Project Algoritmen en Datastructuren II

Jasper Van der Jeugt

30 november 2009

Inhoudsopgave

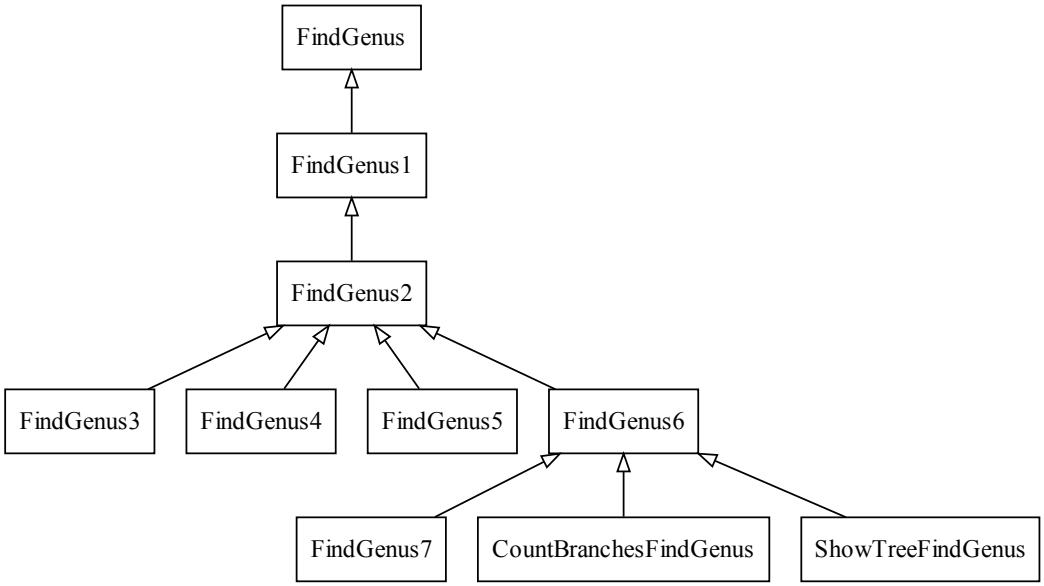
1 De klassestructuur

1.1 Verschillende versies van het algoritme

Om tijdsmetingen te kunnen doen, is het leuk als we verschillende versies van ons algoritme kunnen vergelijken. Natuurlijk is het ook niet de bedoeling dat we telkens voor een nieuwe versie onze volledige code kopiëren, want dan zouden we snel een zeer moeilijk te onderhouden programma hebben. Daarom kiezen we ervoor om met overerving te werken.

De overervingsstructuur voor de verschillende versies van het algoritme is te zien in Figuur 1. Elk algoritme dat een eerder algoritme uitbreidt, erft natuurlijk ook alle eigenschappen van het eerdere algoritme. Zo zullen alle klassen die subklasses zijn van `FindGenus2` de graaf eerst preprocessen.

- `FindGenus` is de te implementeren interface.
- `FindGenus1` implementeert de naïeve versie van ons algoritme zoals beschreven in 6. Het gebruikt de klasse `DefaultGraph`. Let wel dat de verbetering uit 6.4 ook al is geïmplementeerd. Ook maakt het gebruik van de heuristiek uit 9.2.
- `FindGenus2` gaat de graaf eerst preprocessen. Daartoe gebruikt het de `GraphProcessor` interface en de `SimplifyGraphProcessor` implementatie ervan. Welke preprocessing allemaal wordt toegepast, is verder beschreven in 8.
- `FindGenus3`, `FindGenus4`, `FindGenus5` en `FindGenus6` voegen telkens de bounding criteria toe uit respecievelijk 7.2, 7.3, 7.4 en 7.5, waarbij elke versie ook de eigenschappen van zijn voorganger gebruikt. `FindGenus6` is tevens onze beste versie.
- `FindGenus7` implementeert ook heuristiek 9.1.
- `CountBranchesFindGenus` is een klasse die na afloop van het algoritme het aantal afgewerkte branches afdrukt.
- `ShowTreeFindGenus` is een klasse die de call tree van het algoritme afdrukt in *DOT*-formaat.



Figuur 1: Een overervingsdiagram die de verschillende versies van het algoritme toont

1.2 Overige klassen

- `FindGirth` is een klasse die het *girth* van een graaf kan bepalen. Dit wordt gebruikt in 7.3.
- `GraphCloner` is een klasse die een perfecte kopie kan maken van een graaf. Dit wordt gebruikt om een backup te nemen van een graaf die we zullen veranderen door preprocessing.
- `DefaultGraph`, `Vertex` en `Order` zijn klasses die gebruikt worden om de graaf efficiënt voor te stellen tijdens het algoritme.
- `SortedGraph` is een uitbreiding van `DefaultGraph`, gebruikt door de klasse `SortedFindGenus`.

2 Tests

2.1 Performantietests

Om de performantie van ons algoritme te testen zullen we de klasse `TimeTest` gebruiken. Dit is een simpele klasse die op de commandolijn als argumenten een interval voor het aantal toppen, en een aantal klasssenamen neemt. De klasses moeten instanties zijn van `FindGenus`. `TimeTest` zal een aantal keer een willekeurige graaf aanmaken, en dan telkens de verschillende `FindGenus` klassen het genus laten bepalen. Belangrijk is dat elk algoritme de graaf kan veranderen door preprocessing. Daarom maken we ook telkens een clone van de graaf,

met behulp van de klasse `GraphCloner`. Het aantal bogen wordt telkens willeukerig bepaald en ligt in het interval $[v-1, \frac{(v-1)\cdot v}{2}]$.

De resultaten worden uitgeschreven naar standaard-uitvoer. We kunnen deze dan later analyseren en plotten.

Er is ook een klasse `GirthTimeTest`. Dit is een zeer analoge klasse, die de performantie van `FindGirth` kan testen.

2.2 Correctheidstests

Alle correctheidstests bevinden zich in de map `tests/unit` en zijn subklasses van `UnitTest`. Op deze manier is het makkelijk om de testen te controleren als er veranderingen in ons algoritme zijn. Specifieke tests worden in dit verslag besproken waar ze meest relevant zijn. De algemene tests zijn:

- `CompleteGraphGenusTest`: Test het zoeken van het genus van een complete graaf. Dit kunnen we eenvoudig bepalen, zie 4.2.
- `CompleteBipartiteGraphGenusTest`: Test het zoeken van het genus van een complete bipartiete graaf. Dit kunnen we ook eenvoudig bepalen, zie 4.3.
- `RandomGraphGenusTest`: Test het zoeken van het genus van een willeukeurige graaf met twee algoritmes. Het resultaat moet natuurlijk hetzelfde zijn. Dit is uiteraard een zeer sterke test.

3 Implementatie van Graph

Een deel van het practicum bestond uit het schrijven van een klasse die `Graph` implementeerd. Deze implementatie bevindt zich in de klasse `BasicGraph`. We gebruiken deze klasse *niet* in ons eigenlijk algoritme, omdat we dan zeer specifieke informatie willen opvragen op een efficiënte manier. In ons algoritme gebruiken we de klasse `DefaultGraph`. Het spreekt voor zich dat een `Graph` kan worden omgezet in een `DefaultGraph`. Omgekeerd is de omzetting niet nodig, en dit is dus ook niet geïmplementeerd.

3.1 BasicGraph

We willen hier per vertex de buren bijhouden. Vertices worden volledig voorgesteld door integers. Omdat we niet weten of de lijst integers die de vertices voorstellen een lijst van de vorm $0, 1, 2, \ldots n$ zal zijn, kunnen we ze niet opslaan in een array. Daarom kiezen we voor een `HashMap`. We zouden de buren van een bepaalde top kunnen opslaan als een `Set`, maar aangezien in de interface `Graph` staat dat we bij het opvragen van de buren (`getNeighbours`) een `List` moeten teruggeven, kiezen we voor een eenvoudige `ArrayList`. De graaf wordt dus voorgesteld door een datastructuur van de vorm

```
HashMap<Integer, ArrayList<Integer>>
```

Om deze implementatie te testen schreven we een simpele correctheidstest, `SimpleGraphBuildTest`. Deze voegt enkele bogen toe en verwijdert enkele bogen, en kijkt het resultaat van deze operaties na.

