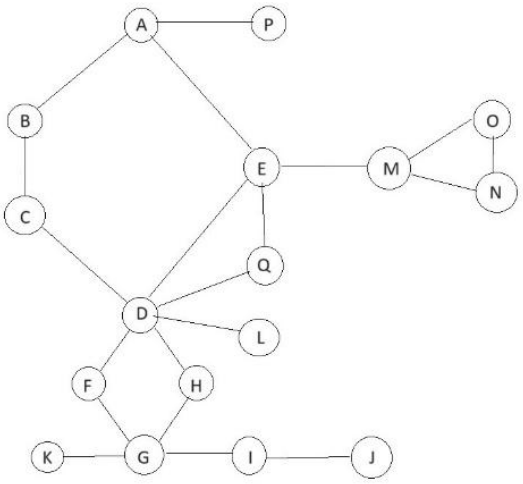
2 Lahenduse kirjeldus

Probleemi lahendamiseks kasutati artikulatsioonipunktide leidmiseks mõeldud algoritmi, mis põhineb graafi sügavuti läbimisel (edaspidi ka DFS ehk *Depth-first search*) ja Tarjani algoritmil. Antud meetodis võrreldakse graafi sügavuti läbimisel leitud tipu järjenumbrit ja tema DFS puus leitud laste *low-link* väärtuseid. Graafi sügavuti läbimise eripärast – punkti, mida on eelnevalt külastatud, uuesti ei külastata – tulenevalt on saadud puus igal punktil ainult üks otsene esivanem. Tipul võib samaaegselt lapsi olla mitmeid. [2]

Artikulatsioonipunktis (edaspidi ka AP) tekkiv tõrge võib graafi sidususe ära lõhkuda. Selle ohu vähendamiseks lisatakse servasid juurde AP vanemast tema ohustatud lasteni. Sellisel juhul on olemas teekond AP lasteni, mis ei läbi artikulatsioonipunkti ennast.

2.1 Lahenduse kirjeldus näite põhjal

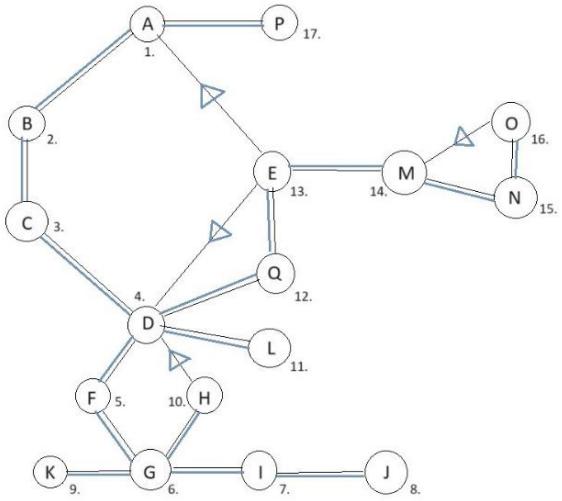
Olgu meie graafiks 17 tipu ja 20 servaga graaf (vt Joonis 3). Ülesande lahendamine koosneb kahest osast. Esmalt tuleb leida graafi artikulatsioonipunktid. Teiseks tuleb graafile juurde lisada servad selliselt, et pärast servade lisamist ühtegi artikulatsioonipunkti enam graafis ei leiduks.



Joonis 3. Näite graaf.

2.1.1 Tippude järjenumbrite leidmine

Artikulatsioonipunktide leidmiseks läbitakse kõige pealt graaf sügavuti ja märgitakse iga tipu juurde mitmendana ta leiti (vt Joonis 4). Tipu järjenumber salvestatakse väljale *Integer discovery*. Sellega on Tarjani algoritmi üks osa – tippude järjenumbrid - leitud.



Joonis 4. Näite graaf koos sügavuti otsingu teekonnaga.

2.1.2 Tippude *low-link* väärtuste leidmine

Järgnev on vajalik, et leida punkti *low-link* väärtus, mis saab iga tipu juures olema salvestatud väljas *Integer low*. *Low* väärtuse leidmiseks leitakse miinimum kolmest väärtusest:

- tipu enda järjenumber *discovery*,
- tipu DFS puu laste seas madalaim *low* väärtus,
- DFS käigus leitud *backedge*'ide madalaim järjenumber.

See väärtus leitakse tipu meetod *low()* abil, kui kogu vajalik info on leitud.

