Project: Datastructuren & Algoritmen 2

Quinten Lootens

November 27, 2016

1 Theoretische vragen

1.1 Vraag 1: oplossing

Beschouw de volgende graaf $G = (V, E)$ waarbij:

$$V = \{v_1, v_2, v'_1, ..., v'_k, v''_1, ..., v''_k\}$$

$$E = \{\{v_1, v'_i\} \mid i = 1...2 * k - 1\} \cup \{\{v_2, v''_i\} \mid i = 1...2 * k - 1\} \cup \{v_1, v_2\}$$

Voor elke $k \in \mathbb{N} \setminus \{0\}$ kan er dan een graaf worden opgesteld, waarvoor de dominante verzameling die men bekomt via het algoritme in de opgave minstens k keer groter is dan de optimale dominante verzameling.

1.2 Vraag 2: oplossing

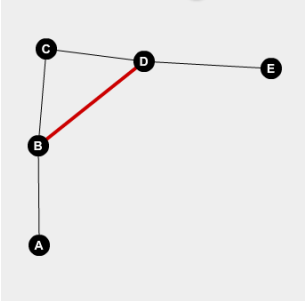
Stelling 1. *Een hamiltoniaanse cykel in een vlakke triangulatie heeft evenveel vlakken binnen de cykel liggen als buiten de cykels.*

Proof. Beschouw de hamiltoniaanse cykel C die in een vlakke triangulatie G ligt. Er liggen geen toppen in C vanwege de definitie van hamiltoniaanse cykel $\Rightarrow$ het aantal vlakken binnen C is $\mid V(G) \mid - 2$. We weten dat het aantal vlakken in G gelijk is aan $2 * \mid V(G) \mid - 4$, dus het volstaat om aan te tonen dat het aantal vlakken binnen $C = \mid V(G) \mid - 2$. We gaan dit bewijzen door inductie op $n = \mid V(G) \mid$

Inductiebasis: Stel $n = 3$, dan hebben we een driehoek. Het aantal vlakken in die driehoek moet dus gelijk zijn aan 1. De stelling geeft ons $\mid V(G) \mid - 2 = n - 2 = 3 - 2 = 1$. Dit klopt.

Inductiestap: Stel het aantal driehoeken in $C = n - 2$ *(Inductiehypothese).* Beschouw een cykel C' op $n + 1$ toppen.

Figure 1: Een deel van de cykel C', namelijk E en A, is nog verder verbonden met de cykel.



Laat top C weg en verbind $B \sim D$. Zo krijgen we een cykel op n toppen. Door de *inductiehypothese* liggen hierbinnen $n - 2$ vlakken. Door $\triangle BCD$ hieraan toe te voegen, bekomen we $(n + 1) - 2 = n - 1$ driehoeken, hetgeen we moesten aantonen. $\square$

1.3 Vraag 3: oplossing

1.3.1 Gretig algoritme om een zo klein mogelijke dominerende verzameling te vinden

We beschouwen de graaf G. De bedoeling is om een zo klein mogelijke dominerende verzameling van G te bekomen.

Om het algoritme goed te begrijpen is het belangrijk om een onderscheid te maken tussen drie verschillende eigenschappen van een knoop: `opgenomen`, `onbedekt` of `bedekt`. Een `opgenomen` knoop is een knoop die in de dominerende verzameling zit. Een `onbedekte` knoop is een knoop die geen `opgenomen` knoop als buur heeft. Een `bedekte` knoop is een knoop die een `opgenomen` als buur heeft. Telkens het algoritme een knoop toevoegt aan de dominerende verzameling, zullen de knopen van de graaf dus mogelijks van status wijzigen.