3.2 DefaultGraph

Deze klasse implementeerd `Graph` niet, maar wordt gebruikt bij het uitvoeren van het algoritme. We maken een `DefaultGraph` altijd via een `Graph`. We willen informatie over de toppen liefst zo efficiënt mogelijk opslaan. Moest de lijst vertices van de vorm $0, 1, 2, \ldots n$ zijn, zouden we een efficiënte array kunnen gebruiken. Dit is niet het geval, maar we lossen dit op door elke vertex een ander nummer te geven, zodat we wel een array kunnen gebruiken.

Informatie over de bogen willen we ook zeer snel kunnen raadplegen. Daarom kiezen we ervoor iets meer geheugen te gebruiken, en gebruik te maken van een soort *adjacentiematrix*.

De informatie over een top zit opgeslagen in de klasse `Vertex`. Deze bevat in feite niet veel meer dan een array met de buren van deze top.

Informatie over de bogen zit opgeslaan in de klasse `Order`. Deze klasse stelt een soort volgorde voor, die we zullen gebruiken in ons algoritme. We willen immers per top een zekere volgorde van bogen vastleggen. Een volgorde wordt klassiek voorgesteld door een lijst. We kiezen ervoor om een soort linked list te gebruiken. Elk object van de klasse `Order` stelt een element uit de lijst voor, maar kan tegelijk ook gebruikt worden als lijst. Hierdoor kunnen we soms sneller informatie opvragen.

4 Input: verschillende grafen

Als input neemt het algoritme telkens een graaf. Daar er enorm veel verschillende grafen bestaan, beschouwen we eerst verschillende manieren om een graaf aan te maken, die we dan in de tests kunnen gebruiken. De verschillende klasses die hierbij horen zitten in `tests/graph`, in het java package `graph`. Ze zijn allemaal subklasses van de klasse `BasicGraph`, die elk een specifieke constructor hebben.

4.1 ZGraph

Op `http://zeus.ugent.be/zgraph`, een project gestart door enkele studenten (Robrecht, Pieter en mijzelf) staan enkele voorbeeldgrafen. Om deze in te laden is het bestandsformaat geïmplementeerd in de klasse `ZGraph`. De constructor van deze klasse neemt een bestandsnaam, en laad deze graaf.

4.2 CompleteGraph

Een specifieke subklasse van de grafen zijn de complete grafen. Deze zijn zeer makkelijk te genereren. Dit is geïmplementeerd in de klasse `CompleteGraph`. De constructor van deze klasse neemt een getal n en maakt vervolgens de graaf K_n aan. Deze grafen kunnen we ook zeer goed gebruiken voor correctheidstests, aangezien we weten dat

$$g_{min}(K_n) = \lceil \frac{(n-3)(n-4)}{12} \rceil$$

4.3 CompleteBipartiteGraph

Naast de complete grafen zijn ook de bipartiete complete grafen makkelijk te genereren en te testen. De constructor van de klasse `CompleteBipartiteGraph` neemt als argumenten n, m en maakt dan de graaf $K_{n,m}$ aan. Ook voor deze grafen kunnen we het genus op voorhand bepalen met de formule

$$g_{min}(K_{n,m}) = \lceil \frac{(n-2)(m-2)}{4} \rceil$$

4.4 RandomGraph

Voor het testen van de performantie is veel data nodig - en dus veel input. Het zou daarom handig zijn als we willeurig grafen konden genereren met v toppen en e bogen. We weten dat $e \geq v - 1$, dit is nodig als we een samenhangende graaf willen construeren. De klasse `RandomGraph` maakt willekeurig grafen aan met het volgende algoritme:

Neem v toppen, zonder bogen. We hebben nu een onsamenhangende graaf die bestaat uit v componenten (Zie Figuur 2).

Nu gaan we in deze graaf een opspannende boom construeren. Hiervoor hebben $v - 1$ bogen nodig. Als we deze boom eenmaal hebben, hebben we zeker een samenhangende graaf.

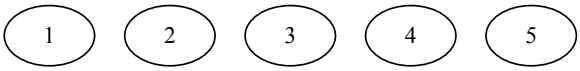
Zolang de graaf niet samenhangend is, voegen we componenten samen op de volgende manier:

Neem twee loshangende componenten c_1 en c_2 uit de graaf. Neem in c_1 een willekeurige top v_1 en in c_2 een willekeurige top v_2. Verbind nu v_1 met v_2. Er is nu één component minder in de graaf. We gaan zo door tot we een opspannende boom verkregen hebben, bijvoorbeeld deze die te zien is in Figuur 3.

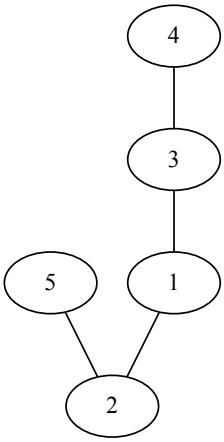
We hebben nu $v - 1$ bogen toegevoegd. We moeten dus nog $e - v + 1$ bogen toevoegen. Stel E_v alle bogen in de complete graaf met v toppen, en T de bogen in onze opspannende boom. Neem nu willekeurig $e - v + 1$ bogen uit $E_v \setminus T$ en voeg deze toe aan onze graaf. We hebben nu een relatief willeukeurige graaf met e bogen en v toppen (Voorbeeld: Figuur 4).

5 Output: Graphviz

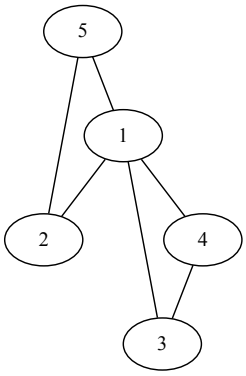
Het is handig als we grafen kunnen visualiseren, om de correctheid van bepaalde delen van ons algoritme te testen. Daarom besliste ik een klasse `GraphDotWriter` te schrijven, deze bevindt zich in het package `writers` (in de map `tests`). Deze klasse bevat functionaliteit om grafen van de klasse `Graph` uit te schrijven naar DOT-bestanden. De specificatie van deze dot-bestanden is te vinden op `www.graphviz.org`. Omdat we enkel simpele grafen willen outputten, is deze klasse niet zo complex.



Figuur 2: RandomGraph, stap 1



Figuur 3: RandomGraph, stap 2



Figuur 4: RandomGraph, stap 3

6 Het naïeve algoritme

6.1 Lazy genereren van embeddingen

Ons naïeve algoritme overloopt alle embeddingen. Het is echter belangrijk dat we de embeddingen op een *lazy* manier genereren. Indien we de permutaties *strict* zouden genereren, zouden we een algoritme krijgen dat (weliswaar met branching) het volgende idee implementeerd:

```
genus = +Infinity
for e in getEmbeddings(graph):
    eGenus = getGenus(e)
    if eGenus < genus:
        genus = eGenus
```

Hierbij is het probleem dat we pas informatie over het genus krijgen op het moment dat e gegenereerd is. Als we hierop bounding criteria willen toepassen, zouden we enkel de bladeren in onze zoekboom kunnen schrappen. Omdat we meer willen schrappen, zoeken we dus naar een beter algoritme, waarin we sneller informatie over het genus verkrijgen.

