Beeldverwerking met de Micron automata Processor

Frank Goyens

9 juni 2017

Abstract

De Micron Automata Processor is een alternatief soort processor opgebouwd volgens het Multiple Instruction Single Data (MISD) model. De architectuur van de Micron Automata Processor is geen von Neumann architectuur. Het is daarom interessant om te onderzoeken of bestaande algoritmes efficiënter kunnen worden geïmplementeerd op dit soort architectuur. Tijdens het onderzoek naar toepassingen wordt een vergelijking gemaakt tussen implementaties op de verschillende architecturen.

Op de Micron Automata Processor zijn reeds toepassingen uit vakgebieden zoals Bioinformatica en Netwerkbeveiliging gevonden. Dit onderzoek tracht toepassingen te vinden in het vakgebied Beeldverwerking.

Momenteel worden de meeste algoritmes uit Visual Computing en Computervision efficiënt uitgevoerd op GPU's die zijn opgebouwd rond het Single Instruction Multiple Data (SIMD) model. Deze algoritmes zijn efficiënt zolang tijdens de uitvoering iedere processor core hetzelfde doet.

Toepassingen implementeren voor de Micron Automata Processor kan een uitdaging zijn voor zelfs ervaring ontwikkelaars. Het vereist dat het probleem moet worden gesteld als een pattern matching probleem. Voor het vakgebied Beeldverwerking is er een bijkomende moeilijkheid dat er wordt gewerkt met twee dimensionale data. Een groot deel van het onderzoek in deze thesis is toegewijd aan het oplossen van dit soort problemen.

Contents

1 Inleiding **5**

1.1 Vergelijking van bestaande architecturen 5

1.1.1 Generieke CPU architectuur 5

1.1.2 GPU architectuur 5

1.1.3 De Micron Automata Processor 7

1.2 Motivatie 7

1.3 Werkwijze van het onderzoek 7

1.4 Overzicht van het onderzoek 8

2 Achtergrond over automaten theorie **8**

2.1 Automaten volgens de Micron Automata Processor 9

3 De Micron Automata Processor **11**

3.1 Ontwikkelen van automaten 11

3.2 Meten van uitvoeringstijden 12

3.3 Limieten opgelegd door de hardware 12

3.4 Andere toepassingen 13

4 Algoritmes voor objectherkenning met automaten **14**

4.1 Kennismaking met de tools 14

4.1.1 1D hole detection in 1D binaire afbeeldingen 14

4.1.2 2D hole in een 2D binaire afbeelding (probleemstelling) . 16

4.2 Exact 2D pattern matching 17

4.2.1 Algoritme van Baker and Bird 17

4.2.2 Het vinden van letters en cijfers in een binaire afbeelding 17

4.2.3 Uitvoering op de Micron AP 21

4.2.4 Tijdscomplexiteit 22

4.2.5 Conclusie 23

4.3 Approximate pattern matching 24

4.3.1 Types automaten voor 1D pattern matching 24

4.3.2 Het opstellen van de encoderingsautomaat 25

4.3.3 Patroon encodering 27

4.3.4 Encodering van de afbeelding 28

4.3.5 Uitvoering van het zoeken naar 2D patronen 30

4.3.6 Samenvatting van het algoritme 33

4.3.7 Implementatie op de Micron AP 34

4.3.8 Tijdscomplexiteit 40

4.3.9 Conclusie 40

4.4 2D hole detection in 2D binaire afbeeldingen (deel 2) 41

4.4.1 Tijdscomplexiteit 44

4.4.2 Conclusie 47

5 Praktische toepassing op de AP 47
5.1 Het vinden van letters en cijfers in een afbeelding van een document 48
5.2 Strategie voor de implementatie 48
5.3 De afbeelding van het document 49
5.4 De afbeeldingen die als patronen dienen van alle letters en cijfers 49
5.5 Het genereren van een SFFRCO automaat voor ieder patroon . . 49
5.6 Het encoderen van documentstring voor ieder patroon met een SFFRCO automaat 51
5.7 Een SFOLCO automaat genereren voor ieder patroon 53
5.8 De geëncodeerde documentstring streamen naar de AP voor ieder patroon 55
5.9 Conclusie 58

6 Algemene conclusie en toekomstig werk 58
6.1 Synopsis 59
6.2 Toekomstig werk 59
6.2.1 Micron Automata Processor met 2D ondersteuning 60
6.2.2 Rotatie en schaal invariante patronen 60
6.2.3 Micron Automata Processor met ondersteuning voor veel output 60
6.2.4 Het zoeken naar woorden in een afbeelding van een document 60

1 Inleiding

Algoritmes uit Computergraphics en uit Computer Vision hebben tegenwoordig veel efficiënte implementaties op bestaande architecturen die men kan terugvinden op doorsnee computers. Er zijn echter nog bepaalde algoritmes waarvoor het niet triviaal is om goede implementaties te vinden; dit toont aan de er plaats is voor verbetering enerzijds bij het zoeken naar implementaties op bestaande architecturen maar anderzijds ook voor hardware architecturen op zich. In deze thesis wordt onderzocht wat de Micron Automata Processor te bieden heeft voor het vakgebied Beeldverwerking.

