Verslag Practicum 3:

Arnout Coenegrachts, r0665757

29 mei 2020

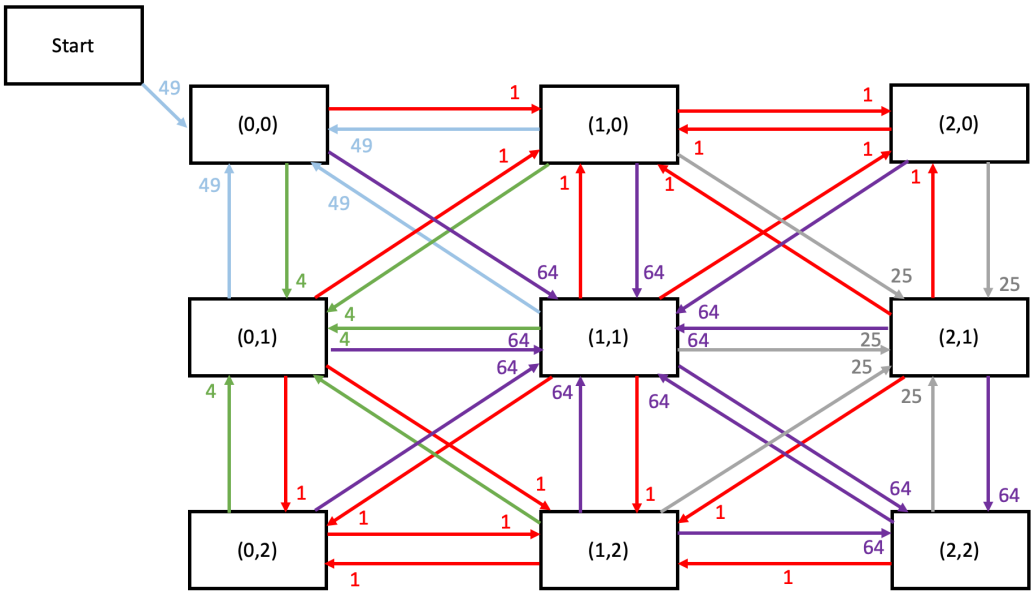

Figure 1: De graaf die als input dient bij de afbeeldingen uit **vraag 1**. De getallen in de (rechthoekige) vertices zijn de (x,y)-coordinaten. Voor de duidelijkheid zijn alle edges met hetzelfde gewicht in dezelfde kleur gezet. Er is één extra vertex gelabeld *Start* toegevoegd, om het gewicht van de werkelijke startpositie (0,0) toe te voegen, ook al zal deze geen invloed hebben om de minimalisatie zelf.

1 Vraag 1:

De graaf die als input dient is gegeven in figuur 1. Elke vertex is benoemd met zijn (x,y)-coordinaten, en de edges met hetzelfde gewicht hebben telkens dezelfde kleur. De rode edges hebben gewicht 1, de groene hebben gewicht 4, de grijze hebben gewicht 25, de blauwe hebben gewicht 49, en de paarse hebben gewicht 64. Elke edge die in dezelfde vertex eindigen hebben hetzelfde gewicht, namelijk het verschil in kleur, gegeven door **ImageCompositor.pixelSqDistance(int x, int y)**. Er is een extra vertex gelabeld *Start* toegevoegd, om de kost van de (0,0)-vertex mee te geven, ook al zal deze geen invloed hebben op het algoritme, aangezien deze vertex toch niet vermeden kan worden. Het resulterende kortste pad wordt gegeven door de volgende coordinaten: (0,0) - (0,1) - (1,2) - (2,2). De kost van dit pad is $49 + 4 + 1 + 64 = 118$.

2 Vraag 2:

Ik vermoed dat de *color*-figuren hierdoor sneller gaat gaan, omdat deze enkel maar de kleuren rood, groen en zwart bevatten, en de blauwwaarde hier dus weinig relevantie gaat hebben. Voor de *zee*- en *spiraal*-figuren denk ik dat deze aanpassing het trager maakt, omdat deze afbeeldingen veel blauw bevatten, en dus een belangrijke kleur is om mee rekening te houden.

3 Vraag 3:

De methode **stitch** bevat alle informatie die nodig is om de tijdscomplexiteit te bepalen.

