Lineair algoritme voor Greedy Weighted Graph Matching en de link met het Stable Marriage probleem

Universiteit Utrecht

Bachelorscriptie

Begeleider: Rob Bisseling

Tinka Veldhuis

19 januari 2017

Inhoudsopgave

1 Introductie **3**

2 Grafen en matchings **4**
2.1 Greedy Matching en het Stable Marriage probleem 5
2.2 De link tussen Greedy Matching en het Stable Marriage probleem . . 9
2.3 Het algoritme van Preis 12
2.4 Bewijs lineaire tijdscomplexiteit Preis 17

3 Resultaten **20**

4 Discussie **22**

5 Conclusie **23**

A Algoritme van Preis geïmplementeerd in python **24**

1 Introductie

Het doel van deze scriptie was het bestuderen van Greedy Weighted Graph Matching en het Stable Marriage probleem en de link hiertussen. Beide problemen gaan over het vinden van een stabiele matching van paren in een gewogen graaf. De problemen hebben veel toepassingen in de praktijk. Hierdoor is het bestaan van efficiënte, snelle algoritmes om oplossingen te vinden relevant. Voor het vinden van een Greedy Matching met maximaal gewicht in een algemene graaf heeft Preis in 1999 een lineaire tijd, $\frac{1}{2}$-approximatie algoritme ontworpen [Pre99]. Manne e.a. deden onderzoek naar de link tussen de twee problemen en bewezen in 2016 dat Greedy Matching een speciaal geval is van het Stable Marriage probleem. Zij concludeerden dan ook met de vraag: kan het Stable Marriage probleem in lineaire tijd worden opgelost, waarbij er ongesorteerde prioriteitenlijsten zijn [Man+16]?

Na het bestuderen van de verschillende artikelen heb ik een eigen implementatie van het algoritme van Preis geschreven in Python.

In deze scriptie zal ik beginnen met een inleiding over graaftheorie. Hierna zal ik Greedy Graph Matching en het Stable Marriage probleem uitleggen en de link tussen de twee laten zien. Vervolgens presenteer ik het algoritme van Preis en mijn implementatie hiervan. In de conclusie kom ik nog terug op de link tussen Greedy Matching en het Stable Marriage probleem.

2 Grafen en matchings

Een graaf G wordt gedefinieerd door een verzameling punten V en een verzameling kanten E, waarbij een kant (a, b) een verbinding weergeeft tussen de punten a en b uit V. Wanneer er tussen twee punten a en b een kant loopt, noemen we a en b buren. Een graaf wordt weergegeven als $G = (V, E)$. We gaan in deze scriptie uit van gewogen grafen, waarbij elke kant een gewicht groter dan 0 heeft. Het gewicht geeft de aantrekkingskracht tussen de twee punten aan. In het vervolg gaan we er vanuit dat we het hebben over enkelvoudige grafen: ten eerste lopen hierin kanten altijd tussen twee punten, dus $(a, a) \notin E$ voor alle a, en ten tweede kan er tussen twee punten maximaal één kant lopen. Verder gebruiken we zowel ongerichte als gerichte grafen. In een ongerichte graaf geeft het gewicht van de kant de wederzijdse aantrekkingskracht aan, ofwel gewicht $g(a, b) = g(b, a)$ voor alle kanten (a, b) in E. In een gerichte graaf geeft het gewicht van kant (a, b) aan hoe 'graag' punt a bij punt b in de buurt wil zijn en het gewicht van kant (b, a) hoe graag punt b bij punt a in de buurt wil zijn. Dit kunnen dus verschillende waarden zijn.

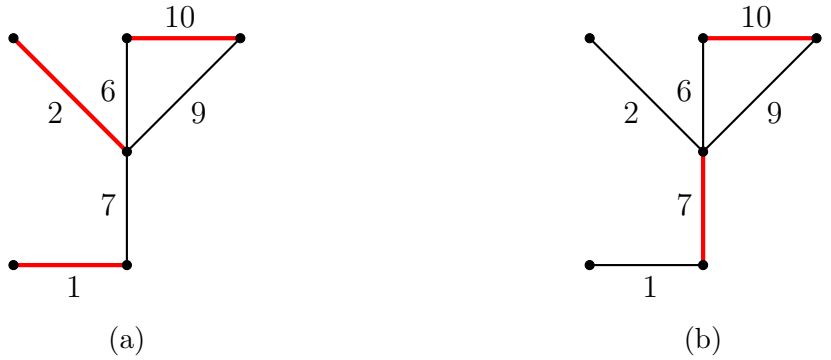
Gegeven een graaf $G = (V, E)$ wordt een matching M gedefinieerd door een deelverzameling van de kanten uit E, waarbij elk punt uit V maximaal één keer voorkomt. Als $(a, b) \in M$, geldt dus voor elke buur c van a dat $(a, c) \notin M$. Het gewicht van de matching M is de som van de gewichten van de kanten in M:

$$g(M) = \sum_{(a,b) \in M} g(a, b).$$

We maken onderscheid tussen een maximale matching en een matching van maximaal gewicht, zie figuur 1: een matching M is maximaal, als er geen extra kant uit E aan kan worden toegevoegd, zonder bovenstaande eigenschap van een matching te verbreken. Een matching M is een matching met maximaal gewicht, als voor alle mogelijke matchings M' van de graaf G geldt:

$$g(M) \geq g(M').$$

De stabiliteit van een matching is erg belangrijk. Wanneer geen enkel paar punten beter af zou zijn als ze naar elkaar toe gaan in plaats van naar hun huidige partners, is de matching stabiel. Denk hierbij aan een huwelijk: als twee mensen liever met elkaar zouden trouwen, dan met degenen waar ze aan gekoppeld zijn, zullen ze erg ontevreden zijn of zelfs beiden hun huidige partner verlaten voor de ander. In dit geval is de matching instabiel.



Figuur 1: In (a) een maximale matching maar geen matching van maximaal gewicht. In (b) een maximale matching met maximaal gewicht.

2.1 Greedy Matching en het Stable Marriage probleem

Matching heeft veel belangrijke toepassingen in de praktijk. In 2012 ontvingen Shapley en Roth de Nobelprijs in de economie voor hun aandeel in het onderzoek naar matchings. Shapley definieerde samen met Gale voor het eerst officieel het Stable Marriage probleem en ze ontwierpen het bekende Gale-Shapley algoritme voor het oplossen hiervan. Roth liet zien dat stabiliteit onmisbaar is voor succesvolle matchings en hij ontwierp systemen voor het matchen van bijvoorbeeld donors aan patiënten en leerlingen aan scholen [Com12]. In 2015 werd er ook voor het eerst matching gebruikt om leerlingen in Amsterdam op middelbare scholen te plaatsen. Er was veel ophef over deze nieuwe methode: veel ouders en leerlingen waren zeer ontevreden. Het algoritme dat werd gebruikt, bleek dan ook instabiel te zijn en na aanpassingen was de tevredenheid over de matching in 2016 al een stuk groter [KBS16].

Door het grote aantal toepassingen is het onderzoek naar algoritmes om goede matchings te vinden, en het parallelliseren van deze algoritmes, flink toegenomen. Het aantal punten en kanten in een graaf kan enorm groot zijn en omdat de algoritmes om matchings te geven van maximaal gewicht vaak een tijdcomplexiteit hebben van het aantal punten maal het aantal kanten, wordt er gezocht naar approximatie-algoritmes [Man+16]. Stel een algoritme heeft een $\frac{1}{2}$-approximatie, dan betekent dat, dat het gewicht van de matching M die eruit komt, minstens de helft is van het maximale gewicht:

$$g(M) \geq \frac{1}{2} * g(M_{max}).$$

Een zeer bekend $\frac{1}{2}$-approximatie algoritme om een matching van een gewogen, ongerichte graaf te vinden is het Greedy Matching algoritme. Hierbij worden alle kanten gesorteerd van hoogste naar laagste gewicht. De kant (a, b) met het hoogste gewicht wordt aan de matching toegevoegd, waarna alle kanten die aan a of b liggen worden

(a) Lloyd S. Shapley

(b) Alvin E. Roth

Figuur 2: Het definiëren en geven van een algoritme van het Stable Marriage probleem en het invoeren in de praktijk leveren respectievelijk Shapley en Roth de Nobelprijs op in 2012 (bron: `www.nobelprize.org/`).