6.2 findGenus en findFaces

Het genus van een graaf en een embedding i wordt gegeven door $v + f_i - e = 2 - 2g_i$. We weten v en e vast liggen, en enkel f variëert als voor een graaf de embeddingen aflopen. Het zoeken van $min(g)$ komt dus neer op het zoeken van $max(f)$, want

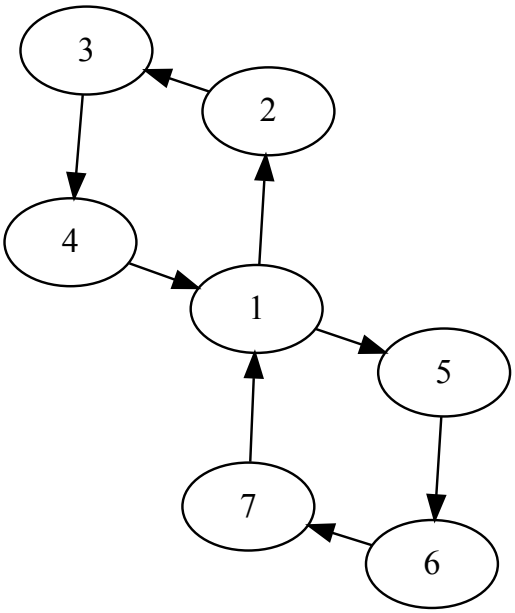
$$min(g) = 1 - \frac{v + max(f) - e}{2}$$

6.3 Algoritme: Het zoeken van het maximaal aantal vlakken

Eerst zetten we de graaf die we krijgen als input om in een gerichte graaf. We beginnen met een willekeurige (gerichte) boog e_1 die nog niet in een pad ligt. Stel dat op het einde van deze boog de top v ligt. We nemen nu de verzameling kandidaat bogen E_v die op e_i kunnen volgen. Deze stap is niet triviaal en wordt verder beschreven in 6.5. Nu moeten we branchen voor elk element van E_v. We voegen de gekozen boog e_{i+1} toe aan ons pad. Indien deze boog dezelfde boog is als de eerste boog in ons pad, e_1, Hebben we van ons pad een cykel gemaakt. Op dat moment kijken we of er nog bogen zijn die niet in een pad liggen. Als dit het geval is, beginnen we een nieuw pad met een willeukerige boog. Anders hebben we een volledige embedding gevonden, en bevinden we ons in een blad van de zoekboom.

6.4 Het pad sneller sluiten

In de vorige paragraaf schreven we dat we als we op een gegeven moment een boog toevoegen die dezelfde is als de eerste boog in ons pad, dat we dan een cykel maken. Dit is natuurlijk waar, maar we kunnen zelfs nog sneller stoppen met ons pas, namelijk *op het moment dat we in de eerste top van het pad terugkomen, en we kunnen het pad sluiten zonder dat dit de permutaties verstoord.*



Figuur 5: Een cykel die gesplitst kan worden

Stel immers dat we in onze begintop komen en we kunnen het pad sluiten. Als we op dit moment meerdere mogenlijkheden hebben en dus branchen, zullen we, voor elke branch die het pad nu nog niet sluit, na een bepaalde tijd weer terugkomen in deze begintop, immers, het pad moet gesloten worden. We krijgen dan een cykel zoals de cykel die achtereenvolgens door de toppen $\{1, 2, 3, 4, 1, 5, 6, 7, 1\}$ gaat (zie Figuur 5). Dus, in het algemene geval, een cykel die een even aantal keren door een bepaalde top gaat. We zien dat we deze cykel eigenlijk kunnen splitsen in twee cykels, in dit geval $\{1, 2, 3, 4\}$ en $\{1, 5, 6, 7\}$. In het algemene geval kunnen we een dergelijke cykel splitsen in tenminste twee cykels. Aangezien we een embedding zoeken met zoveel mogelijk cykels, is het dus niet nodig om nog te branchen als we onze cykel kunnen sluiten, dit zou immers toch een embedding geven met minder cykels. Vandaar dat we dit niet doen.

6.5 Het nemen van kadidaatbogen volgend op een boog

Onze embedding definiëert voor elke top een bepaalde volgorde van de bogen in deze top. We slaan deze volgordes op in de klasse `CycleNode`. Initieel kan elke boog volgen op elke andere boog. We stellen dit voor als n deelvolgordes

$$(e_1)(e_2)(e_3)\dots(e_n)$$

voor een top met n bogen. Stel dat we op een bepaald moment in ons algoritme in de top toekomen via e_i en weggaan via e_j. Vanaf dit moment moeten we hiermee rekening houden,

mochten we nog eens in de top komen. We slaan dit op als

$$(e_1)(e_2)(e_3)\dots(e_ie_j)\dots(e_n)$$

Hoe vinden we nu de kandidaatbogen? Stel dat we uit e_i komen. We hebben in onze top de volgende volgordes opgeslaan:

$$(e_{a1}e_{a2}\dots e_{ax})(e_{b1}e_{b2}\dots e_{by})\dots(e_{c1}e_{c2}\dots e_{cz})$$

We weten dat er nog geen boog volgt op e_i, dus zal e_i het laatste element zijn in een deelvolgorde.

We kunnen bijvoorbeeld niet e_{a2} als volgende boog kiezen, omdat we al gedefiniëerd hebben dat deze op e_{a1} volgt. Stel dat $e_i = e_{ax}$, dan kunnen we niet als volgende boog e_{a1} kiezen, want op dat moment zouden we de deelvolgorde *sluiten*. Dit mag niet, want, zoals in de opgave beschreven wordt, moeten we uiteindelijk een volgorde als $(e_1e_2\dots e_n)$ uitkomen, zodat als we de volgorde doorlopen, elke e tegenkomen.

Meer algemeen zijn de deelkandidaten die kunnen volgen op e_i alle e_j die voldoen aan twee voorwaarden:

- e_j is het eerste element van een deelvolgorde $(e_je_{j+1}...e_{j+n})$.
- e_j zit niet in dezelfde deelvolgorde als e_i.

Een uitzondering bestaat wanneer we slechts één deelvolgorde $(e_1e_2\dots e_i)$ (e_i is dan het laatste element, aangezien het als enige e nog geen opvolger heeft) meer hebben. Dit geval is echter triviaal, dan is de enige kandidaat e_1.

6.6 Tijdscomplexiteit

In het slechtste geval zal ons algoritme alle mogelijke embeddingen overlopen. Een embedding geeft elke top een bepaalde permutatie. Deze permutaties worden gekozen uit de verzameling van alle niet-cyclische permutaties van de buren van de top. Het aantal niet-cyclische permutaties voor een top met n buren wordt gegeven door $(n-1)!$.

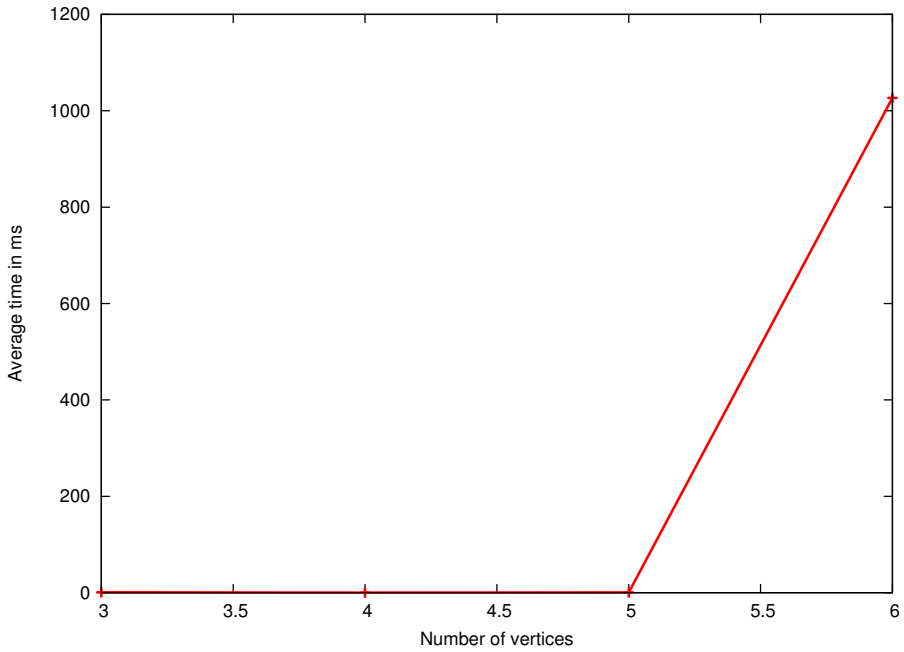
Stel dat de top met het maximaal aantal buren n buren heeft, en dat er in totaal m toppen zijn. Een bovengrens voor het aantal embeddingen E wordt dus gegeven door

$$E(m,n) = O(m \cdot n!)$$

elke embedding is een blad in onze zoekboom. Per top in onze zoekboom bezoeken we een top. De tijd die we over een top doen wordt gegeven door $O(n)$. En dus

$$T(m,n) = O(m \cdot n \cdot n!)$$

Experimentele tijdmetingen ondersteunen het feit dat de tijdscomplexiteit exponentiëel is (zie figuur 6).