Data: Graaf G
Result: Dominerende verzameling van G
`dominerende_verzameling` = {};
`onbedekte_knopen` = {alle knopen van G};
for *knoop in onbedekte_knopen* **do**
  if *knoop heeft 1 buur* **then**
    `dominerende_verzameling`.add(buur);
    buur.`markeer`();
  end
end
while *(! onbedekte_knopen.isEmpty())* **do**
  knoop = `haal_volgende_onbedekte_knoop()`;
  beste_keuze = `grootste_aantal_onbedekte_knopen_`(knoop);
  `dominerende_verzameling.add`(beste_keuze);
  beste_keuze`.markeer()`;
end
return `dominerende_verzameling`;

Algorithm 1: Gretig algoritme om dominante verzameling van graaf te bekomen

**

Bij de eerste twee regels van het algoritme initialiseer ik twee verzamelingen. De verzameling `dominerende_verzameling` is een HashSet<Node> waarin we alle `opgenomen` knopen plaatsen. Initieel is dat dus een lege Set<>.
De verzameling `onbedekte_knopen` is een HashSet<Node> en bevat alle `onbedekte` knopen. Initieel bevat deze alle knopen van de graaf G.

Eerst begin ik met een **for-loop** om de buur van knopen met één buur aan de `dominerende_verzameling` toe te voegen, zoals bijvoorbeeld bij *theorie opgave 1* in graaf G. Ik `markeer()` deze buur (de manier waarop dit gebeurt, wordt toegelicht in volgende paragraaf). De **for-loop** heeft een lineaire tijd aangezien hij exact 1 keer over de knopen itereert.

Met een **while** loop vullen we de `dominerende_verzameling` verder aan. Deze blijft lopen totdat de verzameling `onbedekte_knopen` leeg is.

De eerste stap in de **while** loop is het bepalen van de volgende `onbedekte` knoop. Daarvoor gebruik ik de functie `haal_volgende_onbedekte_knoop()`. Deze neemt de eerstvolgende knoop uit de verzameling `onbedekte_knopen`.

In de tweede stap ga ik op zoek naar de beste keuze rond knoop. Deze bepaal ik aan de hand van de functie `grootste_aantal_onbedekte_knopen`(knoop). Deze werkt zoals hierna aangegeven. Beschouw de knoop v, dan is $u(v) = \mid U(v) \mid$ waarbij $\mid U(v) \mid = \{$ *onbedekte* knopen rond v met v incluis $\}$. De functie `grootste_aantal_onbedekte_knopen`(knoop) gaat dus uit de verzameling $L = \{ v \mid v \in$ knoop.$getBuren() \}$ de knoop terug geven met de hoogste $u(knoop)$. Om de complexiteit laag te houden, maak ik gebruik van een HashSet<Node> in elke knoop. Deze HashSet<> houdt de $u(v)$ van knoop v bij. Het opvragen, of aanpassen, gebeurt dus constant.

In de derde stap voeg ik dan de gekozen knoop toe aan de verzameling `dominerende_verzameling`. Daarna voer ik de bewerking `markeer()` uit op die knoop. In deze bewerking ga ik de status van de knoop en die van zijn buren updaten en de betreffende HashSets<Node> bijwerken. Tijdens deze bewerking worden alle knopen die van status `onbedekt` veranderen, uit de verzameling

`onbedekte_knopen` verwijderd. Aangezien deze bewerkingen over HashSets<> gaan, zijn deze constant.

Als de **while** loop klaar is *return* ik de `dominerende_verzameling`.

De tijdscomplexiteit van dit algoritme is lineair omdat bij elke iteratie van de **while** loop één knoop wordt toegevoegd aan de verzameling `dominerende_verzameling` en bijgevolg dus minstens één knoop wordt verwijderd uit de verzameling `onbedekte_knopen`. Mocht het kwadratisch zijn, dan zouden er in elke iteratie van de **while** loop, minstens n iteraties zijn. De **while** loop zal hoogstwaarschijnlijk niet n keer itereren, tenzij elke knoop juist 2 buren heeft. Indien dit zo is, zal het algoritme $2*n$ keer itereren, hetgeen lineair blijft. Stel nu dat er een knoop v is die $n-1$ buren heeft, en dat hij op die manier $n-1$ keer kan overlopen worden, dan zou het algoritme kwadratisch kunnen zijn. Dit zal echter niet gebeuren, aangezien hij al vanaf de eerste stap zal worden herkend door de functies en opgenomen zal worden. De **while** loop zal hier één iteratie hebben. Tijdens die iteratie zal hij alle buren afgaan. Dit zijn minstens $n-1$ iteraties, hetgeen dus ook lineair is.

