Project Dynamisch Programmeren

Winnie De Ridder

25 november 2021

1 Theoretische vragen

1.1 Het algoritme

Zij $G = (V, E)$ een willekeurige gerichte acyclische graaf met n toppen en m bogen. Zij $P = (v_1, v_2, \ldots, v_{k-1}, v_k)$ een palindromisch pad van lengte k. Noteren we $s(v_i)$ voor het symbool behorend bij top v_i, dan mogen we stellen dat $s(v_j) = s(v_{k-j+1}), \forall j \in \{1, \ldots, k\}$, want anders is P geen palindromisch pad. Merk dus op dat het symbool van de middelste top in een palindromisch pad van oneven lengte niet belangrijk is om de palindromische eigenschap te behouden. Als het pad een even lengte heeft, dan moeten de beide middelste toppen hetzelfde symbool hebben.

Steunend op de hierboven beschreven eigenschappen, kunnen we volgend algoritme opstellen om de maximale palindromische paden en hun doorsnede in een dergelijke graaf G te vinden. Zij $P = (v_1, v_2, \ldots, v_{k-1}, v_k)$ een palindromisch pad van lengte k, dan definiëren we een palindromische uitbreiding van lengte $2l$ van P als een paar van paden $(\{v_{i1}, v_{i2}, \ldots, v_{il}\}, \{v_{o1}, v_{o2}, \ldots, v_{ol}\})$, waarbij de top v_{il} een uitgaande boog heeft naar top v_1 en de top v_k een uitgaande boog heeft naar de top v_{o1}, en waarvoor geldt dat $s(v_{ij}) = s(v_{o(l-j+1)}), \forall j \in \{1, \ldots, l\}$. Dit is equivalent met het feit dat als er een boog zou zijn van top v_{il} naar top v_{o1}, dan zou dit paar van paden zelf een palindromisch pad vormen. Het algoritme zal dus voor een gegeven pad P de palindromische uitbreiding zoeken waarvoor de lengte maximaal is. Om deze uitbreiding te vinden, zullen we dit pad eerst uitbreiden tot een pad P' van lengte $k + 2$ door een paar van toppen (v_0, v_{k+1}) te zoeken waarvoor geldt dat $s(v_0) = s(v_{k+1})$ en $\{(v_0, v_1), (v_k, v_{k+1})\} \subset E$. Hiervoor dienen we alle inkomende bogen van v_1 en alle uitgaande bogen van v_k te beschouwen, en telkens als de toppen die we vinden door deze bogen te volgen hetzelfde symbool hebben, dan voegen we deze toe aan het pad P om P' te bekomen. Vervolgens herhalen we deze stap voor het pad P', tot er geen dergelijk paar meer gevonden wordt, waarna we terugkeren en de andere mogelijke uitbreidingen voor P beschouwen.

Om deze recursieve berekeningen efficiënt te laten gebeuren, ligt het voor de hand om dit probleem met dynamisch programmeren aan te pakken. In de pseudocode gebruiken we een tweedimensionale tabel `paths` zodat op plaats `[i][j]` de beste uitbreiding voor een palindromisch pad opgeslagen wordt, waar $\forall i, j \in \{1, \ldots, n\}$ vooraan de top i en achteraan de top j kan toegevoegd worden. Een gelijkaardige tabel `bitvectors` wordt voorzien om op plaats `[i][j]` de doorsnede bij te houden van de verzamelingen van symbolen van toppen die liggen op alle maximale uitbreidingen die mogelijk zijn voor paden die beginnen met top i en eindigen in top j.
De pseudocode hieronder toont hoe de berekening precies in zijn werk gaat. Aangezien het algoritme bestaande paden verder uitbreidt, bepalen we in de eerste lus paden van lengte 1, waarbij het symbool van deze top niet belangrijk is, of 2, waarbij deze toppen hetzelfde symbool moeten hebben om palindromisch te kunnen zijn. Vervolgens roepen we `ExtendPath` op om dit pad verder uit te breiden. Voor elk paar van toppen dat een mogelijk begin vormt voor een palindromische uitbreiding voor het huidige pad, bepalen we recursief de maximale uitbreiding. Als deze uitbreiding langer is dan de huidige uitbreiding van dit pad, wordt deze ingesteld als beste uitbreiding en wordt de bijhorende bitvector eveneens bijgehouden. Als de nieuwe uitbreiding even lang is als de huidige, dan wordt de doorsnede van de bitvectoren bepaald. Nadat alle mogelijke uitbreidingen bepaald werden, worden de symbolen van de toppen waarvoor we de maximale uitbreiding zochten toegevoegd aan de bitvector van dit pad en vervolgens worden deze waarden opgeslagen in de juiste tabellen. Om een lus op het einde te voorkomen, worden de maximale waarden eveneens bijgehouden, om deze daarna terug te kunnen geven.