- Deze voert eerst de methode **seam**, die een lijst met posities die de seam vormen geeft, één keer uit.
  - Seam zelf doet eerst de methode **setWeights** uit. Deze methode vult een matrix van dezelfde grootte als het overlappende deel van de twee afbeeldingen met de gewichten gegeven door **ImageCompositor.pixelSqDistance**. Hiervoor is er een dubbele loop die over de het aantal rijen en over het aantal kolommen gaat, met daarbinnen 3 array-accesses. Aangezien voor deze oefening ofwel het aantal rijen ofwel het aantal kolommen 1 is, is de complexiteit van deze methode $\sim 3N$.
  - Hierna komt een loop die een andere matrix genaamd *distances* vult met overal ∞. Het doel van *distances* is de afstand van de begin positie tot de overeenkomstige plaats bij te houden. De complexiteit van deze lus is $\sim N$.
  - Daarna zet het programma de afstand tot de beginpositie gelijk aan het gewicht van de beginpositie. Dit zijn 2 array-accesses.
  - Hierna komt de grootste stap van het hele algoritme. Er is een while-lus met daar in een for-lus met daar in een methode genaamd **relax**. Deze methode zorgt er voor dat de afstanden in *distances* telkens de kortste zijn. Hiervoor worden er 5 array-accesses gebruikt. De for-loop waarin **relax** zich bevindt, wordt uitgevoerd voor hoeveel buren de methode **neighbours** kan vinden. Normaal gezien zou dit er ongeveer 8 zijn, maar in dit geval kan dit er maar 2 zijn, namelijk ofwel één edge naar links en één naar rechts ofwel één edge naar boven en één naar onder. De while-lus zelf wordt N keer uitgevoerd. De complexiteit van deze stap is dus $\sim N \times 2 \times 5 = \sim 10N$.
  - Hierna komen nog 2 aparte for-lussen die ieder N keer uitgevoerd worden, met 2 array-accesses per keer, wat samen een complexiteit van $\sim 4N$ opleverd.

  De totale complexiteit van de methode **seam** is dus $\sim 3N+N+2+10N+4N = \sim 18N$.

- Na deze methode komt een for-lus die evenveel uitgevoerd wordt als het aantal posities in de lijst die de methode **seam** heeft gegeven. Dit zal in dit geval N keer zijn. Binnen deze for-lus staat één array-access, wat deze stap een complexiteit van $\sim N$ geeft.

- Hierna wordt de methode **floodfill** gebruikt. Deze heeft een dubbele lus die over het aantal rijen en over het aantal kolommen gaat. De inhoud hiervan wordt dus N keer uitgevoerd. Deze lus gaat per rij, van links naar rechts, de mask vullen met de juiste afbeelding. Hiervoor houdt het een "*filler*" bij, wat de indiceert welke afbeelding momenteel gebruikt wordt. Het gebruikt de methodes **checkAbove** en **checkBelow** samen met de twee booleans *up* en *down* om te kijken wanneer deze *filler* moet overschakelen naar de andere afbeelding. Deze voorwaarde is voldaan als de huidige pixel een *Stitch.SEAM*-pixel is, en er zowel boven als onder ook een *Stitch.SEAM*-pixel is. In het geval van een langwerpige afbeelding met dikte 1, is er maar 1 array-access per keer dat de loop wordt uitgevoerd, namelijk om te kijken of de huidge pixel al dan niet een *Stitch.SEAM*-pixel is. De complexiteit van de **floodfill** methode is dus $\sim N$.

De volledige complexiteit van **stitch** is dan $\sim 18N + N + N =\sim 20N$.

Als de afbeelding vierkant is, zal het algoritme een langere uitvoeringstijd hebben. Dit komt omdat er een aantal stappen ingewikkelder worden omdat het aantal relevante dimensies met 1 toeneemt.

- In de stap van de methode **seam** die eerder de grootste stap genoemd werd, verandert de complexiteit van $\sim 10N$ naar $\sim 40N$. Dit komt omdat de methode **neighbours** nu 8 buren in plaats van 2 buren kan vinden. De complexiteit van **seam** wordt dan $\sim 3N + N + 2 + 40N + 4N =\sim 48N$.