verwijderd. Hierna wordt weer van de overgebleven kanten degene met het hoogste gewicht toegevoegd aan de matching en de aanliggende kanten verwijderd, enzovoort. Het algoritme eindigt, wanneer er geen kanten meer over zijn. Het algoritme ziet er dus als volgt uit [Pre99]:

$M := \emptyset$
while $E \neq \emptyset$ **do**
  neem de kant $(a, b) \in E$ met het hoogste gewicht
  voeg (a, b) toe aan M
  verwijder alle kanten die aan a of b liggen
end

Algorithm 1: Greedy Matching

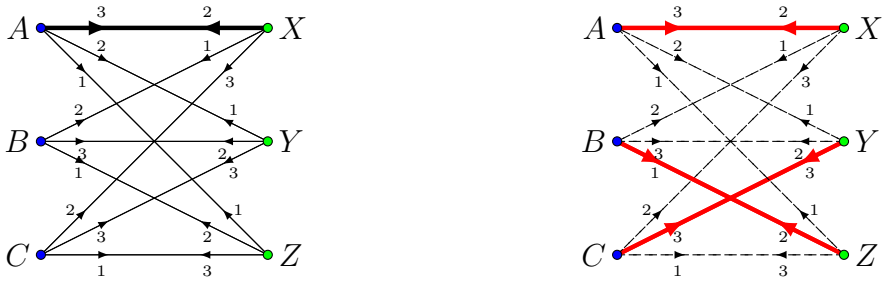
De matching die volgt is maximaal en er bestaat geen vermeerderend pad met drie of minder kanten. Een vermeerderend pad is een pad van kanten, waarbij de eerste kant niet in M zit, de tweede wel, de derde weer niet enzovoort. Het niet bestaan van een vermeerderend pad van 3 of minder kanten betekent dat de oplossing M stabiel is [Man+16]. De gesorteerde lijst van kanten doorgaan kost lineaire tijd, het sorteren echter heeft een tijdcomplexiteit van $|E| * \log |V|$. Preis heeft een algoritme gevonden dat in lineaire tijd een $\frac{1}{2}$-approximatie matching vindt, door in plaats van sorteren op zoek te gaan naar een kant met het hoogste lokale gewicht, ofwel een kant waarvan het gewicht hoger is dan van alle aanliggende kanten [Pre99]. Later kom ik hierop terug.

In veel van de toepassingen van matching, zoals de hierbovengenoemde matching van leerlingen aan scholen en donoren aan patiënten, is er echter sprake van een gerichte, bipartite graaf. Een graaf is bipartiet, als de verzameling punten V bestaat uit

twee niet lege, disjuncte deelverzamelingen V_1 en V_2, waarbij er geen kanten bestaan tussen punten uit dezelfde deelverzameling. Denk hierbij aan patiënten die graag een donornier willen en donoren die hun nier willen afgeven: er zullen geen kanten zijn van donorpatiënten naar donorpatiënten of van nierdonoren naar nierdonoren. Het vinden van een stabiele matching in een gewogen, bipartite graaf, waarbij V_1 en V_2 van gelijke grootte zijn en er van elk punt in V_1 een kant naar elk punt van V_2 loopt en andersom, wordt het Stable Marriage probleem genoemd. De naam wordt verleend aan het standaard voorbeeld van huwelijken tussen mannen en vrouwen, waarbij mannen alleen kanten naar vrouwen hebben lopen en andersom. Omdat er in de hedendaagse cultuur ook huwelijken tussen mensen van hetzelfde geslacht bestaan en er dus kanten tussen punten van dezelfde deelverzameling kunnen zijn, gebruik ik liever werknemers en banen als voorbeeld.

Zoals hierboven genoemd, beschreven Gale en Shapley als eersten dit probleem en gaven zij hun bekende algoritme om het op te lossen. Het algoritme loopt in rondes. De verzameling van n werknemers noem ik W en de verzameling van n banen B. De baan die w het liefst wil, geeft hij ranking n. De volgende baan geeft hij ranking $n-1$, enzovoort. De minst geliefde baan geeft hij ranking 1. Andersom geeft b de werknemer die het best bij de baan past ranking n, enzovoort, tot en met ranking 1. In de eerste ronde geven alle werknemers w aan welke baan b ze het liefst willen hebben door daar te solliciteren. Alleen degene die het best bij de baan past, zal hem krijgen. De andere werknemers worden afgewezen voor de baan. In de volgende ronde zullen alle afgewezen werknemers opnieuw solliciteren bij de baan die ze het liefst willen, de banen waar ze al zijn afgewezen weggelaten. Dit gaat zo door in de volgende rondes. Het niet hebben van een huidige geschikte werknemer voor een baan b, krijgt in het algoritme de ranking waarde -1, zodat elke werknemer beter is dan geen enkele werknemer ($NULL$). Gale en Shapley bewezen dat dit algoritme eindigt en dat na de laatste ronde alle werknemers aan een baan zijn toegewezen en dat deze oplossing stabiel is [Man+16].

Om de werknemers te laten solliciteren, wordt een rij Q gebruikt. Aan het begin staan alle werknemers in de rij. De eerste in de rij mag dan solliciteren bij de baan die hij het liefste wil. Wanneer de baan nog vrij is, of wanneer alle voorgaande sollicitanten minder goed bij de baan pasten, wordt de werknemer w aan de baan gekoppeld. In het algoritme geven we dit aan met $w = kandidaat(baan)$. Wanneer er een nieuwe sollicitant langskomt waarbij de baan beter past, moet de werknemer weer achteraan de rij plaatsnemen. Op deze manier zullen alle werknemers in een ronde gesolliciteerd hebben, voordat de afgewezen werknemers in de volgende ronde gaan solliciteren. In de volgende ronde moet de werknemer bij de baan solliciteren die hij daarna het liefste wil. Dit gebeurt in het algoritme met $volgendeVoorkeur(w)$. Om tijd te besparen, willen we dat de werknemer alleen gaat solliciteren bij een baan,



(a) (b)

Figuur 3: In (a) de eerste stap van het Gale-Shapley algoritme: werknemer A, die als eerste in de rij staat, solliciteert bij zijn hoogst gerankte baan X. De volgende stappen: B solliciteert bij Y, maar daarna C ook en C wint dit. B stapt terug in de rij en solliciteert bij X, maar verliest dit van A. Als laatste solliciteert B bij Z. In (b) de matching aan het eind van het algoritme.

wanneer hij meer geschikt is dan de huidige kandidaat van de baan. Dit zien we terug in de binnenste while-loop van algoritme 2 [Man+16]. Zie figuur 3. De tijdscomplexiteit van dit algoritme is $O(n^2)$, met n het aantal werknemers of banen.

```
Q = W
while Q ≠ ∅ do
    w = Q.dequeue()
    baan = volgendeVoorkeur(w)
    while ranking(baan, w) < ranking(baan, kandidaat(baan)) do
        baan = volgendeVoorkeur(w)
    end
    if kandidaat(baan) ≠ NULL then
        Q.enqueue(kandidaat(baan))
    end
    kandidaat(baan) = w
end
```

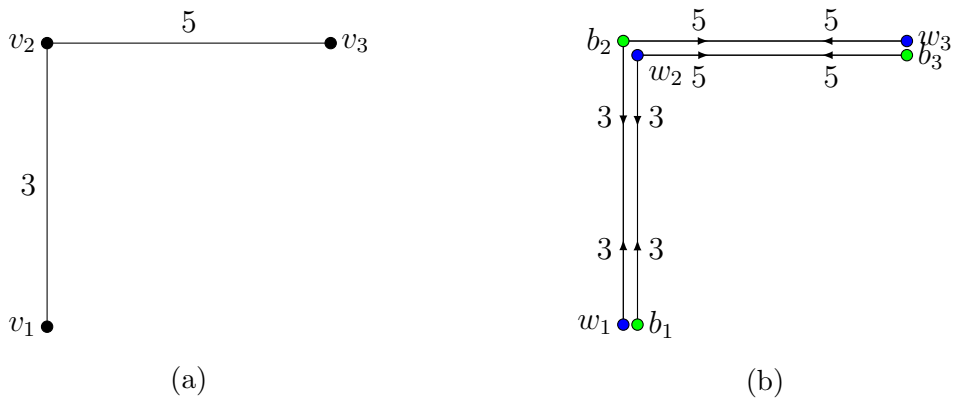
Algorithm 2: Gale-Shapley algoritme

Bij een Stable Marriage probleem met incomplete lijsten hebben de werknemers niet per se alle banen gerankt en andersom. Het Gale-Shapley algoritme geeft hiervoor ook een stabiele matching, er kunnen hierbij echter punten ongematcht blijven, omdat al de door hen gerankte punten al in de matching zitten.

