Gegevensstructuren en Algoritmen: Practicum 3:

Verslag op de implementatie van een Image Compositing algoritme.

Mathijs Hubrechtsen, r0664977

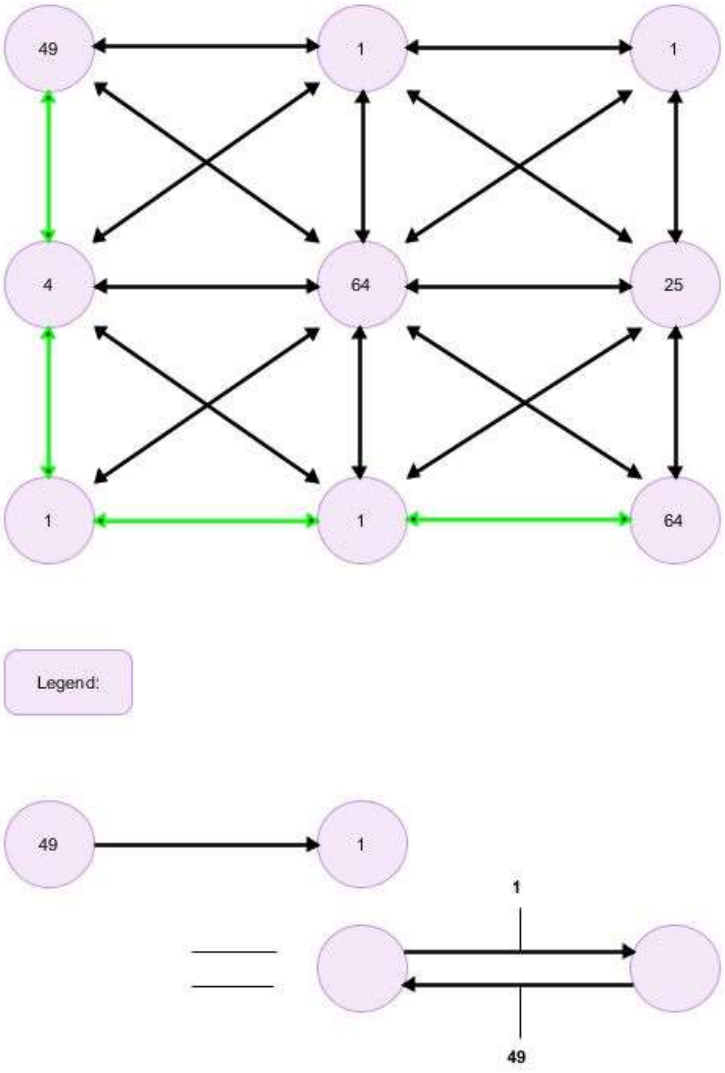
19 mei 2017

Inhoudsopgave:

0. Inleiding:

Dit verslag gaat over mijn implementatie van een Image Compositing Algoritme. Mijn implementatie maakt gebruik van een vrij standaardversie van Dijkstra's algoritme om de seam te vinden. Ook het floodfill algoritme is heel gelijkaardig aan hetgeen dat gespecifieerd werd in de opgave. Met natuurlijk als verschil dat het iteratief en niet recursief is. De bedoeling van dit verslag is op de gegeven vragen te beantwoorden, dit is ook de structuur dat het verslag zal volgen. In dit verslag zal ik vaak spreken over "nodes" of "neighbors". "Nodes", in mijn practicum, zijn gewoon specifieke posities of pixels in een image. "Neighbors" zijn de buren van een bepaalde "node", dit is reeds besproken in de opgave voor dit practicum. Een positie is een neighbor van een andere positie als de gegeven methode isAdjacentTo (Zie Position.html), true teruggeeft als het opgeroepen wordt op de een van de twee posities met de andere positie als argument.

1. Vraag 1: *Stel je programma wordt uitgevoerd op twee (gegeven) afbeeldingen (met geen offset). Geef de grafe (inclusief gewichten) die als input dient voor het kortste pad algoritme. Wat is het resulterende kortste pad?*



Het resulterende kortste pad is (0, 0) → (1, 0) → (2, 0) → (2, 1) → (2, 2).

2. Vraag 2: *Zijn er afbeeldingenparen waarop je programma met deze afstandsfunctie een snellere uitvoeringstijd heeft dan met de standaard afstandsfunctie? Waarom? Zijn er afbeeldingenparen waarop je programma met deze afstandsfunctie een tragere uitvoeringstijd heeft? Waarom?*

Oorspronkelijke afstandsfunctie = $\sqrt{(r_x - r_y)^2 + (g_x - g_y)^2 + (b_x - b_y)^2}$

Nieuwe afstandsfunctie = $\sqrt{(r_x - r_y)^2 + (g_x - g_y)^2}$

Ja, er zijn afbeeldingenparen waarop de theoretische uitvoeringstijd sneller is met de nieuwe afstandsfunctie. Neem als voorbeeld:

Image1:

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

Image2:

(0,0,0)	(1,0,0)
(1,1,0)	(0,0,99)

Met de oorspronkelijke afstandsfunctie zal Dijkstra eerst positie (0, 1) kiezen ($1 < \sqrt{2} < 99$), maar omdat dit niet het einde is zal hier na Dijkstra verder gaan. Het zal nu positie (1, 0) kiezen ($\sqrt{2} < 99$) en opnieuw verder gaan tot het uiteindelijk aan komt bij de eindpositie (1, 1). Dijkstra heeft dus met de oorspronkelijke afstandsfunctie 3 stappen nodig om het einde te bereiken: (0, 0) → (0, 1) → (1, 0) → (1, 1). Met de nieuwe afstandsfunctie wordt er onmiddellijk van (0,0) naar (1,1) gesprongen ($0 < 1 < \sqrt{2}$). Er wordt hier maar 1 stap gezet (0, 0) → (1, 1), wat evident minder tijd vergt dan 3 stappen er zijn dus afbeeldingparen waarop mijn programma met de nieuwe afstandsfunctie een snellere uitvoeringstijd heeft.