De gretigheid van dit algoritme is te vinden in het feit dat je bij de keuze van een knoop die je wil toevoegen aan de verzameling `dominerende_verzameling`, een keuze maakt die op dat moment, kennis houdende van de status van de graaf G op dat moment, het beste lijkt om zo efficiënt mogelijk zo'n klein mogelijke dominerende verzameling van G op te bouwen.

1.3.2 Gretig algoritme om al dan niet een hamiltoniaanse cykel te bekomen

Alvorens ik het algoritme start, ga ik eerst enkel bewerkingen uitvoeren om zo beter het algoritme te kunnen toepassen. Beschouw de triangulatie G met knopen G.knopen() en bogen g.bogen().

Combinatorische inbedding Om de combinatorische inbedding van G te bekomen, maak ik gebruik van het volgend algoritme. Het resultaat sla ik op in een veld `combinatorischeInbedding`.

Data: Triangulatie G
Result: Combinatorische inbedding van G, Map<Node, ArrayList<Node>
for *knoop in G.knopen()* **do**
  `List Buren` = {};
  for *boog in knoop.bogen()* **do**
    `Buren`.add(boog.getBuur(*knoop*));
  end
  `combinatorischeInbedding`.put(*knoop*, `Buren`);
end
return `Map`;

Algorithm 2: Combinatorische inbedding van Triangulatie G

**

De complexiteit van dit algoritme is lineair. Een vlakke triangulatie heeft $3 * n - 6$ bogen (n = aantal knopen in G). Deze bogen zullen we maximum twee keer overlopen. Bijgevolg hebben we maximum $6 * n - 12$ iteraties, wat dus een lineaire complexiteit oplevert.

Driehoeksbogen aanmaken Ik maak speciale driehoeksbogen aan via de combinatorische inbedding. Deze driehoeksbogen steek ik in een veld `koppelKnopenDriehoeksboog`.

Data: `combinatorischeInbedding`
Result: `koppelKnopenDriehoeksboog` = HashMap<Set<Knoop>, Driehoeksboog>
for *knoop in G.knopen()* **do**
  `List<Knoop> listCombi` = `combinatorischeInbedding`.get(knoop);
  for *knoop2 in listCombi* **do**
    `Set<Knoop> set` = {};
    `set`.add(knoop, knoop2);
    `koppelKnopenDriehoeksboog`.put(`set`, newDriehoeksboog);
  end
end

Algorithm 3: Driehoeksbogen van Triangulatie G

**

De complexiteit is analoog aan het voorgaande aangezien ik de knopen op een analoge manier overloop.

Driehoeken aanmaken Nu zijn we in staat om driehoeken aan te maken. Alle driehoeken van G sla ik op in een veld `driehoeken`.

```
Result: driehoeken = List<Driehoek>
for knoop in G.knopen() do
    List<Knoop> listCombi = combinatorischeInbedding.get(knoop);
    for knoop2 in listCombi do
        derdeKnoop = bepaalDerdeKnoop(knoop, knoop2);
        if combinatorischeInbedding.get(derdeKnoop).contains(knoop) then
            driehoek = new Driehoek(knoop, knoop2, derdeKnoop;
            if ! driehoeken.contains(driehoek) then
                driehoeken.add(driehoek);
            end
        end
    end
end
```

Algorithm 4: Driehoeken van Triangulatie G

De complexiteit is analoog aan het voorgaande aangezien ik de knopen op een analoge manier overloop.