2.2 De link tussen Greedy Matching en het Stable Marriage probleem

Manne en anderen hebben laten zien dat Greedy Matching een speciaal geval is van het Stable Marriage probleem [Man+16]. Om dit te laten zien, construeren we vanuit een Greedy Matching probleem op een gewogen graaf $G_{GM}(V, E)$ een Stable Marriage instantie (met incomplete lijsten) G_{SM} als volgt. Laat W en B de verzamelingen van respectievelijk $n = |V|$ werknemers en $n = |V|$ banen zijn. Voor elke kant (v_i, v_j) in E laten we werknemer i een ranking hebben voor baan j met als waarde het gewicht van de kant. Een lager gewicht betekent dus een lagere ranking. Baan j heeft een ranking voor werknemer i met dezelfde waarde. Dit is nodig, omdat de kanten in G_{GM} ongericht zijn. Op dezelfde manier hebben werknemer j en baan i ook een ranking voor elkaar met als waarde het gewicht van de kant. Elke ongerichte kant (v_i, v_j) uit G_{GM} representeert dus vier rankings in de graaf G_{SM}: die van de gerichte kanten (w_i, b_j), (b_j, w_i), (b_i, w_j) en (w_j, b_i). We noemen dit de paren (w_i, b_j) en (w_j, b_i). Zie figuur 4. We willen nu laten zien, dat gegeven een matching M_{GM} in G_{GM}, de paren van de corresponderende kanten in G_{SM} de unieke oplossing M_{SM} voor het Stable Marriage probleem op G_{SM} geven [Man+16].

Het bewijs wordt gegeven door inductie. De lijst van kanten in G_{GM} is gesorteerd op dalend gewicht. We kijken eerst naar de kant (v_i, v_j) in G_{GM} met het hoogste gewicht. Volgens het Greedy Matching algoritme is deze kant dan opgenomen in de matching, dus $(v_i, v_j) \in M_{GM}$. Dit betekent ook, dat werknemer w_i de baan b_j het hoogst heeft staan op zijn ranking, en andersom past de baan b_j ook het best bij werknemer w_i. Het paar (w_i, b_j) moet daarom in elke stabiele matching van G_{SM} zitten. Hetzelfde geldt voor het paar (w_j, b_i).

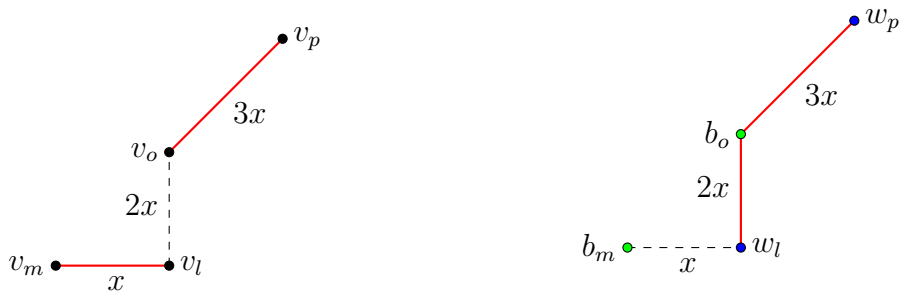


Figuur 4: In (a) een Greedy Matching instantie. In (b) de corresponderende Stable Marriage instantie. Uit het Greedy Matching algoritme volgt de matching $M = \{(v_2, v_3)\}$. De corresponderende paren (b_2, w_3) en (w_2, b_3) moeten de unieke matching van de Stable Marriage instantie vormen.

Stel nu dat de corresponderende paren van de k kanten met hoogste gewicht in M_{GM} ook in de stabiele matching M_{SM} zitten, waarbij $k \geq 1$. We kijken nu naar de kant $(v_l, v_m) \in M_{GM}$ met het $(k+1)$-na-hoogste gewicht en de corresponderende paren (w_l, b_m) en (w_m, b_l). We laten met een bewijs uit het ongerijmde zien, dat deze paren ook in de stabiele oplossing M_{SM} moeten zitten.

Stel het paar (w_l, b_m) zit niet in de stabiele oplossing M_{SM}. Stel dat w_l gematcht is aan een andere baan b_o en b_m gematcht is aan een werknemer w_n. Dat moet betekenen, dat ofwel w_l liever de baan b_o heeft dan de baan b_m, ofwel de baan b_m beter bij w_n past dan bij w_l. Als dit niet zo is, zou de matching namelijk instabiel zijn.

We bekijken het geval dat inderdaad w_l een betere ranking heeft voor b_o dan voor b_m, zie figuur 5. Dit betekent vanwege de constructie van G_{SM} dat de kant (v_l, v_o) een hoger gewicht heeft dan de kant $(v_l, v_m) \in M_{GM}$. Er geldt $(v_l, v_o) \notin M_{GM}$, omdat v_l al aan v_m gematcht is in G_{GM}. Omdat het Greedy Matching algoritme altijd de overgebleven kant met het hoogste gewicht aan de matching toevoegt, kan dat alleen, als er een kant $(v_o, v_p) \in M_{GM}$ is met nog een hoger gewicht dan (v_l, v_o). We gingen er vanuit, dat voor de k kanten in M_{GM} met het hoogste gewicht, de corresponderende paren ook in de stabiele matching M_{SM} moeten zitten. Hieruit volgt dus, dat moet gelden $(b_o, w_p) \in M_{SM}$. Dit spreekt echter tegen dat $(w_l, b_o) \in M_{SM}$. Op dezelfde manier ontstaat er een tegenspraak als b_m gematcht zou zijn aan een w_n in plaats van w_l. Hieruit concluderen we, dat het paar (w_l, b_m) in de stabiele matching M_{SM} moet zitten. Hetzelde verhaal geldt voor het paar (w_m, b_l), waaruit volgt $(w_m, b_l) \in M_{SM}$. We hebben hiermee bewezen dat ook de corresponderende paren van de kant in M_{GM} met het $(k+1)$-na-hoogste gewicht in de matching M_{SM} zitten, en dus dat voor elke kant in M_{GM} de corresponderende paren in M_{SM} zitten.



(a) Deel GM-instantie, matching in het rood.

(b) Corresponderend deel SM-instantie, matching in het rood. Kan niet!

Figuur 5: In (a) kan alleen $g(v_l, v_o) > g(v_m, v_l)$ en $(v_m, v_l) \in M_{GM}$, als (v_o, v_p) nog een hoger gewicht heeft en dus al eerder gematcht is. Vanwege de aanname in de inductiestap, moet dan in (b) $(b_o, w_p) \in M_{SM}$. Dit spreekt tegen dat $(w_l, b_o) \in M_{SM}$.

We moeten nu nog laten zien, dat wanneer alle corresponderende paren van de kanten in M_{GM} zijn toegevoegd aan M_{SM}, er geen extra paren meer bij kunnen. Stel dat dit wel zo is en dat we (w_i, b_j) kunnen toevoegen. Vanwege de constructie van G_{SM} geldt dan $(v_i, v_j) \in G_{GM}$ en dus $(v_i, v_j) \notin M_{GM}$. Zowel v_i als v_j kunnen niet gematcht zijn aan een ander punt in V, omdat in dat geval (w_i, b_j) niet in M_{SM} kan zitten, zie bovenstaand bewijs. Maar uit het Greedy Matching algoritme volgt een maximale matching, terwijl M_{GM} nog zou kunnen groeien met de kant (v_i, v_j) dus dit is een tegenspraak. Hieruit volgt dat de corresponderende paren van de matching M_{GM} inderdaad de unieke, stabiele oplossing voor het Stable Marriage probleem op G_{SM} vormen [Man+16].

Deze uitkomst geeft aan dat een oplossing voor een Greedy Matching probleem ook gevonden kan worden door het vinden van een oplossing voor het Stable Marriage probleem op de geconstrueerde graaf zoals hierboven. Het belangrijkste verschil tussen de twee problemen is, dat in het Gale-Shapley algoritme een werknemer bij een baan gaat solliciteren gebaseerd op zijn eigen ranking voor die baan, en niet op een gemeenschappelijke waarde zoals bij Greedy Matching. Preis heeft een algoritme gevonden dat een Greedy Matching probleem oplost in lineaire tijd [Pre99]. Manne en anderen concluderen dan ook met de vraag, of het mogelijk is een algoritme te ontwerpen, dat een instantie van het Stable Marriage probleem in lineaire tijd oplost [Man+16].