Ja, er zijn ook afbeeldingenparen waarop de theoretische uitvoeringstijd trager is met de nieuwe afstandsfunctie. Neem als voorbeeld;

Image1:

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

Image2:

(0,0,0)	(0,0,99)
(1,0,99)	(1,1,0)

Met de oorspronkelijke afstandsfunctie zal onmiddellijk gesprongen worden naar het einde ($\sqrt{2} < \sqrt{99} < 10$). Er wordt dus maar 1 stap gezet: (0, 0) → (1, 1), Terwijl met de nieuwe afstandsfunctie

elke positie zal worden bezocht voordat het einde wordt bereikt: (0, 0) → (0, 1) → (1, 0) → (1, 1) want: $\sqrt{2} > 1 > 0$. Opnieuw het is evident dat 3 stappen meer tijd vergt dan 1 stap, dus er zijn afbeeldingparen waarop mijn programma met de nieuwe afstandsfunctie een tragere uitvoeringstijd heeft.

3. Vraag 3: *Wat is de tijdscomplexiteit van je programma op een afbeeldingen met als grootte 1xN of Nx1? Zal Dijkstra even snel zijn in functie van het aantal pixels als de afbeelding NxN groot is i.p.v. 1xN of Nx1?*

De tijdscomplexiteit is gelijk aan ~N. Dit is omdat bij elke stap, Dijkstra de volgende positie zal nemen en toevoegen aan de prioriteit queue. Omdat Dijkstra nooit terug gaat naar een vorige positie, zal het altijd de enige keuze nemen: de volgende positie. Dijsktra doet dit tot als het einde bereikt wordt, dit is ~N. In andere woorden, Dijkstra zal steeds de optimale keuze nemen.

Het zal sneller zijn als de afbeelding 1xN of Nx1 is. De reden hiervoor is dat zoals gesteld, Dijkstra altijd de optimale keuze neemt bij een 1xN of Nx1 afbeelding, Dijkstra doet dit niet bij een NxN afbeelding. Daar is de tijdscomplexiteit worstcase: ~ $\frac{3+8}{2}$ * log(N) = ~5,5 * log(n). Er wordt hier het gemiddelde van het aantal "neighbors" als E (het aantal bogen) genomen. Dus worstcase, is Dijkstra trager bij een NxN afbeelding. Het beste geval zou zijn dat Dijkstra steeds de optimale keuze maakt, en dat deze keuze ook diegene is die het dichtste is bij het doel. Dit zou als resultaat de diagonaal geven van de NxN afbeelding, deze is $\sqrt{N^2 + N^2}$ = N * $\sqrt{2}$ pixels lang. Dus zelf in het beste geval zal Dijkstra trager zijn op een NxN afbeelding dan op een 1xN of Nx1 afbeelding.

4. Vraag 4: *Stel dat je wilt dat de seam geen complexe vormen mag aannemen, hoe kan je hiervoor zorgen?*

In mijn implementatie van het seam algoritme bepaal ik de buren van een bepaalde "node" (positie) met een specifieke methode. In deze methode wordt er rond de gegeven positie gekeken welke posities in range zijn van de dimensies van de gegeven image, deze "neighbors" (posities) worden dan teruggegeven. Als er gevraagd wordt dat de seam niet meer terug naar boven kan lopen moet er in deze methode gewoon de optie om naar de positie boven, linksboven en rechtsboven worden uitgesloten, deze "neighbors" zullen dus niet meer worden bekeken. Als er gevraagd wordt dat de seam niet meer terug naar links kan lopen moet er in deze methode gewoon de optie om naar de positie links, linksboven en linksonder weg worden gedaan. Dus enkel de onder, rechter of rechtsonder buren zullen worden bekeken.

5. Vraag 5: *Stel dat je programma niet het kortste pad zou zoeken, maar het langste pad dat niet meer dan 1x eenzelfde node bezoekt. Hoe zou de afbeelding die je programma als uitvoer geeft eruitzien?*

Een eerste opmerking is dat we niet meer hetzelfde algoritme kunnen gebruiken. Dijkstra kan namelijk niet het langste pad vinden. Dit is geweten omdat het kortste pad probleem opgelost kan worden in polynomiale tijd (zie vraag 4 voor T(N)), maar het langste pad kan niet gevonden worden in polynomiale tijd, het is namelijk NP moeilijk. Dit kan bewezen gelijkaardig aan hoe bewezen kan worden dat het Travelling Salesman Probleem NP is. Terug naar de oorspronkelijke kwestie, als we een nieuw algoritme ontwerpen dat dit langste pad vindt, weten we dat het langs elke "node" moet gaan, als dit zo zou zijn, dan kan er een pad ontworpen worden dat wel langs die node gaat en dus langer zal zijn. Het resultaat van dit algoritme zal zijn dat elke node deel zal zijn van de seam, meer bepaald alles ligt dus op de seam. Omdat alles op de seam ligt, zal er geen enkel pixel worden gekleurd kunnen worden, de uitvoer zal dus "leeg" zijn.