Gegevensstructuren en Algoritmen: Practicum 3

Wannes Vande Cauter r0713398

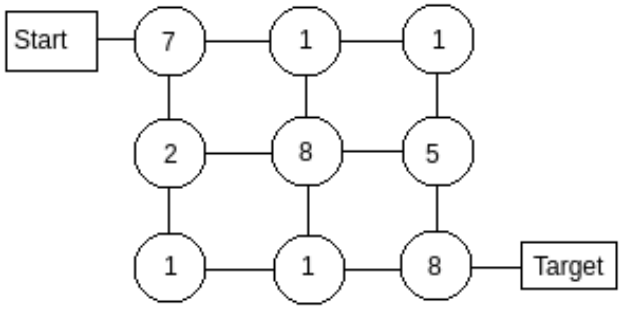
Academiejaar 2019-2020

1 Gegeven de volgende afbeeldingen als invoer (met het eerste element van 3 cijfers de hoeveelheid rood, de 2de de hoeveelheid groen en de derde de hoeveelheid blauw), Geef de grafe die als input dient voor het kortste pad algoritme. Wat is het resulterende kortste pad?

(7,0,0)	(0,1,0)	(0,0,1)
(2,0,0)	(0,8,0)	(0,0,5)
(1,0,0)	(0,1,0)	(0,0,8)

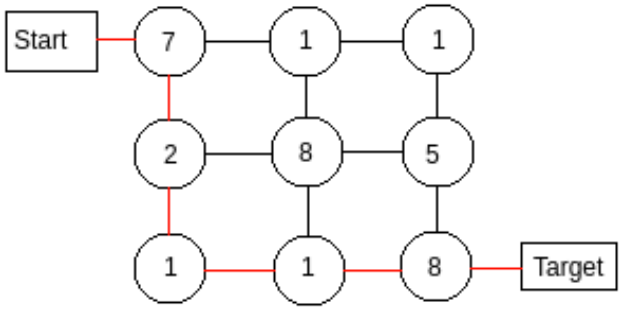
(0,0,0)	(0,0,0)	(0,0,0)
(0,0,0)	(0,0,0)	(0,0,0)
(0,0,0)	(0,0,0)	(0,0,0)

De 'afstand' tussen 2 kleuren is gedefinieerd als $\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$, met (r_i, g_i, b_i) de cijfers dat een kleur definiëren voor een pixel i.
De grafe dat de verbindingen tussen mogelijke pixels geeft en de pixels hun kost erbij definieert is als volgt:



Waarbij het pad moet beginnen bij Start en eindigen bij Target.

Het kortste pad is het volgende (volgens de rode lijnen):



De afstand van dit pad is dan $7 + 2 + 1 + 1 + 8 = 19$.

2 Stel dat de formule voor de 'afstand' tussen 2 kleuren $\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2}$ was in de plaats van $\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$, zijn er dan afbeeldingsparen waarbij het algoritme sneller is? Waarom? Zijn er afbeeldingsparen waarbij het net trager is? Waarom?

Er zijn afbeeldingsparen waarbij dit sneller is, bijvoorbeeld een afbeelding waar de groene en rode waarden op de diagonaal gelijk zijn aan de andere afbeelding zijn groene en rode waarden hier, maar compleet verschillende blauwe waarden, en de waarden niet op de diagonaal verschillen wel voor rode en groene waarden, maar minder extreme dan de blauwe.

Met dit voorbeeld zou het algoritme met de originele formule niet de diagonaal vinden als kortste pad, maar een boel complexere paden berekenen.
Met de nieuwe formule daarintegen is de kost voor pixels op de diagonaal telkens 0, dus dit blijft het pad dat het algoritme blijft gebruiken om verder te rekenen.
Gezien dit ook het kleinst aantal pixels is naar het einde zal dit in ieder geval veel sneller zijn.