Het doel van deze thesis is om te onderzoeken of deze nieuwe architectuur nuttig is voor het parallelliseren van algoritmes uit het vakgebied Beeldverwerking. Dit zal gebeuren door bekend te worden met de sterktes en zwaktes van deze nieuwe architectuur.

1.1 Vergelijking van bestaande architecturen

Het is interessant om eerst bestaande architecturen te beschouwen voordat we een nieuw soort architectuur onderzoeken; iedere architectuur heeft sterktes en zwaktes dus het is belangrijk om te weten op welk vlak de architectuur zich onderscheidt van andere architecturen. We beschouwen een generieke CPU architectuur en ook GPU architectuur.

1.1.1 Generieke CPU architectuur

Een generieke CPU is onmisbaar bij de samenstelling van een computer. De generieke aard van de architectuur laat toe dat de CPU gebruikt kan worden voor een groot aantal zaken. Men kan een generieke CPU gebruiken voor wiskundige operaties maar ook voor het aansturen van hardware die aanwezig is in het systeem. Er is weinig parallellisatie in dit soort architectuur; de meeste hedendaagse CPU's op dit moment hebben 4 cores om parallelle bewerkingen uit te voeren.

Op bepaalde CPU's is het mogelijk om Single Instruction Multiple Data (SIMD) instructies uit te voeren, dit wil zeggen dat éénzelfde operatie op meerdere data eenheden wordt uitgevoerd. Dit soort functionaliteit vindt men bijvoorbeeld terug in cell processoren. Het aantal data eenheden is dikwijls gelijk aan 4. Dit soort instructies wordt daarom vooral gebruikt om efficiënt vector berekeningen te doen.

1.1.2 GPU architectuur

GPU architectuur is gemaakt om massief parallelle bewerkingen uit te voeren. Het aantal cores dat men op dit moment zou kunnen terugvinden in GPU architectuur ligt in de duizenden; het aantal cores in een GPU is dus van een geheel andere grootte orde dan bij generieke CPU architectuur.

GPU architectuur werkt volgens het SIMD principe; iedere thread voert dezelfde bewerking uit maar op andere data als input. Sinds het bestaan van

technologieën zoals NVIDIA CUDA is het mogelijk voor de gebruiker om ook generieke bewerkingen uit te voeren in plaats van enkel fixed functions en shader programs. GPU architectuur is complexer dan generieke CPU architectuur en het is tevens ook complexer om implementaties te programmeren. Dit komt omdat GPU architectuur typisch verschillende soorten geheugen bevat.

Neem Bijvoorbeeld het hardware model van NVIDIA CUDA in Figuur 1. Bij het programmeren beschikt men over het device (global) memory en shared memory, daarnaast zijn er ook caches aanwezig die men kan benutten door constanten te definiëren. Device memory is beschikbaar voor iedere processor, shared memory is enkel beschikbaar binnen de multiprocessor en is schaarser dan het device memory maar veel sneller toegankelijk.

Het doel bij het programmeren volgens het CUDA hardware model is om het strategie te hebben waarbij men het device memory zo weinig mogelijk gebruikt en het shared memory zo veel mogelijk. Met andere woorden, men moet op voorhand kunnen weten hoe men de input data kan opdelen. Men moet ook, net zoals bij shader programs, dynamic branching vermijden.