Algorithm 1 Bepaal maximale palindromische paden in een DAG

Input: Gerichte, acyclische graaf $G = (V, E)$ met n toppen en m bogen, V kan benaderd worden als een array.
Output: Lengte van maximaal palindromisch pad, verzameling van symbolen die voorkomen op alle maximale palindromische paden in G en één dergelijk pad, als het bestaat.

```
function ComputeLongestPalindromicPaths
    bestPath ← ∅
    bestVector ← ∅
    for start ← 1 to n do
        ExtendPath(start, start)
        for all neighbors of V[start] do
            if s(V[start]) = s(V[neighbor]) then
                ExtendPath(start, neighbor)
            end if
        end for
    end for
    return length(bestPath), toString(bestVector), bestPath
end function

function ExtendPath(start, end)
    if paths[start][end] != null then
        return paths[start][end]
    end if
    path ← Path(start, end)
    pathVector ← 0
    for all incoming edges E_start of V[start] do
        for all outgoing edges E_end of V[end] do
            previous ← E_start.source
            next ← E_end.destination
            if s(previous) = s(next) then
                extension ← ExtendPath(previous, next)
                if length(path.getNext()) < length(extension) then
                    path.setNext(extension)
                    pathVector ← bitvectors[previous][next]
                else if length(path.getNext()) = length(extension) then
                    pathVector ← pathVector & bitvectors[previous][next]
                end if
            end if
        end for
    end for
    paths[start][end] ← path
    bitvectors[start][end] ← pathVector || s(V[start]) || s(V[end])
    if length(bestPath) < length(path) then
        bestPath ← path
        bestVector ← bitvectors[start][end]
    else if length(bestPath) = length(path) then
        bestVector ← bestVector & bitvectors[start][end]
    end if
    return path
end function
```

1.2 Correctheid voor gerichte acyclische grafen

Bewijs: We bewijzen de correctheid van het algoritme via *sterke inductie*. We zullen aantonen dat het algoritme alle palindromische paden in een willekeurige gerichte acyclische graaf $G = (V, E)$ kan vinden, en bijgevolg ook alle maximale palindromische paden en hun doorsnede.

Inductiebasis: Het algoritme bepaalt op triviale manier alle paden van lengte 1 die per definitie ook palindromisch zijn. De bijhorende doorsnede is ofwel één symbool, ofwel leeg. Dit gebeurt in de eerste lus waarbij elke top gebruikt wordt als middelpunt van een eventueel groter pad, door `ExtendPath` op te roepen met hetzelfde begin- en eindpunt. Als er geen palindromische uitbreiding mogelijk was, dan zal het pad bestaande uit deze ene top ongewijzigd blijven. Op het einde van deze oproep wordt de doorsnede bepaald door het symbool van deze top toe te voegen aan de bijhorende verzameling, waardoor deze enkel dat symbool zal bevatten. Ook alle palindromische paden van lengte 2, die bestaan uit precies twee toppen met hetzelfde symbool, worden triviaal bepaald tijdens die lus, waarbij elke top samen met een buur met hetzelfde symbool gebruikt wordt als startpunt voor een palindromisch pad. Als ook voor deze paden geen palindromische uitbreiding gevonden kan worden, zal de bijhorende doorsnede opnieuw slechts één symbool bevatten, aangezien beide toppen hetzelfde symbool hebben. We kunnen hieruit besluiten dat alle palindromische paden van lengtes 1 en 2 en hun doorsnedes correct bepaald worden.

Inductiehypothese: We veronderstellen dat het algoritme voor alle lengtes $l < k$ alle palindromische paden van deze lengtes en hun doorsnedes kan vinden.

Inductiestap: Stel dat G een palindromisch pad P van lengte $k > 2$ bevat. In de eerste lus zullen alle mogelijke middelpunten van paden overlopen worden. Als k oneven is, zal dit midden een pad van lengte 1 zijn bestaande uit één top, anders zal het pad een paar van toppen zijn met hetzelfde symbool en dus lengte 2 hebben, noem deze lengte m. Als we nu de functie `ExtendPath` oproepen met dit midden als beginpunt, dan zal het alle mogelijke uitbreidingen bepalen van dit pad, door alle inkomende bogen van de linkertop en de uitgaande bogen van de rechtertop te overlopen. Voor elk toppenpaar (v_i, v_o), dat bekomen wordt door de eindpunten van die bogen te bepalen, waarvoor $s(v_i) = s(v_o)$, wordt recursief de langste palindromische uitbreiding bepaald. Een dergelijke uitbreiding is, zoals beschreven in Sectie 1.1, een paar van paden, zodat als er een boog was die beide delen met elkaar verbond, het gevormde pad palindromisch zou zijn. De functie controleert niet of de gegeven toppen wel degelijk met elkaar verbonden zijn, zodat het bepalen van een uitbreiding equivalent is met het bepalen van een pad. De uitbreiding van het midden is bijgevolg equivalent met het bepalen van een palindromisch pad met als midden (v_i, v_o) van lengte $k - m < k$ zodat we de inductiehypothese kunnen gebruiken. De uitbreiding wordt dus correct bepaald, en daarna ingesteld als uitbreiding van het midden, zodat het pad P bekomen wordt.