Hamiltoniaanse cykel aanmaken Het principe van dit algoritme richt zich op de driehoeksbogen en driehoeken. Elke driehoeksboog heeft exact twee driehoeken, aan elke kant van de driehoeksboog één. We beginnen met een driehoek, dan heeft elke driehoeksboog van deze driehoek een driehoek in de cykel en een driehoek buiten de cykel. De knopen van deze driehoek steken we in een set `hamiltonKnopen`. Een knoop in een cykel heeft een knoop vanwaar hij komt en een knoop waar hij naartoe moet.
De cykel die we op dit moment hebben is de driehoek. Deze kunnen we uitbreiden door de andere driehoek van een driehoeksboog aan te spreken die nog niet in de cykel ligt. Deze driehoek heeft exact één knoop die nog niet in de set `hamiltonKnopen` zit. We kunnen deze dus verbinden met de knopen van de driehoeksboog. Op deze manier vergroten we de cykel totdat we de hamiltoniaanse cykel van G bekomen.

```
Result: Hamiltoniaanse Cykel
stackBogen = new Stack<Driehoeksboog>;
bogenDieOoitInStackZat = new Set<Driehoeksboog>;
hamiltonKnopen.addAll(knopen van beginDriehoek);
stackBogen.pushAll(driehoeksbogen van beginDriehoek);
bogenDieOoitInStackZat.addAll(driehoeksbogen van beginDriehoek);
while (!stackBogen.isEmpty() and hamiltonKnopen.size() < G.knopen().size() do
    driehoeksboog = stackBogen.pop();
    List<Driehoek> driehoekenVanBoog = driehoeksboog.getDriehoeken();
    VoegToe(driehoeksboog, driehoekenVanBoog.get(0));
    VoegToe(driehoeksboog, driehoekenVanBoog.get(1));
end
if hamiltonKnopen.size() == G.knopen().size() then
    printCykel();
else
    Geen cykel gevonden!
end
```

Algorithm 5: Algoritme voor de cykel te maken in G

**

De functie *VoegToe* bepaalt de derde knoop van de driehoek en gaat na of deze niet reeds in `hamiltonKnopen` zit. Indien dit niet het geval is, zal hij de knopen met elkaar verbinden en

de twee nieuwe driehoeksbogen in de `stackBogen` pushen, behalve de driehoeksbogen die niet in `bogenDieOoitInStackZat` zitten. Daarna voegt hij die toe aan `bogenDieOoitInStackZat`.

Aangezien we maximum alle bogen overlopen (door `bogenDieOoitInStackZat` is dit verzekerd) is de complexiteit van dit algoritme lineair.

2 Verslag

2.1 Verslag van dominerende verzamelingen

2.1.1 Eerste benadering

Mijn eerste algoritme dat ik implementeerde was het volgende:

Data: Graaf G
Result: Dominerende verzameling van G
`dominerende_verzameling` = {};
`knopen` = {alle knopen van G};
`knopen`.sort();
while *(! `knopen`.isEmpty())* **do**
    knoop = `knopen`.get(0);
    `dominerende_verzameling`.add(knoop);
    verwijder knoop en al zijn buren uit `knopen`;
end
return `dominerende_verzameling`;

Algorithm 6: Eerste benadering

**

Ik steek alle knopen van de graaf G in de verzameling `knopen`. Deze verzameling is een ArrayList<Node> die ik dan sorteer op basis van de graad van de knoop.

Het probleem bij dit algoritme is dat het niet lineair is. Een sorteeralgoritme met een lineaire complexiteit is hier niet van toepassing. Ook het verwijderen uit een ArrayList<Node> is in het slechtste geval n. Er is ook geen alternatief. Het gaat vooral om het concept dat je op zoek gaat naar de toppen die een hoge graad hebben, zodat je er veel kan schrappen in één bewerking.