Er zijn ook afbeeldingsparen waarbij dit trager is, bijvoorbeeld een afbeelding met dezelfde blauwe waarde op de diagonaal en licht verschillende groene en rode waarde op de digaonaal. Dan zijn er nog extreem verschillende blauwe waarde op de niet diagonale pixels.

Met dit voorbeeld en de oude formule zou het algoritme weer telkens de diagonaal verder berekenen tot het doel is uitgekomen en dus het minst mogelijke aantal berekeningen moeten uitvoeren.

Met de nieuwe formule zou de diagonaal grotere afstanden geven en moet de computer dus complexere paden berekenen om uiteindelijk tot het antwoord te komen met meer rekenwerk.

3 Wat is de tijdscomplexiteit van je programma op afbeeldingen met grootte $1XN$ of $NX1$? Zal het algoritme van Dijkstra even snel zijn in functie van het aantal pixels als de afbeelding vierkant (NXN, met N^2 pixels) is? Waarom?

Voor het bepalen van de tijdscomplexiteit is het in dit programma voldoende om alle modificaties aan lijsten te tellen, gezien dit het meeste rekenwerk is. Dan is onze berekening een goede benadering van de werkelijke tijdscomplexiteit.

De lagere orde termen van de tijdscomplexiteit kunnen ook verwaarloost worden, voor lage waarden van N maakt de complexiteit niet uit en voor hoge waarden van N zijn de lagere termen veel kleiner dan de hogere.

Het programma roept in eerste instantie enkel Sticher.stitch(int[ ][ ] image1, int[ ][ ] image2) op.
Hierin gebeuren geen relevante instructies behalve eerst Stitcher.seam(int[ ][ ] image1, int[ ][ ] image2) oproepen en daarna Stitcher.floodfill(Stitch[ ][ ] mask) oproepen.

Het is dus telkens voldoende de complexiteit van deze 2 functies te berekenen en die bij elkaar op te tellen.

3.1 $1XN$ of $NX1$

1. Stitcher.seam(int[ ][ ] image1, int[ ][ ] image2):

   Voor een paar afbeeldingen van deze vorm zal de seam altijd N elementen lang zijn, want hij begint linksboven (het eerste element van de afbeelding) en eindigt rechtsonder (het laatste element van de afbeelding), terwijl de seam alleen maar pixel per pixel de laatste kan naderen.

   Er staat in deze functie een lus die net zo lang duurt tot de seam gevonden is, in dit geval zal dit ook N iteraties kosten, gezien voor elke pixel telkens alleen maar de volgende kan toegevoegd worden in de lijst van pixels.
   Gezien voor deze lus de lijsten nog leeg zijn, zal de tijdscomplexiteit van operaties voor de lus in constante tijd zijn en dus verwaarloosbaar zijn in vergelijking met complexiteiten in functie van N.

   Binnen de lus zijn er 2 relevante instructies, 1 ervan staat binnen nog een andere lus met altijd 7 iteraties (alle pixels rond de huidige behalve de vorige), maar ook binnen een if-statement dat in dit geval slechts 1 van die 7 keer zal waar zijn (de pixel binnen de grenzen van de afbeeldingen dat niet de vorige is).

   Hier worden Stitcher.addToPriorityList(Node n) en PriorityQueue.poll() dus in het totaal N keer opgeroepen.

   Belangrijk is dat de lijst priorityList altijd een lengte van 0 zal hebben tijdens Stitcher.addToPriorityList(Node n) en een lengte van 1 tijdens PriorityQueue.poll(). Dit is omdat er telkens het enige element uit de lijst wordt gehaald om zijn opvolgers toe te voegen, maar er is telkens maar 1 opvolger dat niet de vorige is en dus in de lijst kan worden toegevoegd.

Ten gevolge zijn de operaties op die lijst dus in constante tijd. In Stitcher.addToPriorityList(Node n) worden er 3 relevante operaties uitgevoerd: HashSet.contains(), PriorityQueue.add(), HashSet.add().

In het totaal is de tijdscomplexiteit van die eerste lus met N iteraties dus $\sim 4N$.