2.3 Het algoritme van Preis

Matching is een belangrijk probleem, dat door de enorme inputgrootte die in de praktijk kan voorkomen om een snelle oplossing vraagt. Preis heeft een $\frac{1}{2}$-approximatie-algoritme ontworpen, dat in lineaire tijd een matching van maximaal gewicht berekent van een ongerichte, gewogen graaf. Dit algoritme zal ik hieronder uitleggen en ik zal laten zien dat het inderdaad een approximatie heeft van $\frac{1}{2}$ en dat de tijdscomplexiteit lineair is.

Gegeven is een graaf $G(V, E)$ met V de verzameling punten en E de verzameling kanten (a, b) met $a, b \in V$. Het algoritme van Preis is gebaseerd op het zoeken naar kanten met het hoogste lokale gewicht: een kant (a, b) heeft het hoogste lokale gewicht, als er geen kant aanliggend aan (a, b) is met strict hoger gewicht. Hieronder is de korte versie van het algoritme te zien [Pre99].

```
M := ∅
while E ≠ ∅ do
    neem een kant (a, b) met het hoogste lokale gewicht
    voeg (a, b) toe aan M
    verwijder alle kanten die aan a of b liggen uit E
end
```

Algorithm 3: Korte versie van het algoritme van Preis [Pre99]

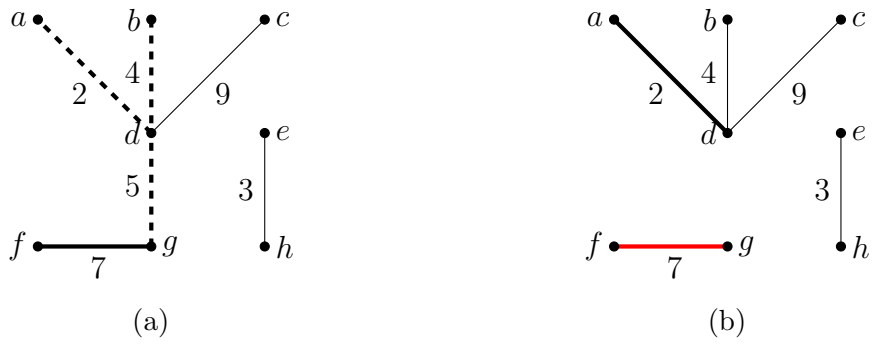
We beginnen met een lege matching M. In de while-loop wordt er gezocht naar een kant met het hoogste lokale gewicht, die wordt toegevoegd aan de matching en vervolgens worden alle aanliggende kanten verwijderd. Zolang er nog kanten over zijn, herhaalt de procedure zich. Het algoritme eindigt, wanneer er geen kanten meer zijn: alle kanten zitten dan of in de machting M, of zijn aanliggend aan een kant in M [Pre99]. Hoe vinden we een kant met het hoogste lokale gewicht? Dat is beschreven in de trymatchfunctie van het gehele algoritme van Preis, te vinden in algoritme 4. Er worden twee lokale, aan het begin lege lijsten gedefiniëerd, $C_{(a,b)}(a)$ en $C_{(a,b)}(b)$. Dit zijn de lijsten van gecheckte kanten liggend aan a respectievelijk b tijdens de huidige aanroep van trymatch. De verzameling van alle gecheckte kanten is $C := \bigcup_{(a,b)\in E} \big(C_{(a,b)}(a) \bigcup C_{(a,b)}(b)\big)$. Dan volgt er een while-loop. Hier wordt alleen doorheen gegaan wanneer zowel punt a als b vrij zijn, dus nog niet gematcht. Daarnaast moet er of een kant aanliggend aan a, of een kant aanliggend aan b ongecheckt zijn:

$$\exists(a, c) \in U \text{ OF } \exists(b, d) \in U.$$

Wanneer aan deze voorwaarden wordt voldaan, wordt er eerst voor a en daarna voor b gekeken, of het punt nog vrij is en of zo'n ongecheckte aanliggende kant bestaat. Wanneer dit zo is, wordt deze kant verplaatst naar de lokale gecheckte verzameling en wordt gecontroleerd of het gewicht van deze kant hoger is dan het gewicht van

kant (a, b). Zo ja, dan wordt de trymatchfunctie recursief aangeroepen met de kant met het hogere gewicht. De nieuwe recursie doet hetzelfde en zo kan er weer een trymatch worden aangeroepen met een kant met hoger gewicht, waardoor de recursie steeds dieper gaat en er een zoekpad ontstaat. De recursie eindigt, wanneer een kant met het hoogste lokale gewicht wordt gevonden. Deze kant wordt toegevoegd aan de matching M, de recursie gaat een stap terug in het zoekpad en gaat door met de whileloop, op zoek naar kanten met een hoger gewicht. Zo lang a en b vrij zijn en er aanliggende ongecheckte kanten zijn, wordt er zo alternerend tussen a en b gecheckt op kanten met een hoger gewicht. De while-loop loopt af, wanneer ofwel een van de punten gematcht is in een dieper recursieniveau, ofwel er geen ongecheckte aanliggende kanten meer over zijn. Hierna volgen de if-else-statements.

In het deel van de if-else-statements wordt, in het geval de punten a en b beide nog vrij zijn, de kant (a, b) toegevoegd aan de matching M en alle kanten in de lokale gecheckte verzamelingen $C_{(a,b)}(a)$ en $C_{(a,b)}(b)$ worden verplaatst naar de lijst R. Dit zijn de verwijderde kanten, waarnaar niet meer wordt gekeken in het vervolg van het algoritme. Wanneer ofwel a, ofwel b vrij blijft, omdat de ander in een dieper recursieniveau is gematcht, worden alle kanten in de lokale gecheckte lijst die aan twee vrije punten grenzen terug verplaatst van de lokale gecheckte lijsten uit C naar U. Let hierbij op: de kanten aangrenzend aan een nieuw gematchte kant worden niet gelijk verwijderd, maar in het recursieniveau waarin de aanliggende kant voor het laatst gecheckt is. Dit is belangrijk bij het bewijs van lineaire tijdscomplexiteit. Wanneer na het if-else-deel de trymatchfunctie afloopt, gaan we terug naar de while-loop van het hoofdalgoritme en zo lang er nog een kant ongecheckt is, wordt de trymatchfunctie opnieuw aangeroepen. Het algoritme eindigt, wanneer er geen ongecheckte kanten meer zijn. Dit betekent, dat er geen kanten meer zijn waarvan beide punten nog vrij zijn, zoals ik zal laten zien in het bewijs van lineaire tijdscomplexiteit. Daarnaast zijn de kanten altijd in één van de lijsten U, C of R: aan het begin zijn alle kanten in



Figuur 6: In (a): het zoekpad vanaf (a, d), via (d, g), waarbij (b, d) gecheckt is, naar (f, g). Kant (f, g) heeft lokaal het hoogste gewicht. In (b) gebeurt het volgende: $(f, g) \to M$, (d, g) wordt van C naar R verplaatst en (b, d) van C terug naar U.

U en later in het algoritme worden er alleen maar kanten verplaatst tussen U en C of van C naar R. Na de while-loop van de trymatchfunctie zijn alle lokaal gecheckte kanten of terug verplaatst naar U, of verplaatst naar R. Uit deze observatie en uit het feit dat aan het eind U leeg is, dus alle kanten in R moeten zitten, kunnen we concluderen dat M een maximale matching is: er kunnen geen extra kanten met twee vrije punten meer worden toegevoegd aan de matching [Pre99].