2.1.2 Lineaire aanpak

In plaats van eerst de verzameling knopen van graaf G te sorteren op basis van hun graad, begin ik direct aan de eerste knoop. Ik vergelijk de graad van de knoop met zijn buren en ik kies dan de knoop met de hoogste graad. Deze knoop voeg ik toe aan de verzameling `dominerende_verzameling` en ik verwijder de knoop en al zijn buren uit de verzameling `knopen`.

Data: Graaf G
Result: Dominerende verzameling van G
`dominerende_verzameling` = {};
`knopen` = {alle knopen van G};
while *(! `knopen`.isEmpty())* **do**
    knoop = `knopen`.get(0);
    `beste_keuze` = `get_knoop_hoogste_graad`(knoop);
    `dominerende_verzameling.add`(beste_keuze);
    verwijder `beste_keuze` en al zijn buren uit `knopen`;
end
return `dominerende_verzameling`;

Algorithm 7: Lineaire aanpak

2.1.3 Optimalisatie

Ik heb nu een algoritme dat mij een dominerende verzameling oplevert, maar ik voel dat er nog verbeteringen mogelijk zijn. Omdat het een gretig algoritme moet zijn, denk ik dat ik de keuze voor de beste knoop wel kan verbeteren.

Additionele for-loop voor de losse einden Vooraleer ik het volledige algoritme uitvoer, lijkt het me beter om eerst een aantal gevallen uit te sluiten. Zoals we in *theorie opgave* 1 hebben bestudeerd, bestaat er een graaf G die een slecht resultaat zou kunnen opleveren. Om daarmee om te gaan zou het beter zijn om eerst een **for-loop** over de knopen van de graaf te gaan en telkens als een knoop maar één buur heeft, deze buur toe te voegen aan de dominerende verzameling.

```
for knoop in knopen_graaf do
    if knoop heeft 1 buur then
        dominerende_verzameling.add(buur);
    end
end
```

Algorithm 8: Additionele for-loop voor losse einden

**

Eigenschappen aan knopen toekennen Tot nu toe heb ik telkens elke knoop die ofwel in de dominerende verzameling zit, ofwel verbonden is met een knoop uit deze verzameling, verwijderd uit de lijst met knopen die we afgaan in de **while** loop.

Hiermee laat ik eigenlijk kansen liggen bij de knopen die dan wel verbonden zijn met een knoop uit de dominerende verzameling, maar eigenlijk nog steeds een goede keuze zouden zijn om deze op te nemen in de dominerende verzameling en zo zijn buren als verbonden te beschouwen. Om deze reden ga ik de graaf anders gaan benaderen door eigenschappen aan de knopen toe te kennen.

De drie verschillende eigenschappen van een knoop: *opgenomen*, *onbedekt* of *bedekt*. Zie de uitleg bij 1.3.1. De **while** wordt dan de volgende:

```
Data: Graaf G
Result: Dominerende verzameling van G
dominerende_verzameling = {};
while ( ! onbedekte_knopen.isEmpty()) do
    knoop = haal_volgende_onbedekte_knoop();
    beste_keuze = grootste_aantal_onbedekte_knopen_(knoop);
    dominerende_verzameling.add(beste_keuze);
    beste_keuze.markeer();
end
return dominerende_verzameling;
```

Algorithm 9: Toekennen van toestanden aan knopen

2.2 Verslag van hamiltoniaanse cykels

2.2.1 Eerste aanpak om tot een algoritme te komen

Voor ik begon aan het maken van een algoritme, heb ik mij verdiept in vlakke triangulaties en combinatorische inbedding. Daaruit besloot ik om eerst een algoritme te schrijven om op een lineaire manier een combinatorische inbedding te krijgen van een vlakke triangulatie.