Figuur 6: Tijdsmetingen van het naïeve algoritme

7 Bounding Criteria

7.1 Een ondergrens voor het maximaal aantal vlakken

We kunnen bounden als we een ondergrens m hebben voor het maximaal aantal vlakken F. We zoeken immers F. Stel dat we in het algoritme zitten en we hebben momenteel f vlakken. We hebben eerder al een embedding gevonden met m vlakken. Stel dat we weten dat we x vlakken kunnen maken met de bogen die we op dat moment nog niet gebruikt hebben. Dat betekent dat we kunnen stoppen met ons algoritme zodra

$$f + x \leq m$$

want we weten dat

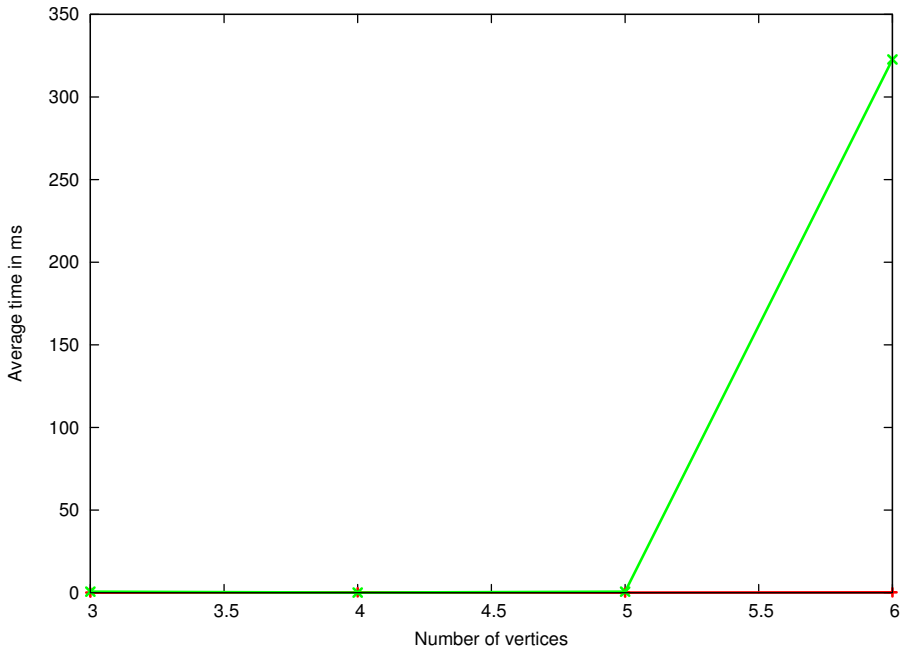
$$m \leq F$$

en dus

$$f + x \leq F$$

Dit laatste impliceert dat we toch geen embedding met meer vlakken zullen vinden in de huidige branch van de zoekruimte, en dus kunnen we maar beter stoppen.

Het is nu de vraag hoe we x kiezen. Om ons algoritme performant te maken, willen we veel bounden, en dus willen we dat $f + x < m$ zoveel mogelijk voorkomt. We willen x dus zo klein mogelijk, maar natuurlijk nog altijd correct, omdat we niet te vroeg willen bounden. We willen natuurlijk ook niet te veel tijd besteden aan x te berekenen, omdat zo het voordeel dat we krijgen door bounding weer zal wegvallen.



Figuur 7: De rode tijden zijn van het algoritme waar het bounden gebaseerd op 3-bogen-per-vlak wordt toegepast. Deze zijn duidelijk veel sneller dan de tijden van het algoritme wanneer dit criterium niet wordt toegepast (groen).

7.2 Een eerste poging

We weten dat we om een vlak te maken tenminste 3 bogen nodig hebben. We houden dus het aantal bogen dat we nog niet gebruikt hebben bij in l. We vinden dan dat

$$x = \lfloor \frac{max(2, c) + l}{3} \rfloor$$

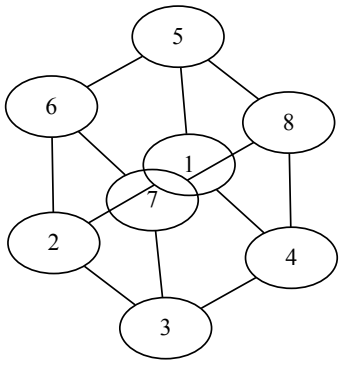
met c het aantal bogen in de huidige cykel. We tellen $max(2, c)$ op bij l omdat we zo x preciezer maken. Als we al meer dan 2 bogen in de huidige cykel hebben, kunnen we met 1 boog uit l een extra cykel maken. Anders hebben we er $3 - c$ nodig. Dit zit op deze manier ook in onze berekening.

Dit bounding algoritme is al een zeer sterke verbetering. Dit is te zien in Figuur 7.

7.3 Girth van een graaf

In 7.2 gingen we ervan uit dat we tenminste 3 bogen nodig hebben om een vlak te maken. Dit is natuurlijk altijd waar, maar stel dat we een graaf hebben als in Figuur 8. Als we deze graaf bekijken, zien we dat we tenminste 4 bogen nodig hebben om een vlak te maken. Aan de hand van deze informatie kunnen we een betere x opstellen voor deze graaf.

$$x = \lfloor \frac{max(3, c) + l}{4} \rfloor$$



Figuur 8: Een graaf met *girth* 4

We vragen ons natuurlijk af hoe we dit algemeen kunnen toepassen. Het *girth* van een graaf is de lengte van kleinste cykel. Dit is precies wat we nodig hebben! We kunnen dus voor elke graaf schrijven:

$$x = \lfloor \frac{max(girth - 1, c) + l}{girth} \rfloor$$

Hoe moeten we het *girth* van een graaf nu berekenen? We kiezen voor een eenvoudige manier. We starten vanuit elke top van de graaf een *breadth-first search* en houden de diepte bij. Op het moment dat we een top terechtkomen waar we al geweest zijn, hebben we de kleinste cykel van de graaf gevonden waar de wortel van de BFS in ligt. Door een BFS te starten vanuit elke top, kunnen we de kleinste cykel van de gehele graaf bepalen. Dit algoritme is geïmplementeerd in de klasse `FindGirth`.