Joonis 4 on siniste joontega ära märgitud kõik servad, mida sügavuti otsing läbis. Kõik ülejäänud servad, mis algses graafis on, kuid mida DFS puus ei ole, tekitavad algses graafis tsükli, ja neid kaari nimetatakse *backedge*'iks. *Backedge* on tipu jaoks, millest see kaar väljub, kaar juba varasemalt leitud tipu juurde. *Backedge*'i avastamise number ehk *backedge dfn* on kaare sihtpunkti number,

Kaar D → H ei ole *backedge*, sest H järjenumber on kõrgem kui D oma, samaaegselt kaar H → D vastab *backedge*'i nõuetele. Seetõttu on joonisel 4 *backedge*'idel noolega ära märgitud ka suund, kumba pidi kaar on. *Backedge dfn* väärtus *backedge* H → D jaoks on D järjenumber ehk 4.

Tippudel, millest väljuvad *backedge*'id, tuleb meelde jätta vähim *backedge*'i avastamise number. Joonisel 4 toodud näites väljub tipust E kaks kaart tagasi, ühe sihtpunkti number on 1 ja teise number 4, nii et E juures jäetakse meelde *backedge dfn* 1. Sarnaselt jäetakse tipu O puhul meelde 14 ja tipu H puhul meelde 4. Programmi lähtekoodis on *backedge* avastamise numbrite meelde jätmiseks igal tipul järjend *List<Integer> backEdgesDfn*, kuhu korjatakse kokku kõik vajalikud numbrid. Arvutades tipu *low* väärtust võetakse sellest järjendist miinimum väärtus meetodiga *minBackEdgeDfn()*.

Tipu *low* väärtuse leidmiseks on vaja veel teada tipu DFS puu laste madalaimat *low* väärtust. Teisisõnu tipu enda *low* väärtuse teada saamiseks on vaja teada tema laste *low* väärtuseid. Seetõttu on vaja *low* väärtuseid hakata leidma DFS puu lehtedest, millel endal lapsi ei ole (laste puudumisel saab saab tipu *low-link* väärtuse teada ilma selleta). Saades teada lehtede *low* väärtused, saab hakata mööda puud üles liikuma, et ka teised väärtused üles leida. Tipu *low* väärtuse võib leida alles, siis, kui kõik temast lähtuvad kaared on ära töödeldud. Leides tipu *low* väärtust meetodiga *low()*, kutsutakse omakorda välja meetod *dfsChildLowValue()*, mis tagastab tipu laste madalaima *low* väärtuse.

Tabel 1 on esitatud *low-link* väärtuste leidmiseks sobivas järjekord. Järjekord tuleneb graafi sügavuti otsingu rekursioonist väljumise järjekorrast. Tulp *id* näitab, mis tipu väärtuseid antud tabeli rida sisaldab. Tabelis on välja toodud iga tipu DFS puu lapsed ja *backedge* kaared ja neile vastavad *low-link* väärtused ja kaare sihtpunkti avastamise numbrid. Miinimum väärtus tipu enda järjenumbri (*discovery*), tema *backedge*'ide sihtpunktide järjenumbrite (*backEdgesDfns*) ja tema DFS puu laste eelnevalt leitud *low-link* väärtuste (*childrenLowValues*) seast on tipu enda *low-link* value ja on kirjutatud *Low* tulpa. DFS puu laste *low-link* väärtused tulenevad tabeli eelmistelt ridadelt, sest DFS puu laste *low-link* väärtused tuleb leida enne, kui saab leida tipu enda *low-link* väärtust. Tabeli lahtrid, milles väärtused puuduvad on täidetud halliga, et neid ridu paremini rõhutada, sest selliste ridade puhul sõltub *low-link* väärtuse leidmine vähematest väärtustest.

Tabel 1. Tippude *low* väärtuste leidmine.

			Low = min(discovery, backEdgesDfns, childrenLowValues)				
jrk	id	Low	discovery	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 Artikulatsioonipunktide leidmine

Graafi juurtipp on artikulatsioonipunkt siis ja ainult siis, kui DFS puus on tal mitu last. Seda seetõttu, et alustades otsingut ühest lapsest ei ole võimalik teise lapseni jõuda mitte mingit teed pidi, välja arvatud läbi juurtipu enda. Kui juurtipul on üks laps, siis ta ei ole artikulatsioonipunkt.

Lisaks ei ole artikulatsioonipunktideks need tipud, millele ei leitud ühtegi last sügavuti otsingu käigus ehk tipud, mis on DFS puu lehed. Nende eemaldamise tagajärjel tippude arv graafis väheneb, kuid graafi sidusus nendest ei sõltu.