We zien ook dat een kant (a, b) pas in de matching komt in het laatste else-deel, wanneer hij het hoogste lokale gewicht heeft. Stel namelijk dat er een aangrenzende kant (a, c) zou zijn met hoger gewicht, waarbij c nog vrij is. Deze kant kan niet in R zitten, want dan zou a of c al gematcht zijn. Als de kant in U zou zitten, zou de while-loop niet geëindigd zijn. Als (a, c) in C zou zitten, heeft vanzelfsprekend de kant (a, b) een hoger of even hoog gewicht, anders zou (a, b) niet hoger in het zoekpad liggen [Pre99]. Deze observatie gebruiken we in het bewijs van de $\frac{1}{2}$-approximatiekwaliteit.

```
Hoofdalgoritme
M := ∅ ;                                  /* lege matching aan het begin */
U := E ;                            /* alle kanten ongecheckt aan het begin */
R := ∅ ;                           /* geen verwijderde kanten aan het begin */
while U ≠ ∅ do
    neem een willekeurige kant (a, b) ∈ U
    trymatch(a,b)
end
Functie trymatch(a, b)
C_(a,b)(a) := ∅; C_(a,b)(b) := ∅
while a is vrij EN b is vrij EN (∃(a, c) ∈ U  OF ∃(b, d) ∈ U) do
    if a is vrij EN ∃(a, c) ∈ U then
        verplaats (a, c) van U naar C_(a,b)(a) ;          /* van U naar C */
        if g(a, c) ≥ g(a, b) then
            trymatch(a, c)
        end
    end
    if b is vrij EN ∃(b, d) ∈ U then
        verplaats (b, d) van U naar C_(a,b)(b) ;          /* van U naar C */
        if g(b, d) ≥ g(a, b) then
            trymatch(b, d)
        end
    end
end
if a is gematcht EN b is gematcht then
    verplaats kanten in C_(a,b)(a) en C_(a,b)(b) naar R ;     /* van C naar R */
else if a is gematcht EN b is vrij then
    verplaats kanten in C_(a,b)(a) en {(b, d) ∈ C_(a,b)(b)|d is gematcht}
        naar R ;                                          /* van C naar R */
    verplaats kanten in {(b, d) ∈ C_(a,b)(b)|d is vrij} terug
        naar U ;                                          /* van C naar U */
else if b is gematcht EN a is vrij then
    verplaats kanten in C_(a,b)(b) en {(a, c) ∈ C_(a,b)(a)|c is gematcht}
        naar R ;                                          /* van C naar R */
    verplaats kanten in {(b, d) ∈ C_(a,b)(b)|d is vrij} terug
        naar U ;                                          /* van C naar U */
else a is vrij EN b is vrij
    verplaats kanten in C_(a,b)(a) en C_(a,b)(b) naar R ;     /* van C naar R */
    voeg (a, b) toe aan M ;                       /* nieuw gematchte kant (a, b) */
end
```

Algorithm 4: Lineaire tijd $\frac{1}{2}$-app. alg. voor matching met max. gewicht [Pre99]

Preis laat zien dat zijn algoritme een approximatiekwaliteit van een half heeft [Pre99]. Ik zal hier echter het bewijs van Bisseling voor $\frac{1}{2}$-approximatie geven [Bis17, Sectie 5.2].

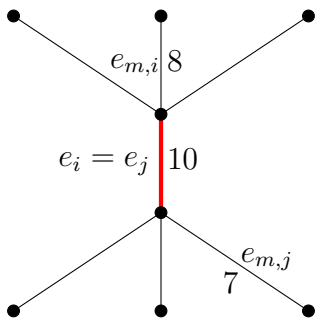
We willen laten zien, dat het gewicht van de uiteindelijke matching M uit het algoritme van Preis minimaal half zo groot is als het gewicht van de matching met maximaal gewicht. Laat M_{max} de matching met maximaal gewicht zijn met kanten $e_{max,i} \in M_{max}$ voor $i = 0, ..., k-1$. We construeren dan kanten e_i als volgt: als $e_{max,i} \in M$, dan $e_i = e_{max,i}$. Als $e_{max,i} \notin M$, dan is e_i de kant in M die ervoor zorgde dat $e_{max,i}$ niet in de matching M kwam te zitten. Deze kant e_i heeft een hoger (of even hoog) gewicht als $e_{max,i}$, want in de matching M zitten alleen kanten met een lokaal hoogste gewicht; als hij een lager gewicht had, zou het algoritme zijn voortgezet langs de kant $e_{max,i}$. Er geldt dus voor alle e_i dat $e_i \in M$. Een kant e_i kan meerdere keren voorkomen, omdat een lokaal hoogste kant meerdere kanten uit M_{max} kan hebben weggelaten. Hij kan echter niet vaker dan twee keer voorkomen: aan beide punten van de kant kan namelijk maar één kant in de matching M_{max} zitten [Bis17, Sectie 5.2]; zie figuur 7.

Door bovenstaand feit, zien we een bovengrens voor het totale gewicht van de e_i:

$$\sum_{i=0}^{k-1} w(e_i) \leq 2w(M). \tag{1}$$

Ook is er een benedengrens. Er zijn namelijk evenveel kanten e_i als $e_{max,i}$ en omdat de kant e_i altijd een hoger of evenhoog gewicht heeft als $e_{max,i}$, geldt

$$\sum_{i=0}^{k-1} w(e_{max,i}) = w(M_{max}) \leq \sum_{i=0}^{k-1} w(e_i). \tag{2}$$



Figuur 7: Kant met lokaal hoogste gewicht e_i/e_j (maximaal 2 keer) en de kanten $e_{m,i}$ en $e_{m,j}$ uit de matching van maximaal gewicht.

Als we de twee vergelijkingen samenvoegen, krijgen we

$$w(M_{max}) \leq \sum_{i=0}^{k-1} w(e_i) \leq 2w(M) \tag{3}$$

dus

$$w(M_{max}) \leq 2w(M) \tag{4}$$

en hieruit volgt

$$\frac{1}{2}w(M_{max}) \leq w(M). \tag{5}$$

Hiermee is de $\frac{1}{2}$-approximatiekwaliteit van het algoritme van Preis bewezen [Bis17, Sectie 5.2].

2.4 Bewijs lineaire tijdscomplexiteit Preis

We gaan nu kijken naar een belangrijk bewijs: gegeven een graaf $G(V, E)$ is de tijdscomplexiteit voor het berekenen van een matching M met het algoritme van Preis lineair in het aantal kanten, $O(|E|)$.

Eerst laten we zien dat de while-loop van het hoofdalgoritme maximaal $|E|$ keer wordt aangeroepen. Hiervoor moeten we inzien, dat in de lijst ongecheckte kanten U alleen kanten zitten met twee vrije punten. Aan het begin is dit zo, want nog geen enkel paar is gematcht. In de while-loop van de trymatchfunctie worden er kanten van U naar C verplaatst. Alleen in het if-else-deel van de trymatchfunctie worden kanten van C terug verplaatst naar U. Dit gebeurt echter alleen, wanneer beide punten nog vrij zijn, zie de else-if-delen. Gedurende de hele looptijd zitten er dus alleen kanten met twee vrije punten in U. Merk op dat aan het eind van elke recursie van trymatch ofwel een van de punten a en b al gematcht is in een diepere recursie, ofwel ze beide nog vrij zijn. In het laatste geval worden ze in het else-deel gematcht. Aan het eind van elke recursie van trymatch(a, b) is er dus minstens één van a en b gematcht en kan de kant (a, b) niet meer in U zitten, hij zit zelfs in R en zal niet meer terug kunnen komen in U. Tijdens elke iteratie van de while-loop van het hoofdalgoritme zal U dus met minimaal één kant afnemen die niet meer terug kan komen en hieruit volgt dat de while-loop maximaal $|E|$ keer kan worden aangeroepen [Pre99].

Ook de functie trymatch kan maximaal één keer per kant worden aangeroepen. Aan het eind van de trymatchrecursie van een kant (a, b) is namelijk minstens een van a en b gematcht, dus (a, b) zit in R en kan niet meer in het hoofdalgoritme als willekeurig element uit U worden aangeroepen, of in de while-loop van andere trymatchrecursies worden gecheckt. In totaal vinden we hierdoor een tijdscomplexiteit van

$$O(|E|) + \sum_{(a,b)\in E} O\left(\text{trymatch}(a,b)\right). \tag{6}$$

De tijdscomplexiteit van de trymatchfunctie hangt af van het aantal verplaatsingen tussen de lijsten U, C en R. Alle overige handelingen gebeuren in constante tijd. Per trymatchrecursie trymatch(a,b) zijn er verplaatsingen in de while-loop en in het if-else-deel. In de while-loop worden alleen kanten van U naar $C_{(a,b)}(a)$ en $C_{(a,b)}(b)$ verplaatst. Aan het eind van de while-loop is het aantal verplaatsingen dus gelijk aan het aantal kanten in de lokale gecheckte verzamelingen: $|C_{(a,b)}(a)| + |C_{(a,b)}(b)|$. In het if-else-deel worden alle kanten uit de lokale gecheckte verzamelingen of terug naar U, of naar R verplaatst. In totaal zijn er dus in het if-else-deel ook $|C_{(a,b)}(a)| + |C_{(a,b)}(b)|$ verplaatsingen. De tijdscomplexiteit van het hele algoritme is dus te schijven als

$$O(|E|) + \sum_{(a,b)\in E} O(|C_{(a,b)}(a)| + |C_{(a,b)}(b)|). \tag{7}$$

Merk op dat wanneer een kant verplaatst is naar R, hij niet meer uit R terug verplaatst kan worden naar C of U. Aan het eind van het algoritme zitten alle kanten in R, dus het aantal verplaatsingen naar R is maximaal $|E|$. We gaan nu laten zien, dat het aantal checkoperaties in een trymatchrecursie niet veel groter is dan het aantal verwijderoperaties (verplaatsingen naar R) [Pre99].