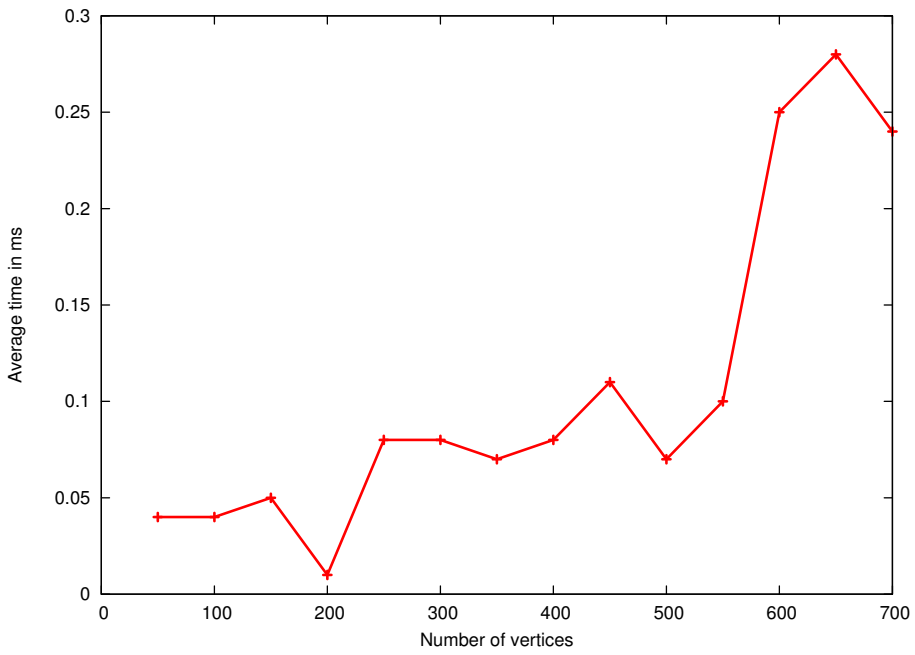
In het slechtste geval moet de BFS voor elke top de gehele graaf doorzoeken. De tijdscomplexiteit voor dit algoritme wordt dus naar boven begrensd door $O(v \cdot e)$ met v het aantal vertices, en e het aantal (ongerichte) bogen. In Figuur 9 zien we dat de tijd die we spenderen aan het zoeken van het *girth* verwaarloosbaar is tegenover de tijd die we spenderen aan het zoeken van het genus.

Na experimentele tijdsmetingen zien we dat het algoritme niet veel sneller gaat (zie Figuur 10). We kunnen dit eenvoudig verklaren: de meeste grafen die we random genereren zullen *girth* 3 hebben.

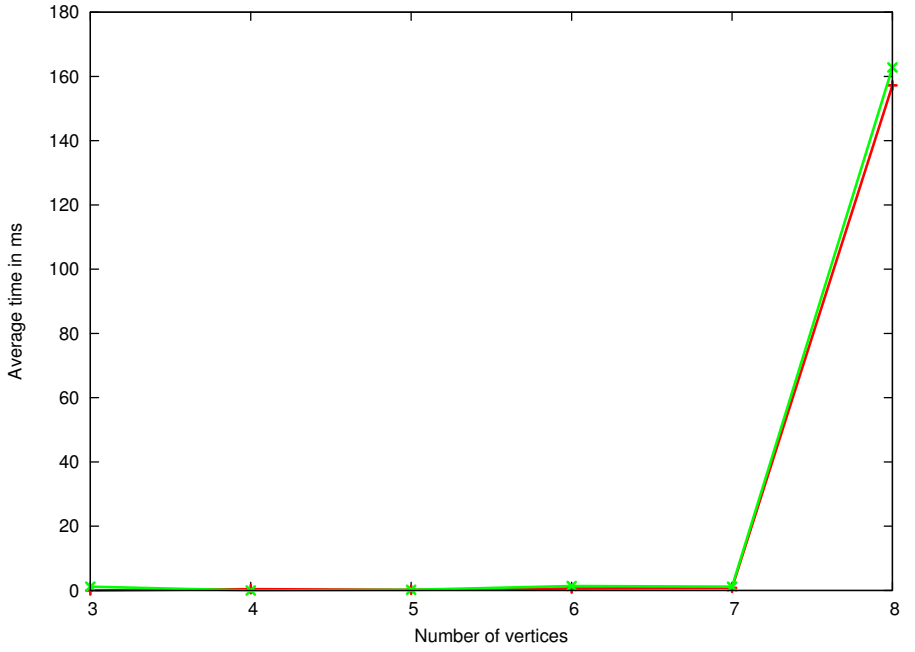
Echter, voor grafen die $girth > 3$ hebben zal het algoritme wel veel sneller gaan. Een voorbeeld is de *4-cube* graaf, die van gemiddeld 13242ms naar 34ms gaat.

7.4 Nog een beetje beter

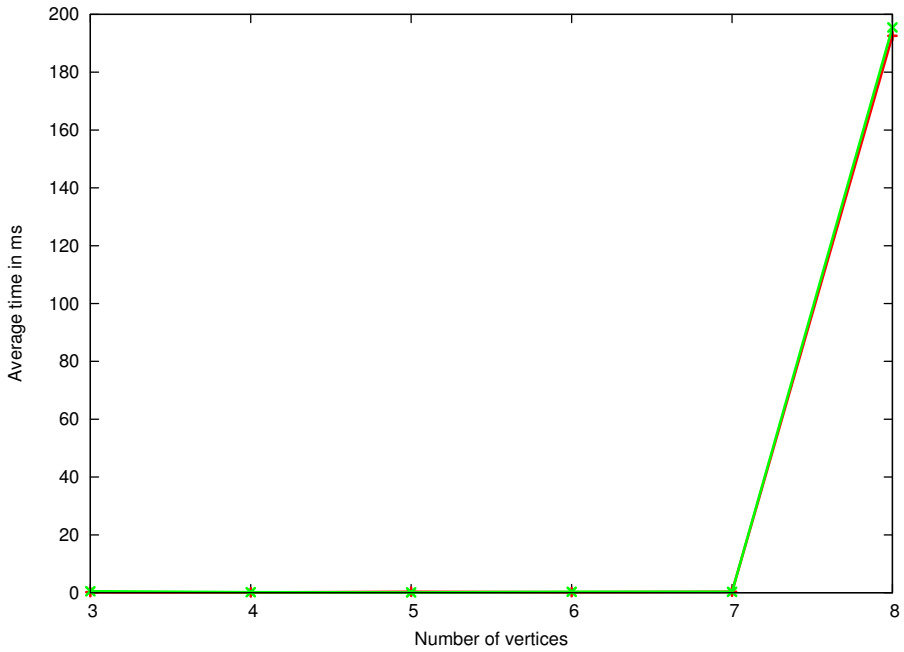
Door de manier waarop `DefaultGraph` geïmplementeerd is, kunnen we in constante tijd kijken of er een boog bestaat tussen twee toppen. Dit kunnen we gebruiken om onze x nog een beetje



Figuur 9: We zien dat het zoeken van het *girth* van een graaf zeer snel kan gebeuren.



Figuur 10: De rode tijden zijn van het algoritme waar we gebruik maken van het *girth* van een graaf. De groene tijden maken hier geen gebruik van. We zien dat er nauwelijks verschil is.



Figuur 11: De rode tijden van het algoritme zijn de tijden waarin de optimalisatie uit 7.4 is toegepast. We zien dat het algoritme een beetje sneller wordt.

beter te maken.

$$x = 1 + \lfloor \frac{l - n}{girth} \rfloor$$

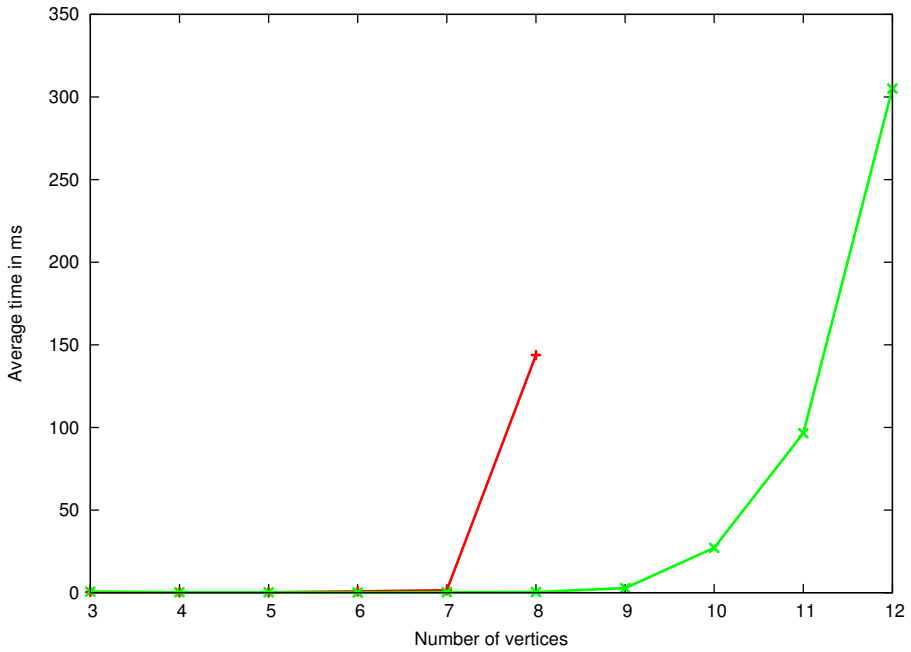
met l opnieuw het aantal bogen dat nog niet gebruikt is, en n staat hier voor het aantal bogen voor dat we nog nodig hebben om het huidige vlak af te maken.