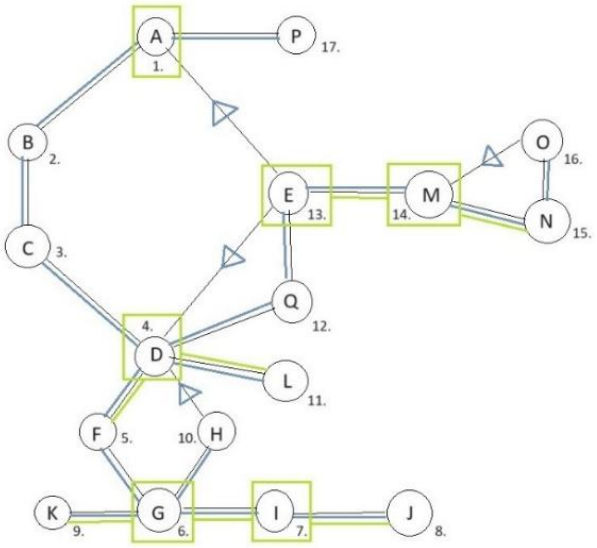
Graafi tipp on artikulatsioonipunkt, kui tipu enda *discovery* väärtus on väiksem või võrdne kasvõi ühe tema DFS puu lapse *low* väärtusega. Selle tipu ära võtmine ohustab neid lapsi, kelle puhul antud tingimus tõene oli ehk AP eemaldamisel ei ole enam võimalik jõuda alguspunktist nende lasteni.

Tabel 1 järjekord põhines graafi DFS rekursioonist väljumise järjekorral ning on kasutatud ka siin (vt Tabel 2). Tabel 2 on kirjutatud välja iga tipu *discovery* number ning tema lapsed ja nende *low* väärtused, neid omavahel võrreldud ning tulemuste põhjal tehtud järeldused. Kui tipu vähemalt ühel lapsel on madalam või võrdne *low-link* value kui tipu enda *discovery* väärtus, siis on tipp artikulatsiooni punktiks. Lehed ehk tipud, millel ei ole DFS puus lapsi ei ole artikulatsiooni punktid ja juurtipp on artikulatsiooni punktiks ainult siis, kui tal on DFS puus rohkem kui üks laps.

Tabel 2. Artikulatsioonipunktide leidmine

jrk	id	discovery		child.low	children	artikulatsioonipunkt?		põhjendus
1	J	8				väär		J on leht
2	I	7	≤	J.low = 8	J	tõene	AP	
3	K	9				väär		K on leht
4	H	10				väär		H on leht
5	G	6	≤	I.low = 7 K.low = 9 H.low = 4	I K H	tõene tõene väär	AP	
6	F	5	≤	G.low = 4	G	väär		
7	L	11				väär		L on leht
8	O	16				väär		O on leht
9	N	15	≤	O.low = 14	O	väär		
10	M	14	≤	N.low = 14	N	tõene	AP	
11	E	13	≤	M.low = 14	M	tõene	AP	
12	Q	12	≤	E.low = 1	E	väär		
13	D	4	≤	F.low = 4 L.low = 11 Q.low = 1	F L Q	tõene tõene väär	AP	
14	C	3	≤	D.low = 1	D	väär		
15	B	2	≤	C.low = 1	C	väär		
16	P	17				väär		P on leht
17	A	1			B P	tõene	AP	A on juur, millel on rohkema kui üks laps

Eelnevalt mainitud tingimuse põhjal, et tipp on artikulatsioonipunkt, siis kui kasvõi ühe tema laste puhul kehtib võrratus *tipu enda discovery number* ≤ *lapse low väärtus* on Tabel 2 tulemuste põhjal on graafi artikulatsioonipunktideks tipud D, E, G, I, M. Graafi juurtipp A omab DFS puus kahte last, mistõttu on ka juurtipp AP. Joonis 5 on esiletõstetud kõik artikulatsiooni punktid – ümbritsetud rohelise kastiga – ja APst teed kõigi lasteni, kelle puhul tingimus tõele vastas – servaga paralleelse rohelise joonega – ehk mis lapsed on DFS puus ohustatud punkti eemaldamisest.



Joonis 5. Näite graaf koos artikulatsioonipunktidega.