Aangezien we nu bewezen hebben dat een willekeurig pad P van lengte k correct bepaald wordt, rest ons nog de correctheid van de doorsnedes van de verzamelingen van symbolen die voorkomen op paden van deze lengte te bewijzen. Stel dat er voor het midden van het pad P nog een andere uitbreiding bestaat met dezelfde lengte $k - m$. Deze uitbreiding wordt gevonden, en wegens de inductiehypothese is ook de bijhorende doorsnede correct. Zoals te zien op lijn 30 in de pseudocode, wordt dan de doorsnede genomen van deze doorsnedes. Na de dubbele lus worden ten slotte de symbolen van de middelste toppen toegevoegd. Deze symbolen komen immers voor op dit pad. Bijgevolg wordt deze verzameling van symbolen correct berekend. Stel nu dat ook de lengte van de maximale palindromische paden in deze graaf gelijk is aan k. De eerste keer dat een dergelijk pad gevonden werd, wordt het ingesteld als langste palindromische pad en wordt de bijhorende bitvector opgeslagen. Als er tijdens de verdere uitvoering een ander pad met dezelfde lengte gevonden wordt, dan wordt opnieuw de doorsnede genomen. Aangezien deze doorsnedes zoals bewezen correct zijn, wordt ook de doorsnede van alle maximale palindromische paden correct bepaald. □

1.3 Complexiteit voor gerichte acyclische grafen

Om de doorsnede van alle maximale palindromische paden te bepalen, moet eerst de input ingelezen en verwerkt worden. Deze stap heeft reeds kost $O(n^2)$, aangezien we eerst het aantal toppen n inlezen, en vervolgens n keren 2 lijnen moeten verwerken.

Het verwerken van de eerste lijn in een dergelijk paar lijnen heeft een constante kost. De tweede lijn, die alle buren van deze top bevat, heeft als kost $O(n)$, aangezien een top hoogstens $n-1$ buren kan hebben, en de kost voor het verwerken van één buur is constant. Het initialiseren van de tabellen om dynamisch programmeren mogelijk te maken, heeft kost $\Theta(n^2)$, dus de totale kost van het inlezen is $\Theta(n^2)$. Aangezien de echte berekening van de paden een hogere kost heeft, heeft het inlezen geen grote invloed op de totale complexiteit van het algoritme.

Het berekenen van de palindromische paden verloopt zoals beschreven in de pseudocode uit Sectie 1.1. Aangezien het algoritme werkt met dynamisch programmeren, is een bovengrens voor de complexiteit in functie van het aantal toppen het aantal deeloplossingen vermenigvuldigd met de kost om een dergelijke tussenoplossing te bepalen. Er zijn precies $\Theta(n^2)$ plaatsen in de tabel voorzien. Een bovengrens voor de kost van het bepalen van één tussenoplossing wordt gegeven door $O(n^2)$, aangezien we per inputpaar alle inkomende bogen van de starttop combineren met alle uitgaande bogen van de eindtop. Beide toppen hebben echter $O(n)$ bogen, zodat we dus $O(n^2)$ tussenoplossingen beschouwen. Alle verdere operaties bij deze berekeningen, zoals testen op condities, waarden opzoeken in tabellen of de doorsnede nemen van twee bitvectoren, hebben allemaal kost $O(1)$, zodat de totale kost om een tussenoplossing te bepalen gegeven wordt door $O(n^2)$. Bijgevolg is een bovengrens voor de complexiteit van het algoritme in functie van het aantal toppen gelijk aan $O(n^2) \cdot O(n^2) = O(n^4)$.

Het bepalen van de complexiteit in functie van het aantal bogen m verloopt op een analoge manier. Bij het verwerken van de input van de graaf wordt elke boog tweemaal beschouwd, aangezien zowel de begintop als de eindtop van een boog deze verbinding bijhoudt. De kost hiervan is dus $\Theta(m)$. In de eerste lus bouwen we startpaden van lengte 1 of 2 op. Om dergelijke paden van lengte 2 te vinden, moeten we alle bogen van de huidige top bekijken. Aangezien we dit voor elke top uitvoeren, bekijken we in deze lus precies $\Theta(m)$ bogen. Als we nu opnieuw het slechtste geval bekijken waarbij elke top hetzelfde symbool heeft, dan zal elke boog in de graaf verschillende keren bekeken worden. De functie om bestaande paden uit te breiden kan hoogstens op n^2 verschillende manieren aangeroepen worden. Herhaaldelijke aanroepen hebben kost $\Theta(1)$, aangezien de waarde dan meteen uit de tabel gehaald kan worden. Als de waarde nog niet eerder berekend werd, zal deze functie alle inkomende bogen van het eerste argument en alle uitgaande bogen van het tweede argument beschouwen, en als de symbolen van de toppen die gevonden werden door deze bogen te volgen overeenkomen, dan zal de functie zichzelf recursief oproepen. Als de functie dus effectief op alle n^2 verschillende manieren aangeroepen wordt, dan zullen alle toppen eens als begintop en eens als eindtop gebruikt worden. Bijgevolg zal elke boog van de graaf gecombineerd worden met elke andere boog in de graaf, wat leidt tot een totale kost van $O(m^2)$. Aangezien een bovengrens voor het aantal bogen in een graaf gegeven wordt door $O(n^2)$, komt deze complexiteit overeen met de eerder bepaalde complexiteit in functie van het aantal bogen.