Dan volgt een lus voor de lengte van de seam (dus weer N iteraties), waarin ArrayList.add() wordt opgeroepen dat in constante tijd gebeurt. Hier is de tijdscomplexiteit $\sim N$ zodat de totale tot nu toe $\sim 5N$ is.

Dan wordt Collections.reverse() uitgevoerd op de lijst met N elementen, dit heeft een tijdscomplexiteit van $\sim N$.

De tijdscomplexiteit van de functie seam is in dit geval dus $\sim 6N$.

2. Stitcher.floodfill(Stitch[ ][ ] mask):

   De seam heeft hier een lengte van N en er zijn ook maar N pixels in het totaal, dus alle pixels liggen op de seam.

   De functie floodfill doet geen enkele betekenisvolle instructie wanneer alle elementen op de seam liggen en heeft dus een constante tijdscomplexiteit.

De totale tijdscomplexiteit is de som van die van seam en floodfill, waarbij lagere orde termen geschrapt worden, dus in dit geval is die $\sim 6N$.

3.2 NXN

Definieer $M = N^2$, de tijdscomplexiteit wordt bepaalt in functie van M, het aantal pixels.

- Stitcher.seam(int[ ][ ] image1, int[ ][ ] image2):

  Om de tijdscomplexiteit in het ergste geval te berekenen moet ik hier bepalen hoeveel elementen er maximaal in de prioriteitslijst kunnen komen te zitten.
  De limiet is gesteld door het aantal pixels - het aantal pixels dat er al in is geweest en er terug uit is waardoor die er niet meer in kan.
  Dit betekend dat wanneer deze lijst zijn limiet haalt, alle pixels oftewel in de prioriteitslijst zitten of hebben gezeten, dus dat ook de eindpixel hierin zit en de oplossing gevonden is, dus het aantal iteraties waarna de lijst zijn maximum bereikt is het maximale aantal iteraties van die functie.

  Na 1 iteratie zijn er 3 elementen in de rij (de 3 pixels aanliggend aan degene linksboven).
  Er is dan 1 pixel dat niet meer in de lijst zit en er niet meer in kan komen.

  Vanaf dan is het maximum aantal nieuwe elementen 5: er zijn er minstens 3 van de aanliggende nog van de vorige pixel al toegevoegd. Er gaat er ook telkens 1 uit dat er niet meer terug in kan.

Bij de volgende ongelijkheid is het linkerlid het maximaal aantal elementen en het rechterlid het aantal elementen, voor i: de i-de iteratie na de eerste.

$$M - 1 - i \geq 3 + 5i$$

$$M - 4 \geq 6i$$

Deze toename van 5 nieuwe elementen is enkel mogelijk voor pixels dat niet aan de rand zijn, daar zijn er $(\sqrt{M} - 1)^2 = M - 2\sqrt{M} + 1$ van.
Als de gelijkheid opgaat wanneer i hieraan gelijk is, dan heeft de lijst zijn maximum nog niet bereikt na alle pixels in het midden te gebruiken en zal er dus een toename met randpixels gebeuren. Anders bereikt hij zijn maximum al eerder en kunnen we de iteratie waarin en het maximum zelf berekenen door de ongelijkheid naar een gelijkheid te veranderen.

$$M - 4 \geq 6(M - 2\sqrt{M} + 1)$$

$$0 \geq 5M - 12\sqrt{M} + 10$$

Voor grote waarden van M gaat dit niet op, dus we kunnen nu het maximum en de iteratie vinden:

$$M - 4 = 6i$$

$$i = \frac{M - 4}{6}$$

Dus het maximum wordt bereikt op iteratie $\frac{M-4}{6} + 1 = \frac{M+2}{6}$.
Het aantal elementen in de lijst is in het ergste geval dus $5(j - 1) + 3 = 5j - 2$ bij de j-de iteratie van de lus.