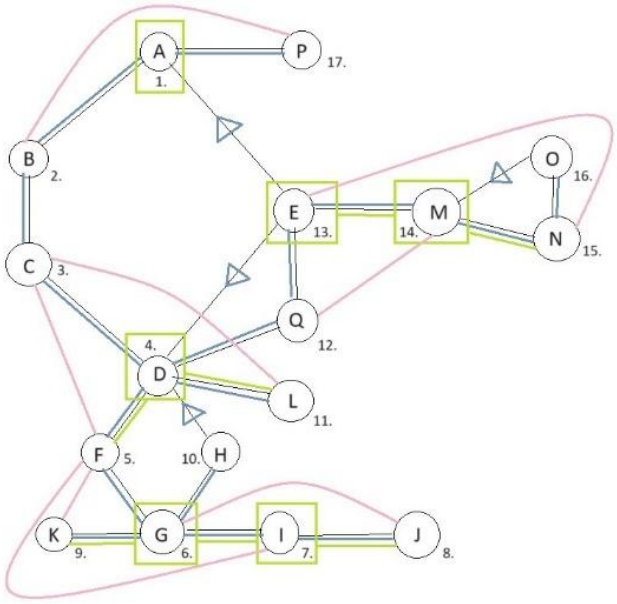
2.1.4 Servade lisamine, et graafis enam artikulatsioonipunkte ei leiduks

Graafile, mis koosneb *n* tipust võib lisada maksimaalselt *n* serva, et artikulatsioonipunktidest graafis vabaneda. Sügavuti otsingus töödeldakse tippu ainult ühe korra ja ühel tipul võib olla ainult üks vanem, järelikult lisades iga tipu jaoks ühe uue serva saabki maksimaalselt lisada kuni n serva.

Juhul, kui juurtipp osutub artikulatsioonipunktiks, siis võib lisada serva iga lapse vahele. Näiteks, kui juurtipul on rohkem kui kolm DFS last, siis lisatakse serv esimese ja teise lapse vahele, teisest kolmanda lapseni jne.

Teiste artikulatsioonipunktide puhul, on oluline, et oleks teada, mis tipp on punkti vanem DFS puus. Uued servad lisatakse AP vanemast AP selliste DFS lasteni, kelle puhul AP enda järjenumbri ja tema lapse *low* väärtuse enne mainitud tingimus vastab tõele.

Antud ülesande puhul eelnevalt rohelisega märgistatud servade otspunkti, mis ei ole AP ise, ja artikulatsioonipunkti DFS puu vanema vahele peab lisanduma uues serv (vt Joonis 6). Samuti on lisatud serv ühendamaks juurtipu lapsi. Kokku lisati graafile 8 uut serva.



Joonis 6. Näite graaf koos lisatud servadega.

On oluline mainida, et graafi artikulatsioonipunktid on alati üheselt määratud, aga lisatavad servad nende elimineerimiseks ei ole. Antud lahendus elimineerib graafist artikulatsioonipunktid, aga ei defineeri, mis on parim komplekt servi, mida graafile lisada.

3 Programmi kasutusjuhend

Ülesande lahendamist juhib klassi *Graph* meetod *biConnectGraph()*. Meetod tagastab artikulatsioonipunktidest koosneva hulga. Meetod tagastab AP hulga, sest olenemata graafi läbimise järjekorrast jääb see hulk alati samaks. Meetodi käigus lisatavad kaared võivad erineda, olenevalt, mis järjekorras kaared lähtekoodis tipust väljuvad.

Programmi kasutamiseks on esmalt vaja luua uus graafi instants. Seejärel tuleb luua graafi sisestruktuur. Selleks võib kasutada graafi meetodit *createRandomSimpleGraph(int n, int m)*, mis loob suvalise lihtgraafi, kus on *n* tippu ja *m* serva. See on sobilik kasutamiseks ka suurte graafide loomiseks. Lisaks võib kasutada graafi meetodit *createGraphFromStructure(LinkedHashMap<String, List<String>> structure)*, millele tuleb ette anda *LinkedHashMap* klassi objekt, kus iga elemendi *key* on tipu nimi ja *value* on järjend tipu nimedest, kuhu peaksid kaared, mis väljuvad *key* nimega tipust, suubuma.

Graafi olemasolul võib kutsuda meetodit *biConnectGraph()* loodud *Graph* klassi instantsile. Meetod leiab üles graafi kõik artikulatsioonipunktid ja lisab graafile juurde servasid, et ühegi üksiku tipu eemeldamisel ei oleks graafi sidusus ohus. Meetod tagastab hulga artikulatsioonipunktidest ehk *Vertex* klassi objektidest. Parima võrdlusmomendi jaoks on soovituslik kuvada graafi struktuur konsooli kaudu enne ja pärast meetodi rakendamist, et näha, kas, kui palju ja kuhu servasid juurde lisati.