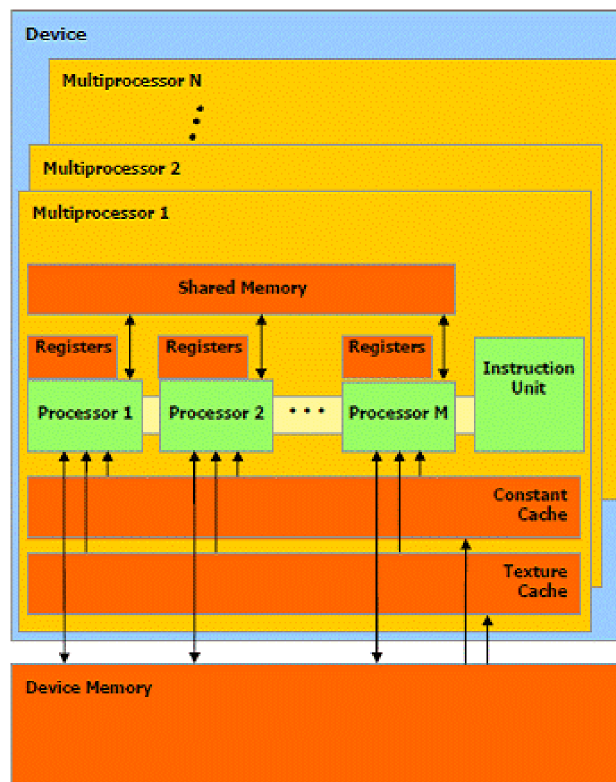


Figure 1: CUDA hardware model.

1.1.3 De Micron Automata Processor

De Micron Automata Processor (AP) is net zoals GPU architectuur gemaakt om massief parallelle berekeningen uit te voeren. De kracht van de AP wordt niet uitgedrukt in een aantal aanwezige cores maar in beschikbaar geheugen. Dit is omdat het geheugen de processor is. Dit betekent dus concreet dat er processing wordt gedaan door het geheugen aan te passen.

De werking van de AP is zeer verschillend dan de werking van generieke CPU's en GPU's. In hoofdstuk 3 worden de belangrijkste principes besproken van de AP voor dit onderzoek.

1.2 Motivatie

Wanneer een nieuwe processor architectuur beschikbaar wordt voor gebruikers is het niet meteen duidelijk wat de mogelijkheden hiervan zijn. Een voorbeeld hiervan is de architectuur van GPU's. De architectuur van GPU's laat toe om massief parallel berekeningen uit te voeren. Eerst waren GPU's niet programmeerbaar en werden GPU's enkel gebruikt voor visualisatie via fixed functions. Daarna werd het mogelijk om de GPU te programmeren met eigen shader programs zodat men niet meer afhankelijk was van fixed functions.

Het is mogelijk om deze shader programs te 'misbruiken' om ook generieke berekeningen te doen met de GPU. Dit was interessant indien men een algoritme kon implementeren als een shader program om zodanig gebruik te maken van het parallellisme van GPU architectuur.

Sinds 2008 is het mogelijk voor de gebruiker om generieke berekeningen uit te voeren op GPU architectuur met frameworks zoals CUDA of OpenCL. Dit heeft ervoor gezorgd dat bepaalde algoritmes (bvb. Convolutie, Discrete Fourier Transform, Mandelbrot set,...) zeer efficiënte implementaties hebben op GPU architectuur.

Hier leiden we uit af dat onderzoek naar toepassingen op een nieuwe architectuur verrijkend kan zijn. Het is reeds bewezen dat de architectuur op de Micron Automata Processor zeer nuttig is voor toepassingen uit de vakgebieden van Bioinformatica en Netwerkbeveliging.

1.3 Werkwijze van het onderzoek

Het onderzoek zal voornamelijk gaan over string matching algoritmes die kunnen worden toegepast in Beeldverwerking. Er komen voornamelijk string matching algoritmes aan bod omdat deze efficiënt kunnen worden toegepast op de AP; dit blijkt uit de reeds bestaande toepassingen op de AP (zie sectie 3.4).

Het onderzoek kan worden opgedeeld in twee delen. Het eerste deel gaat over algoritmes die eventueel niet meteen een praktische toepassing hebben, deze algoritmes dienen om beter inzicht te verkrijgen in de AP of ze dienen als basis voor meer praktische toepassingen. Het tweede deel gaat over toepassingen van algoritmes vanuit een praktisch standpunt.

Eerst zal kennis worden gemaakt met de SDK van de Micron AP. Om kennis te maken met de tools zal het probleem van hole detection worden gebruikt. Vervolgens komen twee pattern matching algoritmes aan bod die reeds bestaan maar nog niet zijn toegepast op de Micron AP, de twee algoritmes zijn 2D exact pattern matchinig en 2D approximate pattern matching [3]. Er zullen voornamelijk algoritmes voorkomen die worden uitgevoerd op binaire afbeeldingen zodat er kan worden gefocust op de toepassing op de AP.

Ten slotte zal het algoritme van 2D approximate pattern matching praktisch worden toegepast en zullen hieruit statistieken worden gehaald in verband met bijvoorbeeld uitvoertijden.