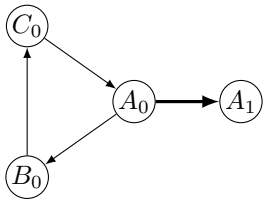
1.4 Bijzondere gevallen

(a) Een acyclische graaf die palindromisch onbegrensd is

Dit is onmogelijk.

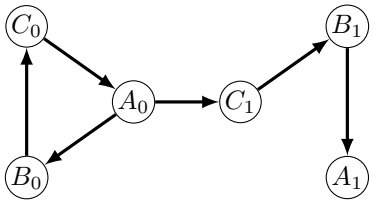
Bewijs: Zij $G = (V, E)$ een gerichte acyclische graaf met n toppen en m bogen. Opdat G palindromisch onbegrensd zou zijn, moet er voor elk natuurlijk getal k een palindromisch pad van lengte $l > k$ bestaan. In een acyclische graaf is de lengte van het maximale palindromische pad echter hoogstens gelijk aan n. Stel immers dat er een palindromisch pad van lengte $p > n$ zou bestaan, namelijk $P = (v_1, v_2, \ldots, v_n, \ldots, v_p)$. Dit pad bevat meer dan n toppen, wegens het duivenhokprincipe mogen we stellen dat er minstens 1 top v_i in P minstens tweemaal voorkomt. Laat deze top voorkomen op indices i en j in P, met $1 \leq i < j \leq p$. Beschouw het pad $P_{(i,j)} = (v_i, \ldots, v_i)$. Dit is een deelpad van P dat begint en eindigt in dezelfde top v_i. Dit is echter de definitie van een cykel, wat betekent dat het pad P een cykel bevat, in tegenspraak met de veronderstelling dat de graaf G acyclisch is. Bijgevolg kan een acyclische graaf nooit palindromisch onbegrensd zijn. □

(b) Een gerichte graaf met cykel die palindromisch begrensd is



Deze graaf bevat 1 cykel, namelijk (A_0, B_0, C_0, A_0), en het maximale palindromische pad, gegeven door (A_0, A_1), heeft lengte 2.

(c) Een maximaal palindromisch pad dat een cykel bevat

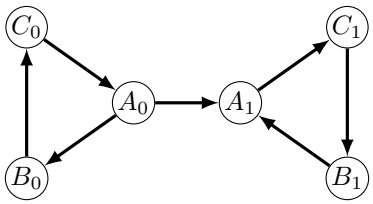


Deze graaf bevat 1 cykel, namelijk (A_0, B_0, C_0, A_0), en het maximale palindromische pad, gegeven door

$$(A_0, B_0, C_0, A_0, C_1, B_1, A_1),$$

met lengte 7 bevat deze cykel.

(d) Een palindromisch onbegrensde graaf zonder palindromische cykels



Deze graaf bevat 2 cykels, namelijk (A_0, B_0, C_0, A_0) en (A_1, C_1, B_1, A_1). Beide cykels zijn niet palindromisch, maar we kunnen voor elke lengte $k \in \mathbb{N}$ een pad van lengte $n > k$ bekomen door k keer beide cykels te doorlopen en vervolgens met elkaar te verbinden. Een voorbeeld voor $k = 2$ is het pad

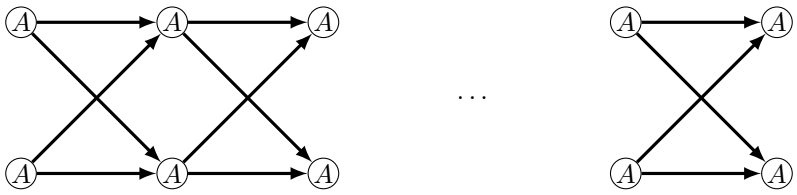
$$(A_0, B_0, C_0, A_0, B_0, C_0, A_0, A_1, C_1, B_1, A_1, C_1, B_1, A_1).$$