Kui soovitakse kohe näha infot algse graafi artikulatsioonipunktide kohta, kui palju selliseid punkte graafis oli, mis punktid need olid ja milliste servade lisamisel graafis artikulatsioonipunktid kaovad, siis võib kasutada meetodit *biConnectGraph(boolean print, boolean detailed)*. Meetodi parameetriteks on kaks tõeväärtus. Seades esimeseks tõeväärtuseks *print* väärtuse *false* ehk väär on meetodi väljakutsumine samaväärne ilma ühegi parameetrita *biConnectGraph()* meetodi väljakutsega. Seades tõeväärtuste paarist ainult esimese väärtuse *print* väärtuseks *true* ehk tõene kuvatakse programmi töökäigu lõpus välja, mitu artikulatsioonipunkti graafis leiti. Seda on kasulik kasutada suurte graafide puhul, millel võib olla palju artikulatsioonipunkte ning nende kõikide ekraanile kuvamine ei anna kasutajale väärtuslikku infot juurde. Seades mõlemad väärtused tõeseks

kuvatakse ekraanile peale leitud artikulatsioonipunktide arvu ka nende punktide nimed ning kõikide kaarte nimed, mis graafile juurde lisati.

Juurde on loodud ka abimeetod *printGraphBeforeAndAfter(String graphName, LinkedHashMap<String, List<String>> graphStructure)*. See meetod loob antud nime ja struktuuriga graafi, kuvab graafi struktuuri enne *biConnectGraph()* meetodi rakendamist ja pärast meetodi poolt servade lisamist.

Lisaks on kirjutatud ka abimeetod *bigGraphTimeMeasuring(String title, String graphName, int i, int m)*. Meetodile on vaja anda täpsustav pealkiri *title*, millega anda infot selle kohta, mille aega hakatakse mõõtma. Lisaks peab ka sellele abimeetodile andme nime, mida graafi loomisel kasutada *graphName*. Viimaks peab ette andma, mitme tipu *n* ja mitme servaga *m* graafi soovitakse. Meetod loob suvalise graafi, millel on *n* tippu ja *m* serva. Seejärel käivitab meetod omakorda *biConnectGraph()* meetodi ning mõõdab, kui palju aega kulub sellel meetodil, et programm oma töö lõpetaks. Lõpuks kuvatakse ekraanile, mitu AP-d leiti ja mitu millisekundit protsess aega võttis.

4 Testimiskava

Programmi testimisel tuleks kontrollida artikulatsioonipunktide leidmist äärmuslikel juhtudel, kus graafil on nii minimaalne kui ka maksimaalne lubatud servade arv. Lisaks peaks testima olukordi, kus graafis ei ole artikulatsiooni punkte, et teha kindlaks, et meetod ei leia APsid, kus neid ei ole, ning olukordi, kus graaf sisaldab APsid, et verifitseerida meetodi võimekust neid leida, regulaarseid graafi struktuure, kuid ka keerulisemaid graafe, et meetod tuleks ka nende puhul toime DFSi ja arvutustega.

Järgnevates näidetes on juurtipp igal graafil tipp A.

4.1 Väike graaf ilma artikulatsioonipunktideta

Artikulatsioonipunkt saab graafil olla alates kolmest tipust. Graafil, millel on vähem tippe, on kõik tipud kas lehed või juurtipud, millel on vähem kui kaks last, millest kumbki ei saa artikulatsioonipunktideks olla.

Näiteks ühe tipuga graafil (vt **Error! Reference source not found.**) ei saa artikulatsioonipunkte olla, sest eemaldades selle punkti ei jää alles ühtegi, mis omakorda tähendab, et ei ole võimalik ka graafi mitmeks osaks jagada.

Joonis 7. Graaf ühe tipuga.

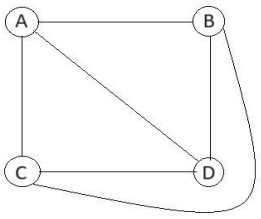
Kahe tipuga graafil ei ole samuti AP-d võimalikud (vt Joonis 8). Ühe otspunkti eemaldamisel jääb alles ainult teine, mis taaskord tähendab, et tipu eemaldamise tulemusena ei saa alles jääda rohkem sidususkompenente kui üks.

Joonis 8. Graaf kahe tipuga.

Nii Joonis 7 kui ka Joonis 8 kujutatud graafide väljatrükid on olemas „Testi näidete tulemused“ tabelis (vt Lisa 1), *run()* meetodis on graafide loomise lähtekood vastavalt nimede all *“Small Graph with one vertex”* ning *“Small Graph with two vertices”*, ning mitte ühegi artikulatsioonipunkti leidmise tulemuse testimiseks on test *biConnectVerySmallGraph()*.

4.2 Täisgraaf

Testida tuleks ka olukorda, kus graaf osutub täisgraafiks, mille puhul väljub igast tipust kaar igasse teise tippu graafis (vt Joonis 9). Sellises graafis ei saa olla mitte ühtegi artikulatsioonipunkti, sest kõik kaare kombinatsioonid on graafis juba olemas.