Gezien voor deze lus de lijsten nog leeg zijn, zal de tijdscomplexiteit van operaties voor de lus in constante tijd zijn en dus verwaarloosbaar zijn in vergelijking met complexiteiten in functie van M.

In de lus staat nog een lus van 7 iteraties met daarin Stitcher.addToPriorityList(Node n), hierin worden 3 relevante instructies opgeroepen: HashSet.contains(), PriorityQueue.add(), HashSet.add(). De instructies op de HashSet gebeuren in constante tijd, terwijl die van PriorityQueue afhankelijk is van van M, dus die van de HashSet worden verwaarloost.
De complexiteit van PriorityQueue.add() met een lijst van x elementen is $\sim log(x)$. De complexiteit is nu dus $\sim \Sigma_{j=1}^{\frac{M+2}{6}} (7log(5j - 2))$.

Na de lus van 7 iteraties binnen de eerste lus wordt ook PriorityQueue.poll() gebruikt, dat dezelfde complexiteit heeft als PriorityQueue.add(), dus de tijdscomplexiteit van de eerste lus is $\sim \Sigma_{j=1}^{\frac{M+2}{6}} (8log(5j - 2))$
Het vervolg van de tijdscomplexiteit is afhankelijk van de lengte van de seam, maar gezien floodfill in complexiteit afneemt bij een grotere lengte van de seam laat ik dit voorlopig symbolisch (SL) staan, zodat ik later kan uitwerken welke waarde hiervoor de grootste totale tijdscomplexiteit bezorgt.

Eerst volgt een lus van SL iteraties met daarin ArrayList.add(), dat een constante tijdscomplexiteit heeft. Daarna wordt Collections.reverse() gebruikt op een lijst van lengte SL.

De totale tijdscomplexiteit van seam is dus $\sim \Sigma_{j=1}^{\frac{M+2}{6}} (8log(5j - 2)) + 2SL$

- Stitcher.floodfill(Stitch[ ][ ] mask):

  De functie begint met 4 lussen van lengte $\sqrt{M}$ met daarin telkens een ArrayList.add(), zodat de tijdscomplexiteit dan al $\sim 4\sqrt{M}$ is.

  Er zijn 2 lijsten waarbij de 2 lijsten samen maximum $M - SL$ elementen bevatten.
  De maximale complexiteit komt voor wanneer alle elementen in 1 van de 2 lijsten terecht komen. Er zijn 2 equivalente lussen, 1 voor elke lijst, dus we beschouwen de complexiteit van 1 ervan voor alle elementen in die ene lijst, zodat we de complexiteit in het ergste geval hebben.

  Nu moet het maximum aantal elementen in die lijst berekend worden:

  De lijst begint met maximum $2\sqrt{M} - 2$ elementen, elke iteratie van de lus worden maximaal 3 nieuwe elementen toegevoegd en 1 verwijdert dat niet meer in de lijst kan komen, tot het maximum van de lijst bereikt is.

  $$M - SL - i \geq 2\sqrt{M} - 2 + 3i$$

  Dit omzetten naar een gelijkheid geeft de iteratie waarop het maximum wordt bereikt:

  $$i = \frac{M - SL - 2\sqrt{M} + 2}{4}$$

  De lus die dan volgt gebruikt telkens ArrayList.add() met een constante tijd, dus de tijdscomplexiteit voor deze functie volgt als $\sim 4\sqrt{M} + \Sigma_{i=1}^{\frac{M-SL-2\sqrt{M}+2}{4}}(2\sqrt{M} - 2 + 3i)$

De totale tijdscomplexiteit word nu gegeven als $\sim \Sigma_{j=1}^{\frac{M+2}{6}}(8log(5j-2)) + 2SL + 4\sqrt{M} + \Sigma_{i=1}^{\frac{M-SL-2\sqrt{M}+2}{4}}(2\sqrt{M} - 2 + 3i)$.