Laat $R_{(a,b)}$ alle kanten bevatten die tijdens de recursie trymatch(a,b) verwijderd worden, dus $\sum_{\text{trymatch}(a,b)} R_{(a,b)} = R$. We onderscheiden nu de vier gevallen in het if-else-deel van trymatch, waarbij er verplaatsingen naar R zijn.

1. a en b zijn beide gematcht:
   alle kanten in de lokale verzamelingen van a en b worden verwijderd, dus

$$|C_{(a,b)}(a)| + |C_{(a,b)}(b)| = |R_{(a,b)}|. \tag{8}$$

2. a en b zijn beide vrij:
   ook hier worden alle kanten in de lokale verzamelingen van a en b verwijderd, waarna a en b gematcht worden. Dus in dit geval geldt ook

$$|C_{(a,b)}(a)| + |C_{(a,b)}(b)| = |R_{(a,b)}|. \tag{9}$$

3. a is gematcht, b is vrij:
   dit is een moeilijker geval. We gaan hierbij gebruik maken van het feit dat in de trymatchrecursie alternerend een kant aanliggend aan a en aanliggend aan b wordt gecheckt. Stel aan het eind van de recursie is a gematcht aan c: $(a,c) \in M$ en stel dit gebeurde na k iteraties van de while-loop in trymatch.

In het if-else-deel zullen uiteindelijk alle kanten uit $C_{(a,b)}(a)$ worden verwijderd. Omdat het totaal aantal verwijderingen maximaal $|E|$ is, willen we graag de lokale lijsten $|C_{(a,b)}(a)|+|C_{(a,b)}(b)|$ afschatten op de lokale verwijderde lijst $|R_{(a,b)}|$, dus in dit geval op $|C_{(a,b)}(a)|$. Hiervoor moeten we laten zien, dat in elke iteratie van de while-loop een kant $(a,d) \in U$ met $d \neq c$ gecheckt wordt, waardoor het aantal verplaatsingen van U naar $C_{(a,b)}(a)$ aan het eind gelijk is aan k. Dit is het geval, wanneer in elke iteratie vóór de k-de iteratie de kant (a,c) ongecheckt is. Een kant zit altijd in een van de lijsten U, C of R. De kant (a,c) kan niet in R zitten, want dan zou minstens een van de punten a en c al gematcht zijn en zouden ze in de k-de iteratie niet aan elkaar gematcht kunnen worden. We moeten dus alleen nog kijken of de kant (a,c) in C zit vóór de k-de iteratie.
Stel (a,c) zit in C, dan is hij gecheckt in de trymatch(a,b)-recursie of in een eerdere recursie. Als een kant gecheckt is in een recursie, was het gewicht of niet hoger dan het gewicht van de kant van die recursie, of het zoekpad ging door via die kant. In dat laatste geval heeft uiteindelijk een matching plaatsgevonden met minstens een van de punten van die kant, of de kant ligt nog steeds in het zoekpad. De kant (a,c) kan niet gematcht zijn, want dan zou hij in R zitten. Het tweede geval zou betekenen, dat (a,b) een hoger gewicht heeft dan (a,c), want alleen dan wordt er verder gezocht via (a,b). Dit spreekt echter tegen dat in de k-de iteratie van trymatch(a,b) kant (a,c) gematcht wordt, want dan moet het gewicht van (a,c) hoger zijn dan dat van (a,b). De conclusie is dat (a,c) in U zat bij elke iteratie voor de k-de en dat er dus in de voorgaande iteraties elke keer een andere kant is gecheckt. Aan het eind van de while-loop is daarom $|C_{(a,b)}(a)| = k$ en omdat de while-loop stopt wanneer a of b gematcht is, kan $|C_{(a,b)}(b)|$ maximaal het aantal iteraties k zijn. Hieruit volgt

$$|C_{(a,b)}(a)| + |C_{(a,b)}(b)| \leq 2 * |C_{(a,b)}(a)| = 2 * |R_{(a,b)}|, \tag{10}$$

omdat alle kanten in de lokaal gecheckte verzameling van a worden verwijderd.

4. b is gematcht, a is vrij:
het bovenstaande bewijs kan ongeveer herhaald worden, maar er is een klein verschil, omdat de checkoperatie van b als tweede komt in de while-loop. Als b in de k-de iteratie wordt gematcht, is b gematcht via een nieuwe recursie van een gecheckte kant (b,d), of b is al gematcht via de checkoperatie van een (a,c) in de k-de iteratie. Voor het aantal lokaal gecheckte kanten van b geldt dus $k-1 \leq |C_{(a,b)}(b)| \leq k$, ofwel $k \leq |C_{(a,b)}(b)| + 1 \leq k+1$. Net als in vorig geval, kan het aantal lokaal gecheckte kanten van a niet meer zijn dan het aantal iteraties, dus $|C_{(a,b)}(a)| \leq k$. Hierdoor krijgen we

$$|C_{(a,b)}(a)| + |C_{(a,b)}(b)| \leq 2 * |C_{(a,b)}(b)| + 1 = 2 * |R_{(a,b)}| + 1. \tag{11}$$

Er geldt dus in totaal altijd

$$|C_{(a,b)}(a)| + |C_{(a,b)}(b)| \leq 2 * |R_{(a,b)}| + 1, \tag{12}$$

en omdat het aantal verwijderde kanten niet meer kan zijn dan $|E|$ volgt de conclusie dat de tijdscomplexiteit lineair is in het aantal kanten [Pre99]:

$$\begin{aligned} & O(|E|) + \sum_{(a,b)\in E} O(|C_{(a,b)}(a)| + |C_{(a,b)}(b)|) \\ & = O(|E|) + \sum_{(a,b)\in E} O(2 * |R_{(a,b)}| + 1) \\ & = O(|E|) + O(|E|) \\ & = O(|E|). \end{aligned} \tag{13}$$

3 Resultaten

Ik heb het algoritme van Preis geïmplementeerd in Python via de online omgeving cloud9. In bijlage A is mijn code te vinden. Ik heb het pakket networkx gebruikt voor grafen [AS08]. Hiermee kunnen grafen in verschillende soorten bestanden worden ingelezen. Daarnaast heeft het pakket verschillende functies, zoals een lijst van alle punten of kanten met commando g.nodes() of g.edges(). Het gewicht van een bepaalde kant (a, b) wordt gevonden met het commando g[a][b]['weight'].

Zoals in algoritme 4 van Preis hebben we een lege matching M aan het begin. Daarnaast is er een array MB, waarin per punt is opgeslagen of het punt gematcht is (True) of niet (False). Aan het begin zijn alle waarden 'False'. Een verschil met Preis zijn algoritme is, dat ik geen aparte array R heb gemaakt voor verwijderde kanten. Het niet terug plaatsen van een kant uit een lokaal gecheckte verzameling naar de ongecheckte kanten, is namelijk hetzelfde als het verwijderen van die kant. In beide gevallen wordt de kant nooit meer bekeken. Deze verplaatsing weglaten scheelt dus tijd.

In plaats van een gewone lijst U van ongecheckte kanten, gebruiken we een datastructuur van Linked Lists: we hebben een array LLS, met op plek x een Linked List met alle kanten aanliggend aan punt x. Hiervoor heb ik eerst alle kanten (a, b) twee keer in de array E toegevoegd, een keer als (a, b) en een keer als (b, a), zodat we de index in die array kunnen gebruiken. Voor elke kant (a, b) in E wordt er een Kant-instantie gemaakt met als waarde de index van (a, b) in lijst E. Deze wordt toegevoegd aan de Linked List van a. De volgende kant in E is de kant (b, a), die aan de Linked List van b wordt toegevoegd. Zo staat elke kant (a, b) 2 keer in de datastructuur. Hierdoor maakt het niet uit of we punt a of punt b bekijken in de trymatchfunctie; we zullen hoe dan ook de kant vinden in de Linked List. Naast

een next- en prevfunctie heeft een Kant-instantie ook een 'twin'. Dat is de hiervoor genoemde zelfde kant in de andere Linked List. Zo kan, wanneer een van de kanten wordt gecheckt en dus uit de Linked List verwijderd wordt, makkelijk de andere versie van die kant ook worden verwijderd met behulp van verwijder(twin). Ook houden we de totale lengte van de Linked Lists bij. Wanneer deze 0 is, zijn er geen ongecheckte kanten meer en moet het algoritme eindigen.