1.5 Is het maximale aantal maximaal palindromische paden exponentieel?

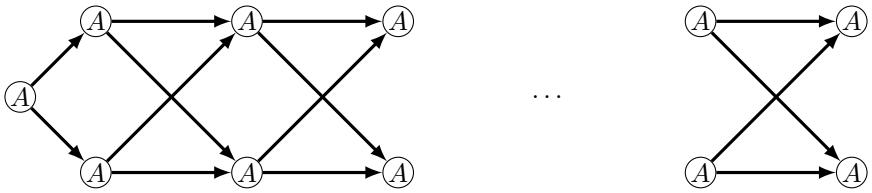
Dit is het geval, en we zullen nu voor alle $n \in \mathbb{N}\backslash\{0\}$ een graaf construeren met n toppen zodat het maximaal aantal maximale palindromische paden in deze graaf $\Omega\left(2^{\lfloor \frac{n}{2} \rfloor}\right)$ is. We veronderstellen dat alle toppen hetzelfde symbool hebben. Dit is niet noodzakelijk, maar maakt het bewijs eenvoudiger. Stel dat n oneven is, dan kiezen we één willekeurige top die het beginpunt van elk maximaal palindromisch pad zal zijn. We plaatsen deze top helemaal links en beschouwen nu enkel nog de andere toppen in de graaf. Uit deze toppen kiezen we vervolgens eerst twee toppen die geen bogen zullen hebben. Vervolgens verdelen we alle toppen in twee rijen, zodat de ene rij boven de andere staat, zoals getoond in de voorbeeldgraaf hieronder.

Als we nu de toppen van links naar rechts overlopen, dan krijgt elke top in de bovenste rij een boog naar zijn rechterbuur en een boog naar zijn rechteronderbuur. Analoog krijgen de toppen op de onderste rij een boog naar hun rechterbuur en een top naar hun rechterbovenbuur. Bij oneven n krijgt de eerder gekozen top een boog naar elke begintop van de zojuist gevormde rijen. Op het einde van de rijen plaatsen we de gekozen toppen zonder bogen. Elke top heeft dus ofwel twee, ofwel geen bogen. Nu beschouwen we de maximale palindromische paden. Als we opnieuw de toppen van links naar rechts overlopen, hebben we één of twee toppen, noem dit aantal b, die een mogelijk beginpunt zijn voor deze paden, en telkens hebben we twee keuzes om een dergelijk pad uit te breiden, afhankelijk van in welke rij de top zich bevindt. Na $\lfloor \frac{n-1}{2} \rfloor$ keer een dergelijke uitbreiding te doen, bereiken we de toppen zonder bogen zodat het pad eindigt in deze toppen. We bekomen dus dat we $\lfloor \frac{n-1}{2} \rfloor$ keer 2 keuzes moeten maken en b keuzes hebben voor de begintop van de paden. Bijgevolg vinden we in deze graaf $b \cdot 2^{\lfloor \frac{n-1}{2} \rfloor}$ paden van lengte $\lfloor \frac{n-1}{2} \rfloor + 1$, of dus $\Theta(2^n)$ maximale palindromische paden, en dit aantal is exponentieel in functie van het aantal toppen, wat te bewijzen was. □

Een voorbeeldgraaf voor een zekere even n volgt hieronder. Voor de eenvoud werd het symbool 'A' gekozen als symbool voor elke top, maar andere symbooltoekenningen zijn mogelijk, zolang de palindromische eigenschap van het pad niet verstoord wordt.



Indien n oneven is, dan wordt een willekeurige top vooraan geplaatst. Hier dient men wel meer aandacht te besteden aan de symbolen op de toppen, maar in dit voorbeeld met slechts 1 mogelijk symbool is een willekeurige keuze toegelaten.



1.6 Polynomiale bovengrens voor de lengte van een maximaal palindromisch pad in functie van het aantal toppen

In een gerichte, acyclische graaf is de lengte van een maximaal palindromisch pad $O(n)$ zoals bewezen in Bijzonder geval (a) van Sectie 1.4. In een algemene gerichte graaf is de maximale lengte van een palindromische pad $O(n^2)$.

Bewijs. We kunnen een palindromisch pad voorstellen als paren van toppen. Zij $G = (V, E)$ een algemene, gerichte graaf met n toppen en m bogen en zij $P = (v_1, v_2, \ldots, v_{k-1}, v_k)$ een palindromisch pad van lengte k in G. Dan zijn de paren van toppen (v_j, v_{k-j}), $\forall j \in \{1, \ldots, k\}$, en geldt dus $s(v_j) = s(v_{k-j})$. Stel nu dat de bogen (v_i, v_1), $(v_k, v_e) \in E$, waarvoor geldt dat $s(v_i) = s(v_e)$, $(i, e) \in \{1, \ldots, n\}$. Dan kunnen we het pad P uitbreiden tot een pad P' van lengte $k + 2$ door het paar (v_i, v_e) toe te voegen aan P, zodat v_i vooraan komt te staan, en v_e achteraan. Echter, als dit paar reeds eerder voorkwam in P, zegge voor $i \in \{1, \ldots, \frac{k}{2}\}$ en $e = k - i$, dan is de graaf G palindromisch onbegrensd. We kunnen dan immers het pad P blijven uitbreiden door de toppen $\{v_i, v_1, \ldots, v_i\}$ vooraan en de toppen $\{v_e, v_{e+1}, \ldots, v_k, v_e\}$ achteraan toe te voegen, en zo kunnen we $\forall n \in \mathbb{N}$ een pad van lengte $l > n$ construeren door deze verzamelingen n keren toe te voegen aan P. Bijgevolg is de maximale lengte van een maximaal palindromisch pad $O(n^2)$, aangezien dit precies het aantal unieke paren (a, b) is, waarbij $a, b \in \{1, \ldots, n\}$. □