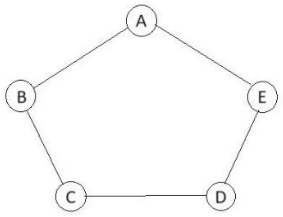


Joonis 9. Täisgraaf.

Täisgraafi loomine on toodud ära ka *run()* meetodis nimega *"Full Graph"* ja selle konsooli väljund enne ja pärast meetodi rakendamist on näha tabelis „Testi näidete tulemused" (vt Lisa 1). Lisaks kontrollib test *biConnectFullGraph()* meetodi toimimist täisgraafi peal, et leitud APde hulk oleks tühi.

4.3 Lihttsükkel

Samuti võiks testida algoritmi graafi peal, mis kujutab endast lihttsüklit. Lihttsükkel on kinnine lihtahel, mille läbimisel tipud ei kordu ja mis algab ning lõpeb samas punkti (vt Joonis 10). Sellises graafis ei saa olla mitte ühtegi artikulatsioonipunkti, sest ükskõik millise tipu eemaldamisel on võimalik igasse teise tippu jõuda mööda alles jäänud lihtahelat.

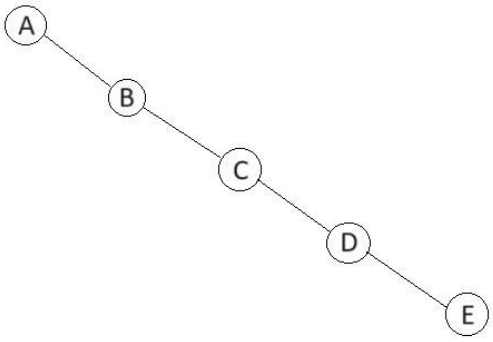


Joonis 10. Lihttsükkel.

Ka lihttsükli loomine on toodud ära ka *run()* meetodis, antud juhul nimega *"Simple Cycle Graph"* ning graafi struktuuri väljatrükk nii enne kui ka pärast *biConnectGraph()* meetodi rakendamist on toodud tabelis „Testi näidete tulemused" (vt Lisa 1). Lisaks on loodud ka test *biConnectCycleGraph()* kontrollimaks, et meetod ei leidnud lihttsüklist mitte ühtegi artikulatsioonipunkti.

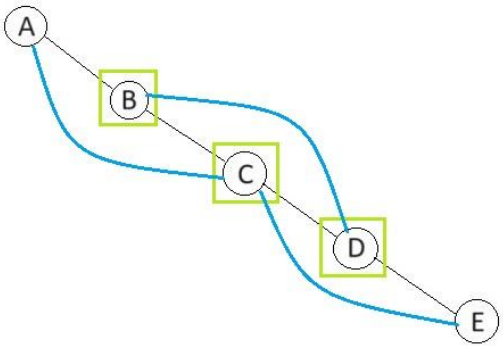
4.4 Lihtahel

Algoritmi on vaja testida ka graafide peal, milles leidub artikulatsioonipunkte. Lihtsamaks näiteks on lihtahel ehk ahel, mille läbimisel tipud ei kordu (vt Joonis 11). Ükskõik, millise punkti, mis ei ole otsmine punkt, eemaldamisel jaotub lihtahel kaheks graafiks, mis ei ole omavahel enam seotud – esimesse graafi kuuluvad kõik tipud, mis on enne ära võetud punkti ja teise graafi kõik tipud, mis on pärast eemaldatud punkti.



Joonis 11. Lihtahel.

Lihtahela graafis on artikulatsioonipunktideks kõik tipud, peale graafi juure ning ahela viimase lehe (vt Joonis 12). Graafi sidususe kadumise ohu vältimiseks lisatakse servad kõikidele AP-dele eelnevast tipust AP-dele järgnevasse tippu. Sellisel juhul ei sõltu graafi sidusus enam üksikutest tippudest.

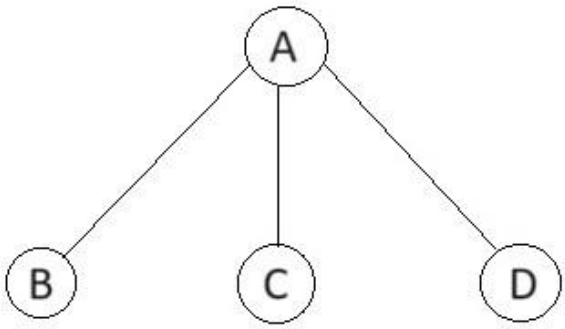


Joonis 12. Lihtahel koos lisatud servadega.