Een willekeurige kant nemen in het hoofdalgoritme gaat niet, omdat je in mijn datastructuur niet kunt itereren over de kanten. In plaats daarvan hebben we array ULL met alle punten erin. In het hoofdalgoritme wordt het eerste punt uit ULL genomen, en op de eerste kant uit de corresponderende Linked List wordt de trymatchfunctie aangeroepen. Wanneer de lengte van die Linked List echter 0 blijkt te zijn, nemen we de head van de Linked List van het volgende punt, enzovoort. De hierdoor gevonden kant wordt uit de Linked Lists van beide punten verwijderd. Deze hoeft niet in de lokale gecheckte verzamelingen te komen, want aan het eind van een trymatchrecursie is de kant altijd ofwel verwijderd ofwel gematcht. Hij komt dus nooit meer terug in de Linked Lists.

In de trymatchfunctie zien we, dat in de while-loop alternerend een aanliggende kant van a en b wordt gecheckt. Bij een checkoperatie worden zowel de gecheckte kant als zijn twin verwijderd uit de bijbehorende Linked Lists. Bij een hoger gewicht van de kant wordt een nieuwe recursie aangeroepen. In het if-else-deel worden bij verplaatsing van C naar de ongecheckte kanten beide twins weer toegevoegd aan de Linked Lists. Wanneer er een matching optreedt, worden de waarden van de punten in de array MB op 'True' gezet.

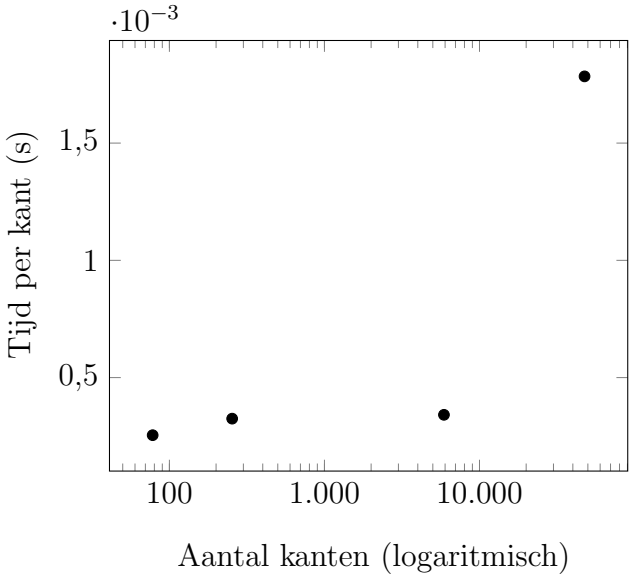
Ik heb verschillende datasets van ongerichte, gewogen grafen gebruikt om de lineairiteit te controleren. Zie hiervoor tabel 1.

Dataset	Referentie	$\|E\|$	$\|V\|$	Tijd per kant (s)
Zachary's Karate Club	[Zac77]	78	34	$0,25 * 10^{-3}$
Les Miserables	[Knu93]	254	77	$0,33 * 10^{-3}$
Contact Patterns in Primary School	[Ste+11]	5899	236	$0,33 * 10^{-3}$
Collaboration network	[New01]	47594	16726	$1,78 * 10^{-3}$

Tabel 1: Gebruikte datasets met hun aantal kanten en punten en de tijd per kant om het programma uit te voeren.

In figuur 8 is de tijd die het programma erover doet om een matching te berekenen per kant uitgezet tegen het totaal aantal kanten in de graaf.

Je ziet dat de waarde van de eerste drie punten ongeveer gelijk is. De waarde van het laatste punt, dat van de grootste dataset, is een factor 6 keer groter.



Figuur 8: De tijd per kant om een matching te vinden uitgezet tegen het totaal aantal kanten in de graaf.

4 Discussie

Voor de inplementatie van Linked Lists heb ik een werkende code geschreven met gewone lijsten. Dit programma was veel langzamer, omdat het itereren over en verwijderen uit een gewone lijst in python lineaire tijd kost. In totaal werd dit dus een kwadratische tijdscomplexiteit. Om dit te voorkomen, heb ik de Linked List datastructuur gebruikt. Hierin kost het verwijderen van een kant constante tijd. De array MB heb ik gemaakt om niet te hoeven itereren over M om te zoeken of een punt gematcht is, maar in constante tijd de waarde te kunnen opzoeken.

In figuur 8 lijkt de waarde van de eerste drie punten constant te zijn. Dit zou betekenen dat het programma een lineaire tijdcomplexiteit heeft. De waarde van het laatste punt is echter zo'n zes keer zo hoog. Een twee keer zo hoge waarde zou verklaard kunnen worden door het aantal checkoperaties in een trymatchrecursie: in het geval dat punt a of punt b in een diepere recursie is gematcht, is het aantal checkoperaties ongeveer twee keer zo groot als het aantal verwijderoperaties. Zie hiervoor het bewijs van lineariteit. In het slechtste geval zal het aantal operaties dus dubbel zo groot zijn.

Een verklaring voor het verdere verschil kan zijn dat het aantal punten in de graaf van belang is. Voor het maken van de arrays MB, ULL en LLS moet over alle punten worden geïtereerd. In de eerste, tweede en laatste graaf is het aantal punten respectievelijk de helft, een derde en de helft van het aantal kanten. In de derde graaf

is het aantal punten $\frac{1}{24}$ van het aantal kanten. Dat de waarde van de tijd per kant van de derde graaf even hoog is als die van de eerste twee grafen, kan dus misschien verklaard worden doordat het aantal punten relatief gezien een stuk kleiner is. Het is daarom mogelijk dat het programma toch niet in lineaire tijd afloopt door het ook gebruiken van het aantal punten in plaats van alleen het aantal kanten. Deze initialisatie waarbij het aantal punten wordt gebruikt, gebeurt echter maar één keer, terwijl het aantal kanten gedurende het hele algoritme van belang is. Ik betwijfel dus dat dit het geval is.

Om de afwijking van de laatste waarde verder te kunnen onderzoeken, zou het programma op meer datasets getest moeten worden. Ook zouden er nog grotere datasets moeten worden gebruikt, om te kijken of het programma daarbij lineair blijft. Dit kon echter niet door de volgende redenen: de datasets die ik kon vinden stonden niet in een bestandstype dat via networkx kan worden ingelezen of de grootte van de dataset was te groot om in cloud9 te kunnen uploaden.

5 Conclusie

Het vinden van een matching in een graaf is een probleem dat in de praktijk vaak voorkomt. Omdat de inputgrootte enorm kan zijn, is het bestaan van een algoritme hiervoor met een lineaire tijdscomplexiteit van belang. Preis heeft zo'n algoritme ontworpen voor Greedy Matching [Pre99]. Dit algoritme heb ik geïmplementeerd in python. De lineariteit van mijn programma is nog te betwisten, omdat er niet genoeg datasets voor mij bruikbaar waren. In verder onderzoek zou hiernaar moeten worden gekeken.

Manne en anderen bewezen dat er een link bestaat tussen Greedy Matching en het Stable Marriage probleem. Hun vraag voor vervolgonderzoek is dan ook, of er een lineair algoritme gevonden kan worden voor het Stable Marriage probleem [Man+16]. Als mijn programma lineair blijkt te zijn, zou zo'n algoritme naar mijn idee gevonden kunnen worden, met de datastructuur van Linked Lists, waarbij voor elk punt een Linked Lists met aanliggende kanten bestaat en waarbij elke gerichte kant (w_i, b_j) als twin de gerichte kant (b_j, w_i) heeft. Het probleem ligt hier nog wel bij het vinden van de twins. De twee gerichte kanten die elkaars twin zijn, staan namelijk niet gesorteerd in de graaf. Misschien is hier een oplossing voor te vinden. In een vervolgonderzoek zou ik hiernaar kijken.