1.7 Correctheid voor algemene gerichte grafen

Er zijn weinig aanpassingen nodig aan het algoritme om ook algemene, gerichte grafen te kunnen verwerken. Het idee is nog steeds om bestaande paden verder uit te breiden, maar er is wel een manier nodig om te voorkomen dat oneindige paden berekend worden. Steunend op het bewijs in Sectie 1.6 zien we in dat we dus enkel unieke paren van toppen mogen toevoegen aan een bestaand pad. Als een zeker paar eerder werd toegevoegd, is de graaf palindromisch onbegrensd en mogen we dit als resultaat teruggeven. Als we dus een zeker pad P uitbreiden, dan houden we in een tabel de paren van indices van toppen bij die deel uitmaken van dit pad. Als het een nieuw paar is, slaan we dit op, anders gooien we een uitzondering op die weergeeft dat de graaf palindromisch onbegrensd is. Elke verdere, recursieve uitbreiding zal dit herhalen tot een (maximaal) palindromisch pad gevonden wordt en zal vervolgens dit paar weer als ongebruikt aanduiden in de tabel, tenzij een palindromische cykel ontdekt wordt, dan worden alle verdere berekeningen stopgezet.

1.8 Complexiteit voor algemene gerichte grafen

Er is zeer weinig veranderd aan de implementatie van het algoritme. Om de onbegrensdheid van de graaf te controleren, wordt een tabel met booleaanse waarden bijgehouden. Het opzoeken en aanpassen van waarden hierin kan in constante tijd gebeuren. De benodigde hoeveelheid geheugen is wel met $\Theta(n^2)$ toegenomen, maar er waren reeds reeds tabellen van deze grootte aanwezig, dus de vereiste hoeveelheid geheugen blijft ook $\Theta(n^2)$. De totale complexiteit van het algoritme blijft dus zoals eerder bewezen $O(n^4)$ in functie van het aantal toppen en $O(m^2)$ in functie van het aantal bogen.

2 Beschrijving van de implementaties en optimalisaties

2.1 Beschrijving van implementaties

Mijn eerste implementatie van dit algoritme maakte veelvuldig gebruik van lijsten van `Strings` om alle paden op een leesbare manier op te slaan. Dit leidde tot een eenvoudige berekening van de doorsnedes door alle karakters in de `Strings` te doorlopen en toe te voegen aan een bitvector om daarna de doorsnedes van deze bitvectoren te nemen, maar was op vlak van geheugengebruik bijzonder inefficiënt. Een voordeel was wel dat testen eenvoudiger werd, aangezien ik toegang had tot alle berekende paden en zo fouten kon opsporen. Nadat ik zeker was van de correctheid, heb ik naar manieren gezocht om minder geheugen nodig te hebben, want grafen met meer dan 200 toppen kostten te veel tijd en geheugen om te verwerken.

Een eerste verbetering was om over te stappen op lijsten van `Integers` om minder geheugen te gebruiken, maar ik hield nog steeds alle paden bij in deze vorm. Vervolgens heb ik het dan verder aangepast om slechts één lijst van `Integers` nodig te hebben per positie in de tabel, namelijk de beste uitbreiding voor een pad met begintop i en eindtop j. Dit had wel als gevolg dat een aparte tabel nodig was om de bitvectoren op te slaan, aangezien ik ze niet meer op het einde kon bepalen uit de lijst met alle maximale palindromische paden. Echter, aangezien `longs` niet veel geheugenruimte innemen en de operaties erop heel efficiënt zijn, was dit een goede keuze om het gewenste resultaat te bekomen.

Lijsten zorgden echter voor het veelvuldig opslaan van dezelfde waarden, aangezien telkens een kopie genomen werd van de hele lijst en er vervolgens twee elementen aan toegevoegd werden. Daarom besloot ik een klasse `Path` aan te maken die een eenvoudige verwijzing bijhoudt naar het volgende deel van het pad. Aangezien er slechts één volledig pad teruggegeven moet worden, is dit veel geheugenefficiënter. Een extra methode die een dergelijk pad omzet naar de gezochte oplijsting van de indices zorgt voor de juiste weergave. De methode `toString` is hiervoor uiteraard ideaal.