Lihtahela graafi, nimega *"Simple Chain Graph"*, loomise lähtekood on *run()* meetodis ning stsenaariumit on testitud kasutades *biConnectChainGraph()*. Graafi algne struktuur ja meetodi *biConnectGraph()* töö käigus leitud artikulatsioonipunktid ning juurde lisatud servad on tabelis „Testi näidete tulemused" nimega *"Simple Chain Graph"* (vt Lisa 1).

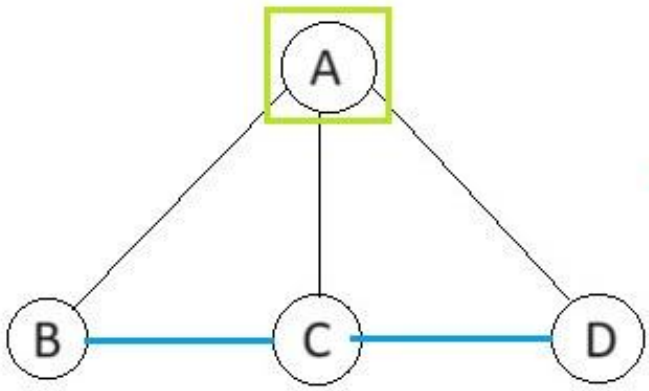
4.5 Graafi juurtipp on artikulatsioonipunkt

Graafi juurtipu puhul on artikulatsioonipunkti tingimus erinev, kui teistel graafi tippudel. Juurtipu puhul on ainsaks tingimuseks see, et DFS puus leidub juurtipul vähemalt kaks otsest järglast, sest mingit muud teed pidi, peale juurtipu läbimise, neid külastada ei õnnestunu (vt Joonis 13).



Joonis 13. Graaf, kus juurtipp on artikulatsioonipunktiks.

Juhul, kui AP-ks osutub muu hulgas ka juurtipp, siis tuleb juurtipu lapsed omavahel ühendada (vt Joonis 14). Eemaldades Joonis 14 näidatud graafilt juurtipu A, jääb alles lihtahel ehk kõik tipud on omavahel ühendatud.

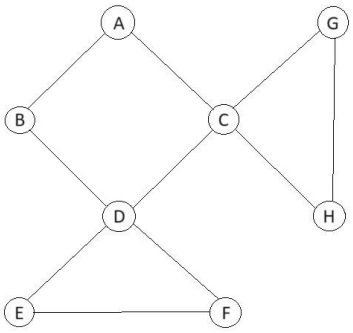


Joonis 14. Juurtipust artikulatsioonipunktiga graaf koos lisatud servadega.

Joonisel 13 kujutatud graafi loomist on näha klassi *GraphBiConnector* meetodis *run()*, kus graafi nimi on *"Root AP"* ning graafi väljatrükki enne ja pärast *biConnectGraph()* meetodi rakendamist on toodud tabelis „Testi näidete tulemused" (vt Lisa 1). Olukorra jaoks on kirjutatud ka test *biConnectRootAp()*, mis kontrollib, kas antud graafis leiti ainult üks artikulatsioonipunkt ning kas selleks punktiks on juurtipp A.

4.6 Artikulatsioonipunktid keerulisemates graafides

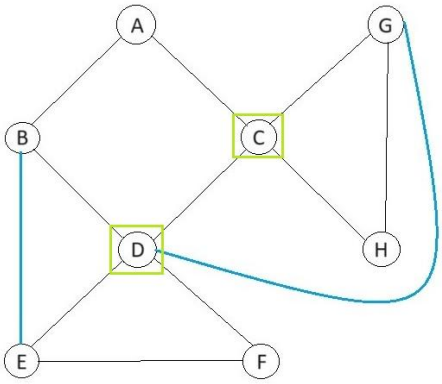
Loomulikult tuleb meetodit testida ka ebaregulaarsete graafide peal, et veenduda, et eelnevad tulemused ei olnud õiged juhuslikult. Näiteks võib testida graafi, mida on kujutatud Joonis 15. Joonis 15 graafil on kaks nii-öelda pudelikaela, mida tuleb graafi läbimisel külastada, et teiste punktideni jõuda. Need pudelikaelad, tipud C ja D, ongi graafi APdeks. Eemaldades tipu C eralduvad ülejäänud graafist tipud G ja H ning tipu D eemaldamisel, ei ole võimalik jõuda enam punktideni E ega F. Antud graaf on sobilik ka seetõttu, et juurtipul on graafil ühendus mitme lapsega, kuid mõlemasse neist on võimalik jõuda ka teist teed pidi. DFS puus on juurtipul ainult üks järglane, mistõttu see ei ole AP.