A Algoritme van Preis geïmplementeerd in python

```
import networkx as nx
from networkx import *

#voorbeeldgraaf 'Zachary's Karate Club'
h = nx.read_graphml('gr-karate.GraphML')
g = convert_node_labels_to_integers(h)

class Kant:
    def __init__(self, waarde, tuple=None, gewicht=None, next=None, prev=None, twin=
        None):
        self.waarde  = waarde
        self.tuple  = tuple
        self.gewicht = gewicht
        self.next   = next
        self.prev   = prev
        self.twin   = twin

class LinkedList:
    def __init__(self):
        #dummywaarden:
        self.head = Kant(-1)
        self.end = Kant(-1)
        self.head.next = self.end
        self.end.prev = self.head
        self.lengte = 0

    def voegToe(self, i):
        if self.lengte == 0:
            i.prev = self.head
            i.next = self.end
            self.head.next = i
            self.end.prev = i
            self.end = i
        else:
            self.head.prev.next = i
            i.prev = self.head.prev
            self.head.prev = i
            i.next = self.head
        self.head = i
        self.lengte += 1

    def verwijder(self, kant):
        kant.next.prev = kant.prev
        kant.prev.next = kant.next
        if self.head == kant and self.end == kant:
            self.head = kant.prev
            self.end = kant.next
        elif self.head == kant and self.end != kant:
            self.head = kant.next
        elif self.head != kant and self.end == kant:
            self.end = kant.prev
        kant.prev = None
        kant.next = None
        self.lengte -= 1


#initialisatie:
#lege matching aan begin
M = []
```

```
#array om op te zoeken op een punt gematcht is; beginwaarden 'False'
MB = []
for punt in g.nodes():
    MB.append(False)

#elke kant wordt twee keer toegevoegd
E = []
for e in g.edges():
    E.append((e[0],e[1]))
    E.append((e[1],e[0]))

#voor elk punt een Linked List waar de aanliggende kanten in komen
LLS = []
i = 0
while i < len(g.nodes()):
    LL = LinkedList()
    LLS.append(LL)
    i += 1

#elke kant wordt aan de Linked List van beide punten toegevoegd en ze zijn elkaars
    twin
i = 0
while i < len(E):
    nieuwekant = Kant(i)
    nieuwekant.tuple = E[i]
    nieuwekant.gewicht = g[E[i][0]][E[i][1]]['count']
    LLS[E[i][0]].voegToe(nieuwekant)
    i += 1
    nieuwekant2 = Kant(i)
    nieuwekant2.tuple = E[i]
    nieuwekant2.gewicht = g[E[i][0]][E[i][1]]['count']
    LLS[E[i][0]].voegToe(nieuwekant2)
    i += 1
    nieuwekant.twin = nieuwekant2
    nieuwekant2.twin = nieuwekant



#bijhouden om te weten of er nog ongecheckte kanten zijn
totalelengte = 0
for i in LLS:
    totalelengte += i.lengte


#de trymatchfunctie, wordt aangeroepen in het hoofdalgoritme
def trymatch(a,b):
    global totalelengte
    #lokale arrays van gecheckte kanten
    Ca = []
    Cb = []

    #beide punten moeten vrij zijn en er moet nog een aanliggende te checken kant
        aanwezig zijn
    while MB[a] == False and MB[b] == False and (LLS[a].lengte > 0 or LLS[b].lengte >
        0):
        #alternerend tussen a en b wordt er gecheckt
        if MB[a] == False and LLS[a].lengte > 0:
            i = LLS[a].head
            #verplaatsing ongecheckte kanten naar C
            Ca.append(i)
            punti = i.tuple[1]
            twini = i.twin
            LLS[a].verwijder(i)
            LLS[punti].verwijder(twini)
```

```
            totalelengte -= 2
            #nieuwe recursie als het gewicht hoger is
            if i.gewicht > g[a][b]['count']:
                trymatch(i.tuple[0],i.tuple[1])

        if MB[b] == False and LLS[b].lengte > 0:
            j = LLS[b].head
            Cb.append(j)
            puntj = j.tuple[1]
            twinj = j.twin
            LLS[b].verwijder(j)
            LLS[puntj].verwijder(twinj)
            totalelengte -= 2
            if j.gewicht > g[a][b]['count']:
                trymatch(j.tuple[0],j.tuple[1])

    #als b gematcht, maar a vrij, moeten er kanten van C terug naar ongecheckt
    if MB[a] == False and MB[b] == True:
        for i in Ca:
            if MB[i.tuple[1]] == False:
                punti = i.tuple[1]
                twini = i.twin
                LLS[a].voegToe(i)
                LLS[punti].voegToe(twini)
                totalelengte += 2
    if MB[a] == True and MB[b] == False:
        for j in Cb:
            if MB[j.tuple[1]] == False:
                puntj = j.tuple[1]
                twinj = j.twin
                LLS[b].voegToe(j)
                LLS[puntj].voegToe(twinj)
                totalelengte += 2
    #beide punten nog vrij, dus lokaal hoogste gewicht
    if MB[a] == False and MB[b] == False:
        M.append((a,b))
        MB[a] = True
        MB[b] = True

#array om ongecheckte kanten aan te roepen in hoofdalgoritme
ULL = []
for n in g.nodes():
    ULL.append(n)
begin = ULL[0]


#hoofdalgoritme
#loopt af wanneer er geen ongecheckte kanten zijn (totalelengte = 0)
while totalelengte != 0:
    while LLS[begin].lengte == 0:
        begin += 1
    #neem een punt
    np = begin
    #neem daarvan de eerste kant in de Linked List
    nk = LLS[np].head
    twinnk = nk.twin
    a = nk.tuple[0]
    b = nk.tuple[1]
    LLS[a].verwijder(nk)
    LLS[b].verwijder(twinnk)
    totalelengte -= 2
    trymatch(a,b)
```

Referenties

[Pre99] Robert Preis. "Linear Time 1/2-Approximation Algorithm for Maximum Weighted Matching in General Graphs". In: *Meinel C., Tison S. (eds) STACS 99. STACS 1999. Lecture Notes in Computer Science, vol 1563.* Berlijn, Heidelberg: Springer, 1999, p. 259–269.

[Man+16] Fredrik Manne, Md. Naim, Hkon Lerring en Mahantesh Halappanavar. "On Stable Marriages and Greedy Matchings". In: *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing.* Albuquerque, New Mexico, VS: SIAM, 2016, p. 92–101.

[Com12] Nobel Committee. *The Prize in Economic Sciences 2012.* 2012. URL: `http://www.nobelprize.org/nobel_prizes/economic-sciences/laureates/2012/`.

[KBS16] Leonique Korlaar, Tessa Groot Beumer en Jessica Steur. *Evaluatie van het plaatsingssysteem. De centrale loting en matching van de Amsterdamse Kernprocedure 2016.* 2016. URL: `http://www.verenigingosvo.nl/wp-content/uploads/2013/09/2016.058-Evaluatie-van-het-plaatsingssysteem-centrale-loting-en-matching-Kernprocedure-2016.pdf`.

[Bis17] Rob H. Bisseling. *Parallel Scientific Computation, A Structured Approach using BSP.* 2de ed. Oxford University Press, 2017.

[AS08] Daniel A. Schult Aric A. Hagberg en Pieter J. Swart. "Exploring network structure, dynamics, and function using NetworkX". In: *7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds).* Pasadena, CA USA, aug 2008, p. 11–15.

[Zac77] W. W. Zachary. *An information flow model for conflict and fission in small groups.* Journal of Anthropological Research 33, p. 452-473. 1977.

[Knu93] D. E. Knuth. *Les Miserables.* The Stanford GraphBase: A Platform for Combinatorial Computing, Addison-Wesley, Reading, MA. 1993.

[Ste+11] Juliette Stehlé, Nicolas Voirin, Alain Barrat, Ciro Cattuto, Lorenzo Isella, Jean-François Pinton, Marco Quaggiotto, Wouter Van den Broeck, Corinne Régis, Bruno Lina en Philippe Vanhems. "High-Resolution Measurements of Face-to-Face Contact Patterns in a Primary School". In: *PLOS ONE* 6.8 (aug 2011), e23176. DOI: `10.1371/journal.pone.0023176`. URL: `http://dx.doi.org/10.1371/journal.pone.0023176`.

[New01] E. J. Newman. *The structure of scientific collaboration networks.* Proc. Natl. Acad. Sci. USA 98, p. 404-409. 2001.