In totaal heb ik twee verschillende algoritmen geïmplementeerd en vergeleken. Het eerste algoritme volgde de "top-down-approach", waarbij het startpunt en het eindpunt van een mogelijk palindromisch pad bepaald werden en vervolgens geprobeerd werd om deze met elkaar te verbinden op de langst mogelijke manier. Dit algoritme had echter een belangrijk nadeel: het berekende te veel nutteloze waarden, aangezien niet elk paar toppen met hetzelfde symbool met elkaar verbonden is. Dit maakte ook het detecteren van onbegrensde palindromische paden moeilijker, aangezien je niet met zekerheid kon weten of je wel degelijk een bestaand pad aan het berekenen was.

Het tweede algoritme, dat een "bottom-up-approach" volgt, garandeert dat je op een bestaand pad verder aan het opbouwen bent. Bijgevolg kunnen onbegrensde palindromische paden eenvoudiger gedetecteerd worden. Als immers bij het uitbreiden van een pad een toppenpaar bereikt wordt dat reeds voorkomt in het pad, dan moet het huidige pad wel een onbegrensd palindromisch pad zijn. De snelheidswinst is ook groot: het eerste algoritme heeft bijna een minuut nodig om een willekeurige graaf met 400 toppen te verwerken, terwijl dit algoritme de klus kan klaren in minder dan 5 seconden.

Om de toegang tot de bogen van de graaf en de symbolen van de toppen mogelijk te maken, heb ik een klasse `Node` geïmplementeerd. Objecten van deze klasse beschikken over de volgende velden: een symbool, een index, een collectie van toppen die een uitgaande boog naar deze top hebben en een collectie van toppen waarnaar deze top een uitgaande boog heeft. Dit zorgt voor een efficiënte en leesbare toegang tot alle benodigde data opdat het algoritme het gevraagde zou kunnen berekenen. De collecties zorgen ervoor dat ook andere implementaties voor deze datastructuur kunnen gebruikt worden. Zo bleek de `ArrayDeque`-implementatie iets performanter dan de `ArrayList`-implementatie, maar het verschil bleef beperkt tot enkele honderden milliseconden voor relatief grote tests. Elders in de code worden deze toppen geabstraheerd tot hun indices in de `Node[]`, zodat deze ook gebruikt kunnen worden om waarden in de tabellen op te zoeken.

De klasse `Path` zorgt voor een redelijk geheugenefficiënte manier om paden voor te stellen, door twee indices en een verwijzing naar een palindromische uitbreiding van dit pad bij te houden. Deze uitbreidingen hebben dezelfde functionaliteit als echte paden, maar dienen niet apart gebruikt te worden.

Dankzij de eerste lus, die de beginpaden voorziet, zullen deze uitbreidingen altijd deel uitmaken van één groot pad, zodat het algoritme altijd enkel nuttige berekeningen maakt.

De klasse `CharBitVector` abstraheert de omzetting van symbolen naar binaire posities en omgekeerd. Hier worden echter geen klasse-objecten van gebruikt omdat dit de geheugenefficiëntie van het gebruik van `longs` zou teniet doen. Zo kunnen ook de eenvoudige binaire operatoren rechtstreeks toegepast worden om bijvoorbeeld de doorsnede van twee verzamelingen te berekenen. De klasse voorziet echter wel statische methoden om de verwerking van symbolen leesbaar te houden.

Ten slotte zorgt de abstracte klasse `PalindromicPathFinder` ervoor dat de code-duplicatie beperkt blijft. De klasse `DG` verschilt immers bijzonder weinig van de klasse `DAG`, dus gemeenschappelijke functionaliteit kan hierin perfect gebundeld worden. Dit zorgt er ook voor dat de testklassen `DGTests` en `DAGTests` eenvoudiger geïmplementeerd kunnen worden door opnieuw overerving toe te passen. Deze klassen zorgen ervoor dat de abstracte testklasse `PalindromicPathTests` correct geïnitialiseerd wordt, zodat dezelfde tests probleemloos voor beide algoritmen uitgevoerd kunnen worden. De `DAGTests`-klasse voorziet ook aparte testgevallen voor cyclische grafen.

2.2 Experimenten en resultaten

Voor ik aan de implementatie begon, had ik reeds enkele testgevallen voorbereid, zodat ik snel kon ingrijpen als het fout liep. Deze gevallen zijn te vinden in de directory *test*, meer bepaald de gevallen waarbij de grafen in `String`-vorm zijn geschreven. Hierin bevindt zich de graaf uit de opgave van het project en enkele grafen die ik zelf bedacht heb. Na deze fase heb ik een algoritme geschreven dat een willekeurige gerichte acyclische graaf in `String`-vorm teruggaf, die voldoet aan enkele parameters zoals het aantal toppen en het aantal verschillende symbolen op de toppen. De afwezigheid van cykels wordt gegarandeerd door een top enkel uitgaande bogen te geven naar toppen met een grotere index, wat er wel voor zorgt dat de laatste toppen weinig uitgaande bogen hebben. Daarna heb ik eerst verder de correctheid geverifieerd door relatief kleine grafen te genereren en manueel na te gaan of de juiste oplossing gevonden werd. Hierna ben ik begonnen met het testen van de geheugenefficiëntie van het algoritme door relatief grote grafen te genereren en deze als input te gebruiken. Daardoor stelde ik vast dat de implementatie met slechts één lijst van `Integers` per tabelpositie beduidend minder geheugenruimte gebruikte, aangezien er geen `OutOfMemoryException` werd opgegooid, zelfs bij grafen met 400 toppen of meer, in tegenstelling tot de eerste implementatie. Bij de finale implementatie met de `Path` -klasse en de "bottom-up-approach" kon zelfs de voorbeeldgraaf met 5000 toppen zonder geheugenproblemen verwerkt worden in een relatief korte tijd. Echter, als er slechts 512 Mb geheugenruimte toegekend werd aan het programma, dan werd er nog steeds een `OutOfMemoryException`opgegooid. Bij een toekenning van 700 Mb lukte dit wel, en om geen afbreuk te doen aan de leesbaarheid van de code, vond ik dit resultaat goed genoeg.