$$\sim \Sigma_{j=1}^{\frac{M+2}{6}}(8log(5j-2)) + 2SL + 4\sqrt{M} + (2\sqrt{M} - 2)(\frac{M - SL - 2\sqrt{M} + 2}{4}) + \Sigma_{i=1}^{\frac{M-SL-2\sqrt{M}+2}{4}}(3i)$$

Hier wordt al snel duidelijk dat de tijdscomplexiteit zal afnemen voor hogere waarden voor SL zolang die niet veel groter is dan M, gezien de negatieve termen al minstens de orde $SL\sqrt{M}$ hebben.
De seam kan niet groter dan M zijn gezien er maar M pixels zijn en moet minstens $\sqrt{M}$ zijn gezien die van linksboven tot rechtsonder moet gaan.
De grootste tijdscomplexiteit zal dus zijn wanneer $SL = \sqrt{M}$:

$$\sim \Sigma_{j=1}^{\frac{M+2}{6}}(8log(5j-2)) + 6\sqrt{M} + (2\sqrt{M} - 2)(\frac{M - 3\sqrt{M} + 2}{4}) + 3\Sigma_{i=1}^{\frac{M-3\sqrt{M}+2}{4}}(i)$$

Gezien de formule $\Sigma_{x=1}^{y}(x) = \frac{1}{2}y(y+1)$:

$$\sim \Sigma_{j=1}^{\frac{M+2}{6}}(8log(5j-2)) + 6\sqrt{M} + (2\sqrt{M} - 2)(\frac{M - 3\sqrt{M} + 2}{4}) + \frac{3}{2}(\frac{M - 3\sqrt{M} + 2}{4})(\frac{M - 3\sqrt{M} + 6}{4})$$

Hier is duidelijk te zien dat er een term van orde M^2 voorkomt. alle lagere termen kunnen dus verwaarloost worden. (De eerste sommatie zal nooit groter dan orde $Mlog(M)$ kunnen zijn, wat nog steeds een kleinere orde dan M^2 is)

$$\sim \frac{3}{2}(\frac{M}{4})(\frac{M}{4}) =\sim \frac{3}{32}M^2$$

De tijdscomplexiteit $\sim \frac{2}{32}M^2$ is trager dan $\sim 6N$.
De reden dat het algoritme minder efficiënt is, is voornamelijk omdat in een 1-dimentionale afbeelding het algoritme heel wat vereenvoudigingen heeft, voornamelijk het feit dat elk element op de seam ligt, waardoor er geen lussen moeten gebeuren om de niet-seam elementen in te vullen.

Het is uiteindelijk het invullen van die seam elementen dat voor de hogere orde van de tijdscomplexiteit heeft gezorgd.

4 Stel je wil dat je seam geen complexe vormen kan aannemen, hoe kan je hiervoor zorgen? (hoe voorkom je dat de seam terug naar boven of links kan lopen)

In de klasse Node.java wordt een functie Node.neighbors() gebruikt om mogelijke nieuwe elementen van de seam te vinden.
Als daar de waarden in de lus voor i en j worden aangepast zodat deze van 0 tot en met 1 gaan, zullen enkel de mogelijke elementen dat stijgen in minstens een van de 2 coördinaten en in geen van beide dalen worden teruggegeven.

Op die manier kan een opvolgend seam element alleen maar naar onder, naar rechts, of schuin naar rechts-onder gaan.

5 De lengte van een pad is gedefinieerd als de som van de gewichten van de edges van dat pad. Stel dat je programma niet het kortste maar het langste pad (zonder meermaals op eenzelfde node te komen) zoekt, hoe zou de afbeelding die je programma dan teruggeeft eruit zien?

Om een zo groot mogelijk pad te vinden, zal de seam in de meeste gevallen veel meer nodes bezoeken alvorens naar het einde te gaan.

Dit zorgt voor een heel complexe vorm waarbij vele kleine stukken van de verschillende afbeeldingen door elkaar lopen.

Het eindresultaat zal op het overlappende gebied er dus uitzien als een soort gevlekte structuur waar vlekken met kleuren van beide afbeeldingen zich door elkaar hebben gemengd.