- De seam die teruggegeven wordt door de gelijknamige methode kan nu wel korter zijn dan die van een langwerpige afbeelding. De seam gaat minimaal van grootte $\sqrt{N}$ zijn, wat de diagonale seam is.

- Tenslotte gaat de methode **floodfill** ook een grotere complexiteit hebben. Er zijn twee opties: als de huidige pixel geen *Stitch.SEAM*-pixel is, komt er één array-access bij om de pixel in te vullen met de relevante afbeelding. Als de huidige pixel wel een *Stitch.SEAM*-pixel is, er 6 tot 7 array-accesses bij: de methodes **checkAbove** en **checkBelow** geven er ieder 3, en er kan er nog één extra bijkomen wanneer er gecontroleerd wordt of de volgende pixel ook een *Stitch.SEAM*-pixel is. Het gaat veel vaker voorkomen dat een pixel geen *Stitch.SEAM*-pixel is in een vierkante matrix, dus gemiddeld gezien gaat de complexiteit van **floodfill** dus van $\sim N$ naar $\sim 2N$.

De volledige complexiteit van **stitch** in het geval van een vierkante afbeelding is dan $\sim 48 + \sqrt(N) + 2N =\sim 50N$.

4 Vraag 4:

Deze twee eisen kunnen we gemakkelijk aan voldoen door een paar eenvoudige aanpassingen te maken in de **neighbours**-methode. Deze code kan je vinden in figuur 2. Deze methode maakt een lijst aan met maximaal 8 edges die vertrekken uit de gegeven positie en eindigen in de aangrenzende posities.

- Om er voor te zorgen dat de seam niet terug naar boven kan, moeten we de code, die edges naar posities met als y-coordinaat $y-1$ toevoegen, verwijderen. Deze code wordt enkel uitgevoerd als $y > 0$, en komt overeen met regels 127 tot en met 138.

- Om er voor te zorgen dat de seam niet terug naar links kan gaan, verwijderen we de code die edges naar posities met als x-coordinaat $x-1$ hebben. Deze code wordt enkel maar uitgevoerd als $x > 0$, wat deze keer niet in één blok samen staat. De code die moet worden weggelaten staat in regels 130 tot en met 133, regels 142 tot en met 145, en regels 151 tot en met 154.

5 Vraag 5:

Aangezien elke pixel een positief gewicht heeft, zal het langste pad elke pixel bevatten. Volgens de method **ImageCompositor.main** worden de *Stitch.SEAM*-pixels ingevuld als pixels uit afbeelding 2. Bijgevolg zal het overlappende deel uit de nieuwe afbeelding er volledig uitzien als de tweede afbeelding.

```
121
122     public List<Edge> neighbours(Position current, int row, int column) {
123         int x = current.getX();
124         int y = current.getY();
125         List<Edge> buren = new ArrayList<Edge>();
126         Position temp;
127         if (y > 0) {
128             temp = new Position(y-1, x);
129             buren.add( new Edge(current, temp, weightOf(temp) ) );
130             if (x > 0) {
131                 temp = new Position(y-1, x-1);
132                 buren.add( new Edge(current, temp, weightOf(temp) ) );
133             }
134             if (x < column-1) {
135                 temp = new Position(y-1, x+1);
136                 buren.add( new Edge(current, temp, weightOf(temp) ) );
137             }
138         }
139         if (y < row-1) {
140             temp = new Position(y+1, x);
141             buren.add( new Edge(current, temp, weightOf(temp) ) );
142             if (x > 0) {
143                 temp = new Position(y+1, x-1);
144                 buren.add( new Edge(current, temp, weightOf(temp) ) );
145             }
146             if (x < column-1) {
147                 temp = new Position(y+1, x+1);
148                 buren.add( new Edge(current, temp, weightOf(temp) ) );
149             }
150         }
151         if (x > 0) {
152             temp = new Position(y, x-1);
153             buren.add( new Edge(current, temp, weightOf(temp) ) );
154         }
155         if (x < column-1) {
156             temp = new Position(y, x+1);
157             buren.add( new Edge(current, temp, weightOf(temp) ) );
158         }
159         return buren;
160     }
161
```

Figure 2: De **neighbours**-methode die de edges van de gegeven pixel naar alle pixels aanliggend aan de deze pixel teruggeeft in een lijst.