Het aantal vlakken dat we nog kunnen maken is dus 1 (namelijk, het vlak dat we momenteel aan het maken zijn) plus $\lfloor \frac{l-n}{girth} \rfloor$: het aantal bogen dat we nog over zullen hebben nadat het huidige vlak af is, gedeeld door het *girth* van de graaf.

Nu moeten we dus nog vinden wat n is, dus het minimaal aantal bogen dat we nog nodig hebben voor het huidige vlak af te maken. Stel c het aantal bogen dat we al gebruikt hebben in het huidige vlak. Als $c < girth$, dan $n = girth - c$.

Anders, als $c \geq girth$, dan zijn er twee mogenlijkheden. Als de huidige top een buur is van de top waar we het vlak begonnen zijn, dan $n = 1$. Anders hebben we nog zeker 2 vlakken nodig, en dus $n = 2$.

Dit versnelt ons algoritme een klein beetje, zoals we kunnen zien in Figuur 11.



Figuur 12: De groene tijden zijn van het algoritme waar we de pariteitseigenschap uit 7.5 toepassen. We zien dat het algoritme veel sneller wordt.

7.5 Pariteit van het aantal vlakken

Als we de formule

$$g = 1 - \frac{v + f - e}{2}$$

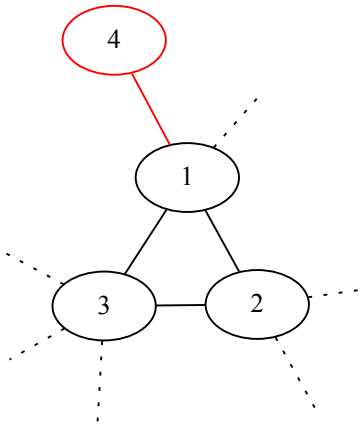
bekijken, en we weten dat g altijd een geheel getal is, kunnen we hieruit afleiden dat $v + f - e$ altijd even is. Dit betekend dat, aangezien v en e vastliggen voor de graaf, f dezelfde pariteit zal behouden. Met andere woorden, vinden we een embedding E_1 met f_1 vlakken, en een embedding E_2 met f_2 vlakken, dan weten we dat ofwel f_1 en f_2 beide even zijn, ofwel beide oneven.
Als we ons boundingcriteria uit 7.1 bekijken,

$$f + x \leq m$$

kunnen we dit nog verbeteren. Stel immers dat er een embedding kan gevonden met aantal vlakken f_1 zodat $f_1 > m$. Aangezien m ook een aantal vlakken gevonden in een embedding voorstelt, zal wegens het feit dat f_i en m dezelfde pariteit bezitten ook gelden dat $f_1 > m+1$. We kunnen dus nog iets sneller bounden, namelijk zodra

$$f + x \leq m + 1$$

Dit gegeven versneld het algoritme zeer sterk. Dit is te zien in Figuur 12.



Figuur 13: Een graaf waaruit top 4 verwijdert kan worden

8 Preprocessing

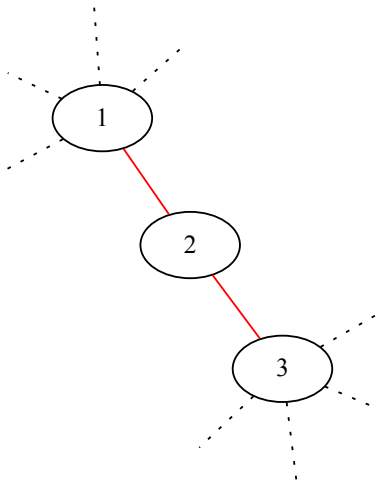
8.1 Vereenvoudingen van de graaf

We vragen ons af of sommige ingewikkelde grafen niet kunnen gereduceerd worden tot eenvoudigere grafen met hetzelfde genus.

8.2 Toppen met slechts 1 buur

Stel dat een bepaalde graaf G kan getekend worden op een bepaalde figuur zonder dat er bogen snijden. Voeg nu een bepaalde top v toe en verbind deze met 1 top w die al in de graaf zat. Noem de boog e. Positioneer eerst e zo dat e niet samenvalt met een andere boog van w. Dit is mogenlijk, aangezien w een eindig aantal bogen heeft. Stel dat e nu een andere boog f snijdt. Kies de lengte van e kleiner zodat e niet meer snijdt met f. Ga zo door tot e geen enkele boog meer snijdt. Dit is mogenlijk aangezien G een eindig aantal bogen heeft. We hebben nu aangetoond dat we aan een graaf 1 top mogen toevoegen die slechts 1 buur heeft, en dat het genus niet zal veranderen. Dit impliceert dat we ook een top met 1 buur kunnen verwijderen (zolang het resultaat geen ledige graaf oplevert).

De klasse `SimplifyGraphProcessor` zal dus toppen met slechts 1 buur verwijderen, op een manier dat het genus van de graaf hetzelfde blijft (zie bijvoorbeeld Figuur 13).



Figuur 14: Een graaf waaruit top 2 kan verwijdert worden, mits we een boog toevoegen tussen top 1 en top 3

8.3 Toppen met precies 2 buren, waarbij de buren onderling niet verbonden zijn

Stel dat er een top v is in onze graaf, die precies twee buren heeft. Noem deze buren u en w. Stel dat er geen boog is tussen u en w. Nu kunnen deze twee opeenvolgende bogen vervangen worden door 1 boog. We kunnen dus v verwijderen uit de graaf en een boog tussen u en w plaatsen.

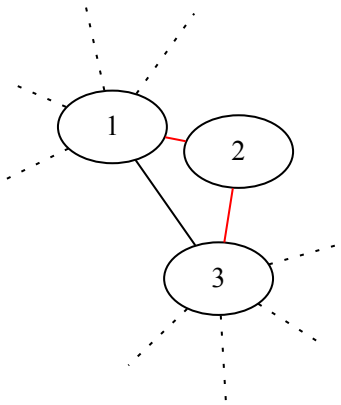
Bijvoorbeeld, men kan eenvoudig zien dat in Figuur 14 de bogen $(1, 2)$ en $(2, 3)$ vervangen kunnen worden door de boog $(1, 3)$.

Ook dit is geïmplementeerd in de klasse `SimplifyGraphProcessor`.

8.4 Toppen met precies 2 buren, waarbij de buren onderling wel verbonden zijn

Stel nu dat we een top v hebben met precies twee buren. Noem deze buren u en w. Stel dat er een boog ligt tussen u en w. We kunnen in dat geval v verwijderen, en het genus zal ongewijzigd blijven. Een voorbeeld is te zien in Figuur 15.

Net zoals we in 8.2 argumenteerden dat we een top met één buur konden toevoegen aan de



Figuur 15: Een graaf waaruit top 2 verwijdert kan worden

graaf door hem zeer dicht bij zijn buur te leggen, kunnen we immers ook een boog toevoegen tussen twee toppen waar er al een boog is. Als we deze zeer dicht bij de originele boog leggen, zullen er geen snijpunten van bogen ontstaan. Als we dan onze nieuwe boog opsplitsen in twee bogen met 1 top ertussen, volgt het gestelde.