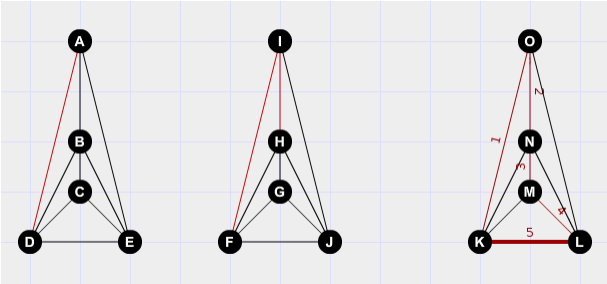
```
Data: Triangulatie G
Result: Combinatorische inbedding van G, Map<Node, ArrayList<Node>
for knoop in G.knopen() do
    List Buren = {};
    for boog in knoop.bogen() do
        Buren.add(boog.getBuur(knoop));
    end
    Map.put(knoop, Buren);
end
return Map;
```

Algorithm 10: Combinatorische inbedding van graaf G

**

Met een combinatorische inbedding was ik in staat om, na het kiezen van een start knoop, een keuze te maken voor de volgende knoop. Door op papier veel met de vlakke triangulaties een

Figure 2: Algoritme in werking



hamiltoniaanse cykel te zoeken, kwam ik op het idee om steeds bij de volgende knoop, de knoop te kiezen die je verkrijgt door in de lijst van combinatorische inbedding de index van de huidige knoop - 1 te doen. Indien deze al op de cykel zit, dan doe je opnieuw - 1, totdat je een knoop vindt die nog niet op de cykel ligt.

Data: Triangulatie G
Result: List<Node> HamiltonCykel
`HamiltonCykel` = {};
`preOrigin` = startpunt;
`knoop` = beginknoop;
while *bestaatErEenVolgendeKnoop*(*knoop*) **do**
    `knoop` = *verkrijgVolgendeKnoop*(`knoop`, `preOrigin`);
    `HamiltonCykel`.add(`knoop`;
end
return `HamiltonCykel`;

Algorithm 11: Hamilton cykel zoeken via meest linkse knoop

**
**
Data: Knoop origin, Knoop preOrigin
Result: Knoop volgendeKnoop
`indexPreOrigin` = `combinatorischeInbedding`.get(origin).indexOf(preOrigin);
`volgendeKnoopIndex` = `getNextIndexToTheLeft`(indexPreOrigin,
 `combinatorialEmbedding`.get(origin));
`volgendeKnoop` = `combinatorialEmbedding`.get(origin).get(`volgendeKnoopIndex`);
while *volgendeKnoop in HamiltonCykel* **do**
    `volgendeKnoopIndex` = `getNextIndexToTheLeft`(indexPreOrigin,
     `combinatorischeInbedding`.get(origin));
    `volgendeKnoop` = `combinatorischeInbedding`.get(origin).get(`volgendeKnoopIndex`);
    `preOrigin` = origin;
end
return `volgendeKnoop`;

Algorithm 12: *verkrijgVolgendeKnoop*

2.2.2 Tussenliggende driehoeken

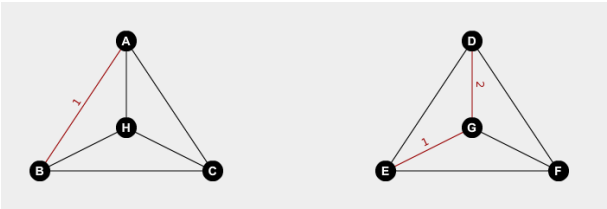
In de vorige sectie heb ik een basis algoritme gemaakt. Nu ga ik verder op zoek om het al dan niet te verbeteren. Voorlopig beschouwde ik enkel de triangulatie aan de hand van een combinatorische inbedding. Combinatorische inbedding is zeer handig bij vlakke grafen, maar een triangulatie heeft ook nog een andere specifieke eigenschap: het bestaat enkel uit driehoeken.

Ik ga nu een onderscheid maken tussen driehoeken en tussenliggende driehoeken. Een tussenliggende driehoek bestaat uit drie driehoeken in één driehoek. Dit is dus een driehoek met een knoop in.