Joonis 15. Tsüklitega näite graaf.

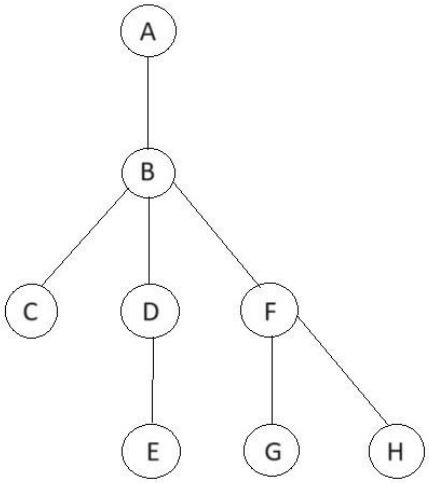
Antud graafi loomise lähtekood on välja toodud meetodis *run()*, kus on graaf nimega *"Complex Graph With Cycles"* ning graafi struktuuri ja artikulatsioonipunktide väljatrükki saab näha ka tabelis „Testi näidete tulemused" (vt Lisa 1). Test meetod *biConnectComplexGraphWithCycles()* kontrollib, et õiged punktid tuvastatakse APdena.

Rakendades *biConnectGraph()* meetodit antud graafile lisatakse graafile servasid juurde, et pudelikaelte oht eemaldada (vt Joonis 16). Sinisega on tähistatud juurde lisatud servad ning rohelisega leitud artikulatsiooni punktid.



Joonis 16. Tsüklitega näite graaf pärast servade lisamist.

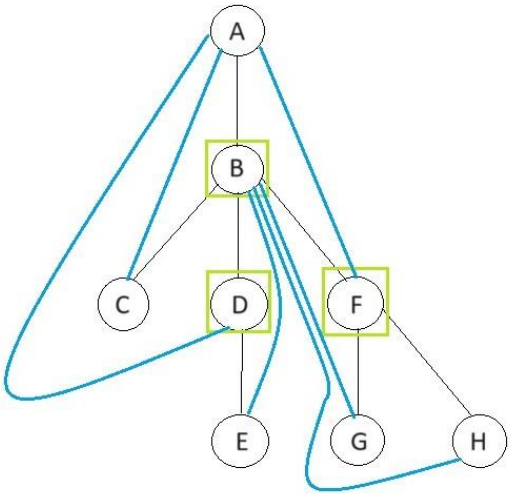
Eelmises graafi näites on graafis sees mitu tsüklit ehk DFS käigus kõiki kaari läbi ei käida. Näiteks alustades punktist A ja liikudes edasi punkti B kaudu, siis lõpuks tippu C jõudes ei ole vaja A juurde enam minna, sest seda juba külastati. Algses graafis on seega rohkem servasid, kui DFS puus. Sellele vastanduvaks näiteks võiks võtta graafi, kus ei ole ühtegi tsüklit ning seega ei ole ühtegi serva, mida sügavuti läbimine läbi ei läbiks (vt Joonis 17).



Joonis 17. Tsükliteta näite graaf.

Antud graafis leiduvad artikulatsioonipunktid on tõstetud esile rohelisega ning meetodi *biConnectGraph()* töö tulemusena lisatud juurde servad on joonisel sinised (vt Joonis 18).

Antud graafi loomise lähtekood on välja toodud meetodis *run()*, nimega *"Complex Graph Without Cycles"* ja graafi struktuuri väljatrükk on ka tabelis „Testi näidete tulemused" (vt Lisa 1). Test meetodiga *biConnectComplexGraphWithoutCycles()* verifitseerib *biConnectGraph()* meetodi töötulemusi.



Joonis 18. Tsükliteta graaf koos lisatud servadega.

4.7 Keeruline graaf

Näite graaf Joonis 3 on samuti kaasatud testimiskavasse. Test meetod *biConnectComplexGraph()* kontrollib, et meetodi *biConnectGraph()* töö kaigus saadud tulemus vastaks peatüki 2.1 lahenduse käigus saadud tulemusele. Antud näite graaf on kasulik, sest sisaldab endas mitmeid erinevaid eelnevalt mainitud elemente.

Graafi loomise lähtekood on *GraphBiConnector* klassi meetodis *run()* ning konsooli väljatrükid on olemas tabelis „Testi näidete tulemused“ nime all *„Complex graph“* (vt Lisa 1).

Kasutatud kirjandus

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

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

Lisa 1 – Testi näidete tulemused

```
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
-------------------------------------------------------------------------------------------
```