Ook dit is geïmplementeerd in de klasse `SimplifyGraphProcessor`. Na deze drie gevallen bekeken te hebben, vragen we ons af wat de snelheidswinst is. Dit is te zien in Figuur 16.

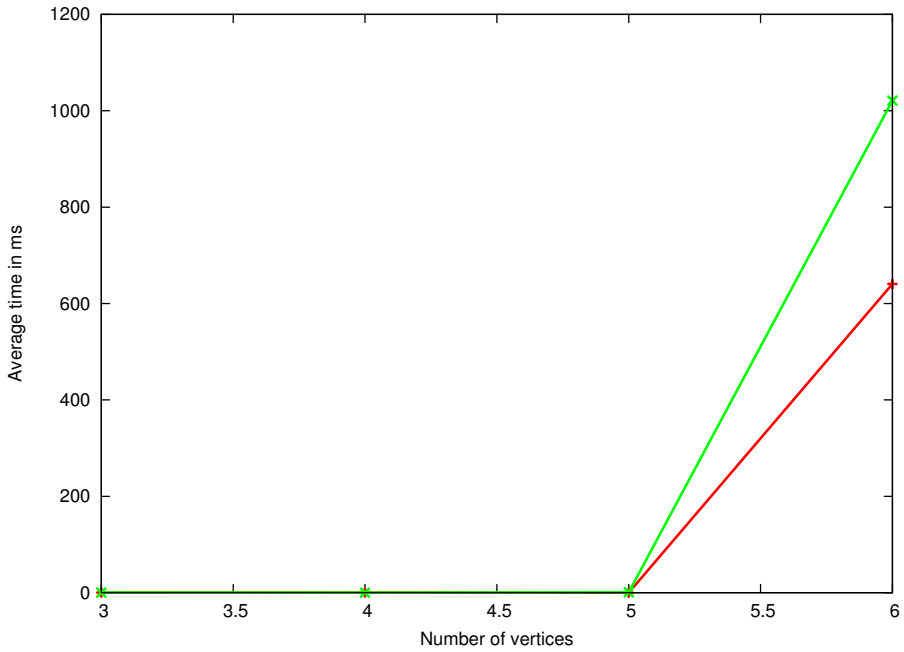
9 Heuristieken

9.1 Een smalle stam voor de zoekboom

Intuïtief willen we een zoekboom die een smalle stam heeft en een brede kruin (zoals in Figuur 17), in plaats van een zoekboom met een brede stam en een smalle kruin (zoals in Figuur 18). Met andere woorden, we willen dat ons algoritme in het begin zo weinig mogelijk moet branchen, en naarmate we dieper in de zoekboom afdalen, mag het meer branchen.

Het voordeel hiervan is dat wanneer we delen van de zoekboom mogen uitschakelen, meestal grotere delen kunnen uitschakelen.

De vraag is natuurlijk hoe we onze zoekboom een zodanige vorm kunnen geven. Dit is echter vrij eenvoudig. We weten dat ons algoritme zal branchen als we in een top staan, en er zijn meerdere kandidaatbogen. Het aantal branches wordt natuurlijk bepaald door het aantal kan-



Figuur 16: We zien dat de tijden van het algoritme met preprocessing (rood) duidelijk onder de tijden zonder preprocessing (groen) liggen.

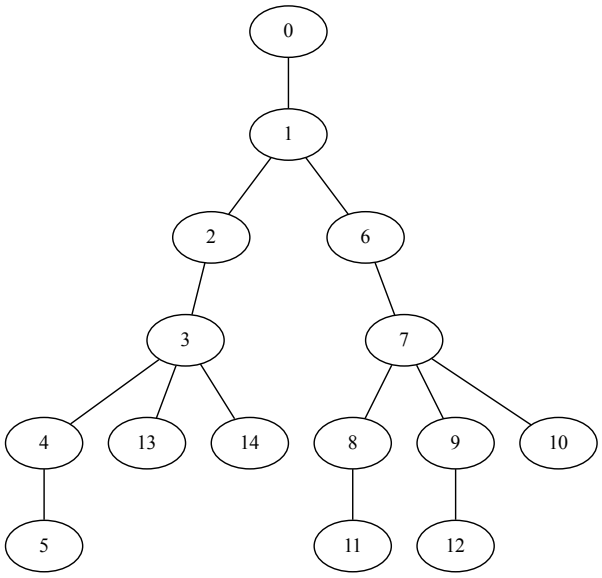
didaatbogen. Het aantal kandidaatbogen is dan weer grotendeels afhankelijk van het aantal bogen in die top.

We willen dus dat ons algoritme eerst de toppen bezoekt met weinig bogen, en vervolgens de toppen met meer bogen. Dit implementeren we door de toppen van de graaf te sorteren op het aantal bogen.

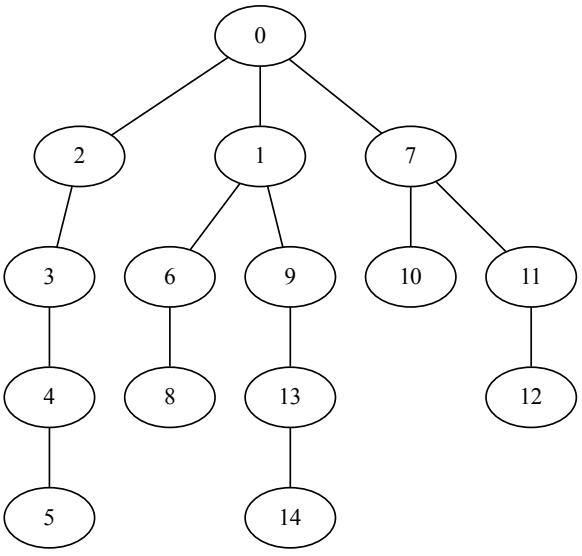
Verrassend genoeg vertraagd dit het algoritme enorm (zie Figuur 19). Dit is zeer contra-intuïtief en we willen natuurlijk een verklaring. We schrijven een klasse `ShowTreeFindGenus` als subklasse van `FindGenus` met als bedoeling de call tree van ons algoritme op een verstaanbare manier te visualiseren. Hiertoe gebruikt het ook de *DOT-taal* van graphviz.

Aan het sorteren zelf kan het niet liggen, dit is een zeer snelle operatie die slechts éénmaal gebeurt, namelijk bij het aanmaken van onze graaf. We aanschouwen de call tree met sorteren (Figuur 20) en de call tree zonder sorteren (Figuur 21).