Figure 3: *links* : een driehoek, *rechts* : een tussenliggende driehoek

Ik ga nu het vorige algoritme opnieuw implementeren, maar ik ga nu nakijken of de volgende knoop onderdeel is van een tussenliggende driehoek met zijn oorsprong. Indien dit zo is, ga ik de volgende knoop bereiken door eerst de oorsprong naar de knoop in de tussenliggende driehoek te verbinden, en daarna met de volgende knoop, op voorwaarde dat die knoop nog niet in de cykel zit.

Figure 4: *links* : oude manier, *rechts* : nieuwe manier



Data: Triangulatie G
Result: List<Node> HamiltonCykel
`HamiltonCykel` = {};
`preOrigin` = startPunt;
`knoop` = beginknoop;
while *bestaatErEenVolgendeKnoop*(*knoop*) **do**
  if *heeftTussenliggendeDriehoek.*(`knoop`, `preOrigin` **then**
    Verbind `preOrigin` met de knoop in de tussenliggende driehoek ;
    verkrijgVolgendeKnoop(`knoop`, `preOrigin(TussenliggendeKnoop)`);
  else
    verkrijgVolgendeKnoop(`knoop`, `preOrigin`);
  end
  `knoop` = *verkrijgVolgendeKnoop*(`knoop`, `preOrigin`);
  `HamiltonCykel`.add(`knoop`;
end
return `HamiltonCykel`;

Algorithm 13: Hamilton cykel zoeken via meest linkse knoop rekeninghoudend met tussenliggende driehoeken

**

De resultaten van dit algoritme vielen wat tegen. Ik vond veel cykels die ik ervoor niet vond bij bepaalde grafen, maar vond dan ook bij sommige grafen geen cykels die ik wel met het vorige algoritme vond.

2.2.3 Cykel bepalen aan de hand van graad van de knoop

Hier start ik terug van nul. Ik beschouw de graaf niet meer aan de hand van een combinatorische inbedding, maar ik probeer op een andere manier een cykel op te bouwen. *Ik beschouw in dit deel de graad van een knoop aan de hand van het aantal buren die nog niet in de cykel zitten.*

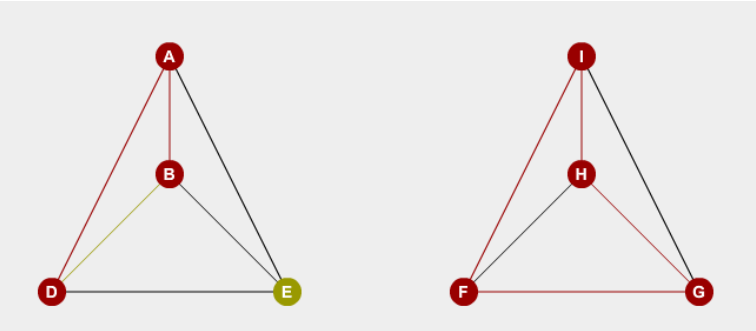
Wat is de meest logische keuze om bij een cykel ergens te starten? Logischerwijze zou dit de knoop zijn met de hoogste graad. Zo heb je het meeste kans om terug bij deze knoop op het einde te geraken. Eenmaal deze knoop gevonden, kunnen we starten met het opbouwen van de cykel. Dan lijkt het logisch om telkens de volgende knoop te kiezen met de laagste graad. Je hebt namelijk minder kans om die nog tegen te komen vergeleken met een knoop met een hogere graad. Zo ga ik telkens verder tot ik uiteindelijk alle knopen heb gevonden of tot ik ergens vastloop.

De resultaten hier waren niet speciaal veel beter of slechter. Aangezien ik niet onmiddellijk een manier vond om dit verder te verbeteren, besloot ik dit pad niet verder te bewandelen en terug te keren met het fixeren op combinatorische inbedding en de driehoeken.