1.4 Overzicht van het onderzoek

In hoofdstuk 2 zal kort de theorie achter automaten worden toegelicht, hier zullen ook bepaalde verschillen aan bod komen tussen klassieke automaten en automaten in Anml, de taal waarin automaten worden gedefiniëerd voor de AP. In hoofdstuk 3 wordt toegelicht hoe men automaten ontwikkeld in Anml, hoe uitvoertijden kunnen worden berekend en met welke limieten men rekening moet houden. Ook worden hier andere toepassingen toegelicht van de AP om een betere achtergrond te scheppen. Hoofdstuk 4 is het onderzoek naar beeldverwerking algoritmes op de AP. Een praktische toepassing van een algoritme in hoofdstuk 4 is te vinden in hoofdstuk 5. Ten slotte is een algemene conclusie te vinden in hoofdstuk 6.

2 Achtergrond over automaten theorie

De Micron Automata Processor of AP is ontworpen rond de theorie van automaten. De manier waarop deze hardware wordt gebruikt is door een *non-deterministic finite automaton* of NFA op te stellen. Deze NFA wordt gecompileerd tot een binaire image die direct op de AP kan worden geplaatst. Het is niet nodig om in detail in te gaan op deze theorie dus enkel de belangrijkste concepten zullen kort worden toegelicht.

Een type automaat dat reeds is vermeld is een NFA. Formeel gezien is een NFA de quintuple (Q, A, δ, I, F):

- Q is een set van states.
- A is een eindig input alfabet.
- δ is een functie die een state uit Q neemt een een symbool uit $A \cup \{\epsilon\}$ om nieuwe states uit P(Q) te berekenen waarbij P(Q) de powerset is van Q.
- $I \subseteq Q$ is een set van initiële states.
- $F \subseteq Q$ is een set van final states.

Bij een NFA kunnen er dus meerdere initiële states en final states zijn. Bij een overgang van een state kunnen er meerdere nieuwe states actief worden omdat er meerdere overgangen hetzelfde symbool hebben. Een NFA kan ook 'ϵ' overgangen bevatten; 'ϵ' betekent in dit geval de "lege string", een 'ϵ' overgang is een spontane overgang van één state naar een andere state.

Soms is het echter gewenst dat een automaat deterministisch is, dit betekent dat er op ieder moment slechts één actieve state is. Wanneer dit het geval is spreekt men van een *deterministic finite automaton* of DFA. Formeel is een DFA de quintuple $(Q, A, \delta, q0, F)$:

- Q is een set van states.
- A is een eindig input alfabet.
- δ is een functie die een state uit Q neemt een een symbool uit A om een nieuwe state uit Q te berekenen.
- $q0 \in Q$ is de initiële state.
- $F \subseteq Q$ is een set van final states.

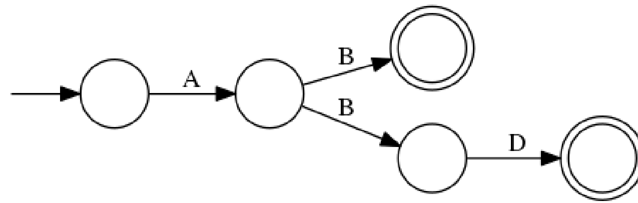
In tegenstelling tot een NFA heeft een DFA slechts één initële state en kan er slechts één nieuwe state actief worden bij een overgang. Een overgang bij een NFA of DFA is een operatie waarbij de huidige state(s) niet langer actief is/zijn, de nieuwe actieve state(s) is/zijn het resultaat van de δ functie, door deze functie toe te passen op de huidige actieve state(s) krijgt men de nieuwe actieve state(s).

Bij het uitvoeren van een automaat neemt men eerst een bepaalde input string en enkel de initiële state(s) van de automaat is/zijn actief. De symbolen in de string worden één voor één geconsumeerd door de automaat, bij ieder symbool worden de overgangen berekend.

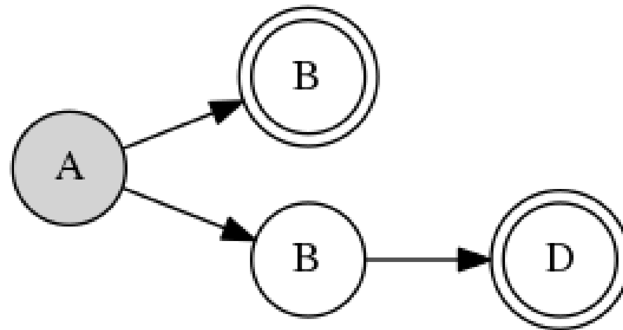
2.1 Automaten volgens de Micron Automata Processor

Op de Micron Automata Processor is het mogelijk om automaten te configureren. Het type automaat op de AP is minstens even sterk als de klassieke NFA die beschreven is in het bovenstaande onderdeel en verschilt enkel op een structureel niveau.