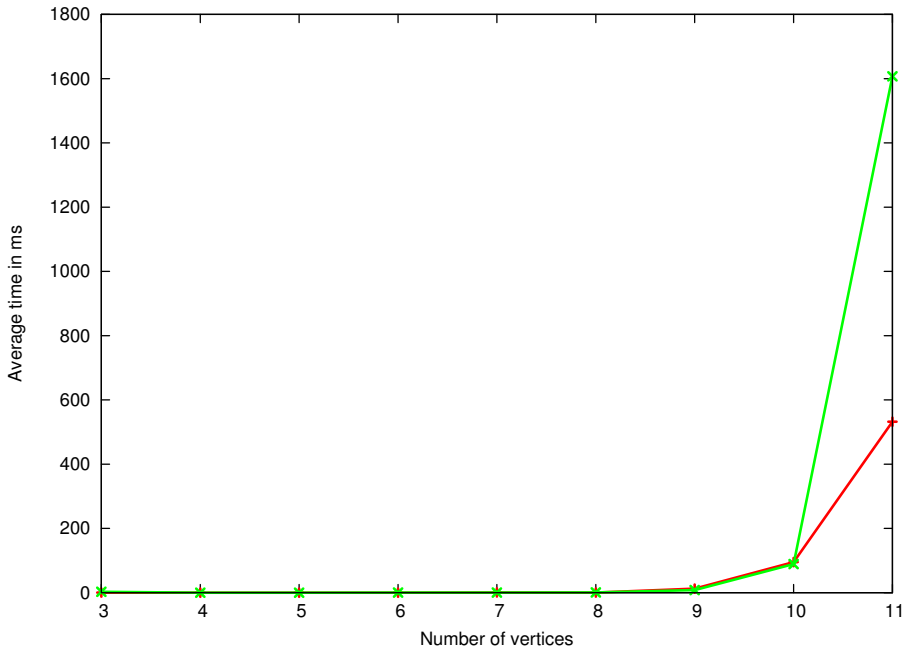
We zien onmiddelijk dat de call tree zonder sorteren veel eenvoudiger is! Ook het aantal bladeren in de boom is verschillend. Dit wordt veroorzaakt door het feit dat we ons pad soms sneller kunnen sluiten (zie 6.4, pagina 8). Dan is het soms beter om te vertrekken vanuit een top met veel buren. Als we immers terug in deze top komen, kunnen we het pad sluiten waardoor er een aantal mogenlijkheden tot branchen zullen wegvallen in deze top. Als er relatief meer buren zijn, zullen er relatief meer mogenlijkheden wegvallen. Langs de andere kant zullen er zo soms meer branches in de zoekboom zijn.



Figuur 17: Een boom met een smalle stam en een brede kruin



Figuur 18: Een boom met een brede stam en een smalle kruin



Figuur 19: We zien dat de tijden van het algoritme met sorteren (groen) duidelijk boven de tijden zonder sorteren (rood) liggen.

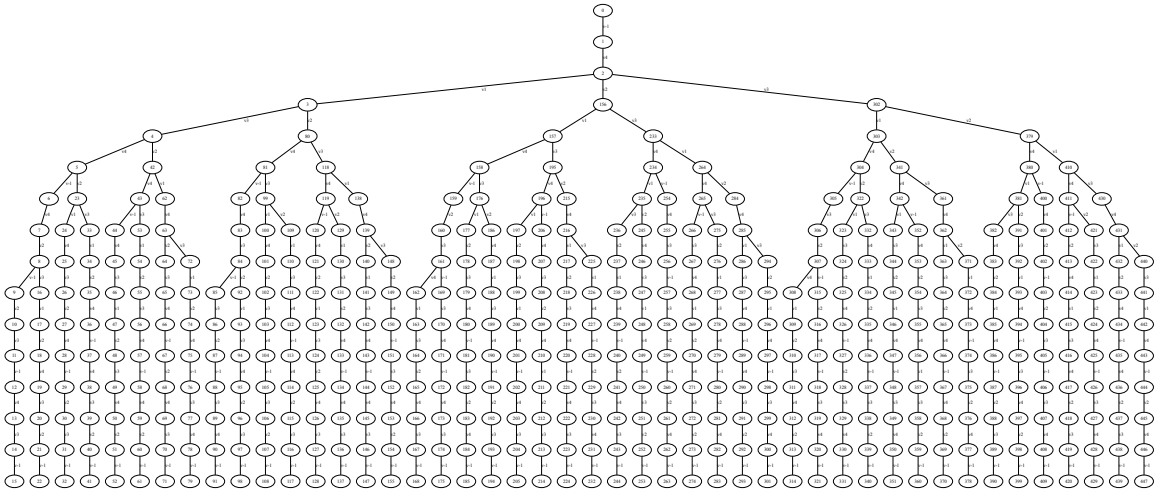
We besluiten dus dat sorteren soms goed kan zijn voor specifieke grafen, maar dat het in het algemene geval meestal geen verbetering brengt.

9.2 Complete grafen

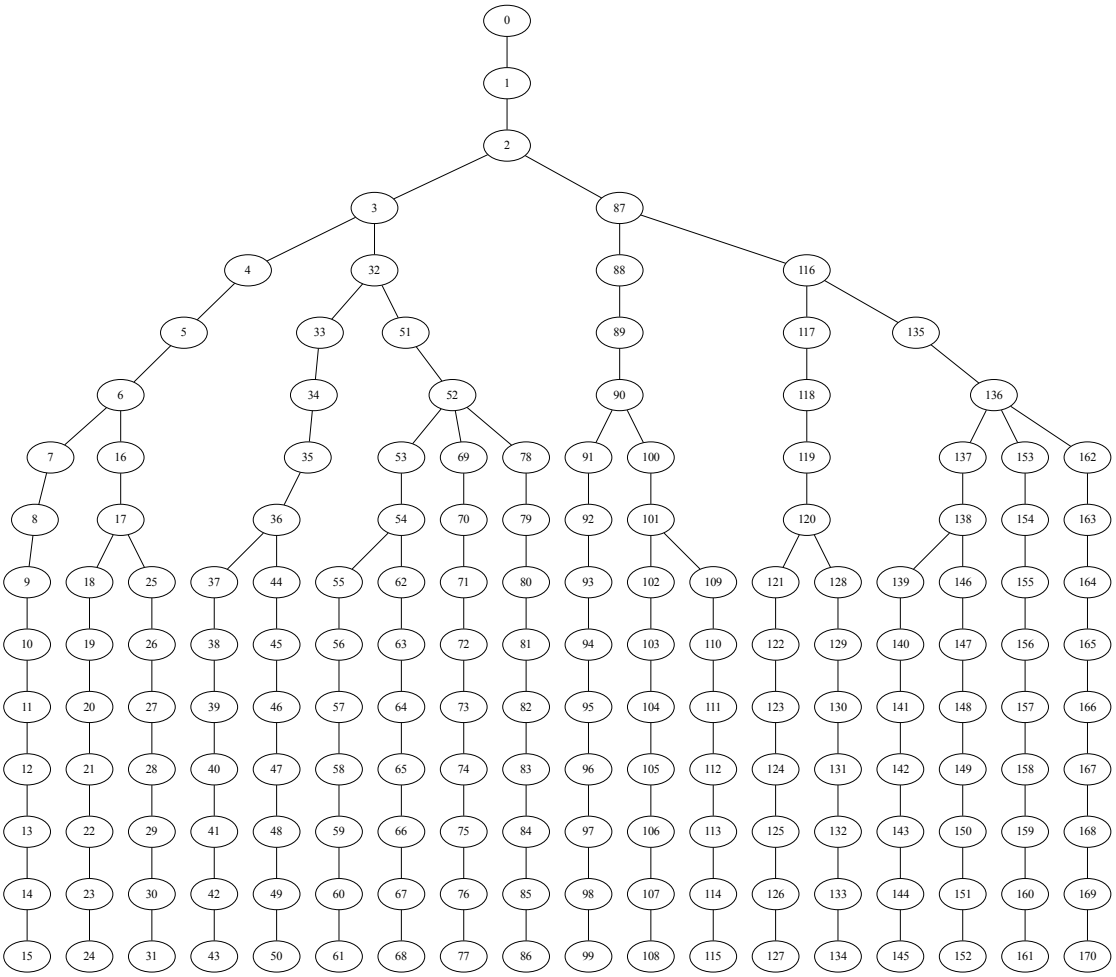
Het is zeer makkelijk om na te gaan of een gegeven graaf een complete graaf is. Aangezien we het genus van een complete graaf zeer snel kunnen bepalen met de formule die we zagen in 4.2, zullen we dit ook gebruiken om ons algoritme te versnellen.

We voegen dus een simpele `if` toe in het begin van ons algoritme die kijkt of onze graaf een complete graaf is. Indien dit het geval is, wordt de formule gebruikt. Anders gaan we verder met ons gewoon algoritme.

Deze heuristiek zorgt er dus voor dat we het genus complete grafen zeer snel kunnen berekenen.



Figuur 20: De call tree met sorteren



Figuur 21: De call tree zonder sorteren