2.2.4 Elke boog heeft twee driehoeken

Tot nu richtte ik mij vooral op de gretigheid van de knopen. In plaats van te kijken vanuit het oogpunt van de knopen, ga ik nu kijken vanuit het oogpunt van de bogen van de driehoeken. Een driehoek is altijd een cykel, en elke driehoeksboog heeft aan elke zijde een driehoek. Mijn plan is nu om eerst een algoritme te schrijven om alle driehoeken van een triangulatie te vinden. Vanuit de driehoeken ga ik mijn cykel opbouwen. Ik kijk telkens naar een driehoeksboog en ga na of er bij een driehoek nog knopen zijn die nog niet in de cykel liggen. Ik begin van een kleine cykel met drie knopen en ga zo uitbreiden door naar de driehoeksbogen te kijken. Dat wordt de kern van mijn algoritme.

Figure 5: Werking Algoritme



3 Experimenten

3.1 Dominerende verzamelingen

Om te experimenteren neem ik als *basis* **Algorithm 9**.

3.1.1 For-loop om de beste beginknoop te bepalen

Tot nu toe heb ik telkens gewoon de eerste knoop uit de verzameling `onbedekte_knopen` genomen. Ik dacht dat het beter zou kunnen zijn om, vooraleer ik aan het algoritme begin, met behulp van een **for-loop** een knoop met de hoogste graad te kiezen. De test gaf aan dat het beter was om gewoon de eerste knoop uit de verzameling te kiezen.

```
max = 0;
max_knoop;
for knoop in knopen_graaf do
    if knoop.graad() > max then
        max_knoop = knoop;
        max = knoop.graad();
    end
end
```

Algorithm 14: For-loop om de beste beginknoop te vinden

**

Diepte twee zoeken Tot nu toe hebben we alles benaderd via diepte 1. We testen eens uit door nu met diepte 2 te werken. We zullen dus meer iteraties hebben, maar wat zou het verschil zijn met diepte 1? De test gaf aan dat het beter is om met diepte 1 te werken.

```
Data: Graaf G
Result: Dominerende verzameling van G
dominerende_verzameling = {};
while ( ! onbedekte_knopen.isEmpty()) do
    knoop = haal_volgende_onbedekte_knoop();
    beste_keuze = grootste_aantal_onbedekte_knopen_diepte_(knoop);
    dominerende_verzameling.add(beste_keuze);
    beste_keuze.markeer();
end
return dominerende_verzameling;
```

Algorithm 15: Diepte 2 zoeken

3.2 Hamiltoniaanse cykels

De experimenten die ik heb uitgevoerd zijn beschreven in het verslag.

4 Resultaten

4.1 Dominerende verzamelingen

Testset	Algorithm 1	Algorithm 7	Algorithm 9	Algorithm 9 + 14	Algorithm 15
$graaf1.sec$	1985	2208	2065	2098	2093
$graaf2.sec$	9341	10302	9616	9741	9731
$graaf3.sec$	2086	2315	2194	2188	2183
$graaf4.sec$	9806	10885	10152	10170	10163
$graaf5.sec$	2500	2882	2850	2910	2911
$graaf6.sec$	10000	11504	11370	11699	11700
$graaf7.sec$	4000	4867	4665	4677	4676
$graaf8.sec$	15000	18235	17468	17549	17549
$triang1.sec$	1697	1852	1738	//	//
$triang2.sec$	8288	9134	8480	//	//

4.2 Hamiltoniaanse cykels

Testset	Algorithm 5	Algorithm 11	Algorithm 13	grafen
*triangalle*5	1	1	1	1
*triangalle*6	2	1	2	2
*triangalle*7	5	4	5	5
*riangealle*8	13	6	12	14
*triangalle*9	48	29	37	50
*triangalle*10	215	102	141	233
*triangalle*11	1094	456	584	1249
*triangalle*12	6344	2151	2591	7595
*triangnonham*1	0	0	0	1
*triangnonham*2	0	0	0	1
*triangnonham*3	0	0	0	1

5 Testen

Voor beide delen zijn de testen geïmplementeerd in de source file.