Om de efficiëntie te meten, gebruikte ik mijn willekeurige-grafengenerator om tien grafen van lengte 100 met één mogelijk symbool aan te maken, aangezien dit het slechtste mogelijke geval is en de benodigde verwerkingstijd de eerder bekomen bovengrens het beste zou benaderen. Deze liet ik dan verwerken door mijn algoritme; de benodigde tijd hiervoor was 637 ms. Als ik daarna tien grafen van lengte 200, opnieuw met één mogelijk symbool, liet verwerken, duurde dit 3 s 542 ms. Bij het verdubbelen van de inputgrootte heeft het algoritme ongeveer 5.5 keer zoveel tijd nodig. Echter, dit is een te kleine inputgrootte om al conclusies uit te trekken. Om tien willekeurige grafen met 400 toppen te verwerken, duurde het 58 s 74 ms, of dus een verhouding van 16.4 vergeleken met de tijd voor de grafen met 200 toppen.

Om de correctheid van het algoritme voor cyclische grafen na te gaan heb ik een algoritme geschreven dat cyclische grafen in de gewenste vorm genereert. Hiervoor heb ik enkele kleine aanpassingen gedaan aan het algoritme dat acyclische grafen aanmaakt, zodat toppen nu ook bogen kunnen hebben naar toppen met lagere indices dan hun eigen index, maar dit zorgt er wel voor dat quasi elke graaf cykels bevat. Het slechtste geval, waarbij alle toppen hetzelfde symbool hebben, heeft dus een zeer grote kans om palindromisch onbegrensd te zijn. Bijgevolg, aangezien het algoritme stopt zodra het een palindromische cykel tegenkomt, worden deze grafen zeer snel verwerkt, ongeacht hun grootte. Bijgevolg is de enige logische keuze om te kijken hoeveel invloed het controleren op cykels heeft op de uitvoeringstijd. Ik heb dus dezelfde tests herhaald met willekeurige, acyclische

grafen. De eerste implementatie met een `HashSet` om te controleren op cykels had voor de tien grafen elk van lengte 100, 200 en 400 respectievelijk 966 ms, 4 s 849 ms en 79 790 ms nodig, met verhoudingen 5.01 en 16.46. Dit toont dat de test op de aanwezigheid van cykels wel invloed heeft op de uitvoeringstijd van het algoritme, maar de complexiteit is hetzelfde gebleven, wat overeenstemt met de eerdere resultaten. Als we nu de `HashSet` vervangen door een eenvoudige tweedimensionale `boolean`-array (want zowel voor hashing als voor equals werden enkel de indices van de buitenste toppen van het pad gebruikt), dan loopt de uitvoeringstijd quasi gelijk(zie hieronder) met die van het acyclische algoritme, maar de benodigde hoeveelheid geheugenruimte is natuurlijk wel groter geworden.

Om de finale implementaties met elkaar te vergelijken, heb ik het bovenstaande herhaald voor 100 grafen met 100, 200 en 400 toppen en heb ik beide algoritmen exact dezelfde grafen laten verwerken om ervoor te zorgen dat eventueel willekeurige "eenvoudige" grafen geen invloed zouden hebben op de effectieve verschillen in snelheden.

- Om 100 grafen met 100 toppen te verwerken had het DAG-algoritme 2.104 seconden nodig, en het DG-algoritme 1.977 seconden.
- Om 100 grafen met 200 toppen te verwerken, had het DAG-algoritme 21.193 seconden nodig, en het DG-algoritme 20.381.
- Om ten slotte 100 grafen met 400 toppen te verwerken, had het DAG-algoritme 467.691 seconden nodig, en het DG-algoritme 472.365 seconden.

De afwijking van de factor 16 kan verklaard worden door het weglaten van constante termen bij het bepalen van de complexiteit. Bovendien is een willekeurige graaf met 200 toppen niet equivalent met het verdubbelen van een graaf met 100 toppen. De bekomen waarden benaderen wel duidelijk de berekende waarden.