Automaten op de AP worden opgesteld volgens de *Automata Network Markup Language* (ANML). In deze taal zijn er meer elementen beschikbaar dan bij de klassieke NFA, dit zal later nog worden toegelicht. In de ANML taal worden de states en overgangen gecombineerd tot *State Transition Elements* of STE's; het gevolg hiervan is weergegeven in Figuur 2. De twee automaten in Figuur 2 zien er niet hetzelfde uit maar hun werking is hetzelfde. ANML ondersteunt niet direct ϵ overgangen maar de taal is krachtig genoeg om deze impliciet uit te drukken, dit kan door de ϵ closure voor alle states te berekenen [2].



(a) Een klassieke *non deterministic finite state automaton* (NFA). De start state is aangegeven met een half-open pijl. Final states zijn aangegeven met een dubbele rand.



(b) Een ANML automaat. De start state is aangegeven met een donkere achtergrond. Final states zijn aangegeven met een dubbele rand.

Figure 2: Twee equivalente automaten, de automaat in Figuur 2a is een klassieke NFA. De automaat in Figuur 2b is dezelfde automaat maar volgens de ANML structuur.

3 De Micron Automata Processor

De Micron Automata Processor (AP) is een processor met een soort architectuur die anders is dan generieke processoren (CPU) of grafishe processoren (GPU).

Deze processor heeft geen last van de von Neumann bottleneck voor instruction fetching omdat de hardware steeds wordt geherconfigureerd. Er wordt dus geen processing gedaan in de von Neumann wijze, in plaats daarvan komt de processing voort uit het uitlezen van het geheugen van de AP. De AP werkt hiervoor met DRAM technologie. Het verschil met conventionele toepassingen van DRAM technologie is dat een row access niet resulteert in het opvragen van een 'word' maar in match & route operations.

Deze nieuwe architectuur is gebouwd volgens het Multiple Instruction Single Data (MISD) model. Dit betekent dat iedere thread dezelfde data verwerkt maar op een andere manier. Bij de AP komt dit concept tot uiting door meerdere automaten in een automatennetwerk uit te voeren op dezelfde datastream.

Deze nieuwe architectuur staat in contrast met het populaire Single Instruction Multiple Data model; dit model vindt men vooral terug in GPU's maar is ook aanwezig in bepaalde specifieke CPU architecturen.

De AP is gebouwd met als doel efficiënt algoritmes uit te voeren die zijn geformuleerd met Finite State Automata. Het interessante aan deze Finite State Automata is dat deze ook niet-deterministisch kunnen zijn terwijl een symbool nog steeds in één cycle wordt verwerkt.

Op von Neumann architecturen zoals de generieke CPU architectuur of de GPU architectuur kunnen dit soort algoritmes last van state explosion hebben; een moment waarop veel states tegelijk actief zijn, en het veel tijd kost om de volgende states te berekenen. De AP heeft hier geen last van, hierin ligt de kracht van deze nieuwe architectuur.

Het programmeermodel voor deze architectuur is anders dan het programmeermodel voor CPU's of GPU's, dit zorgt ervoor dat het een uitdaging is om algoritmes te ontwikkelen voor de AP zelfs voor ervaren programmeurs. In de volgende delen zal dit programmeermodel worden toegelicht aan de hand van voorbeelden.

3.1 Ontwikkelen van automaten

De automaten die men kan uitvoeren op de AP zijn een uitbreiding op klassieke NFA's. De automaten voor de AP bevatten extra soorten van elementen, deze nieuwe automaten worden gedefiniëerd door de *Automata Network Markup Language* (ANML). Bovenop *State Transition Elements* of STE's heeft men ook boolean elements en counter elements. Deze uitbreidingen dienen vooral om automaten compacter te maken, dit betekent dat ANML automata niet meer kunnen dan klassieke NFA's maar wel met minder states kunnen worden geformuleerd.

Bij het opbouwen van een automaat is het nodig om "reporting" states te definiëren. Reporting states zijn belangrijk voor het genereren van output. Wanneer de automaat in een reporting state terechtkomt dan krijgt men als

output de huidige index van de inputstring. Men kan ook per reporting state een speciale code instellen die wordt weergegeven bij de output om het interpreteren van output te verbeteren.

Om te ontwikkelen voor de AP is er een SDK beschikbaar. Men kan via de SDK automaten opbouwen in een grafische editor of met de C/C++ libraries. Deze C/C++ libraries kan men ook gebruiken via Java of Python bindings. Via de SDK wordt een automaten netwerk samengesteld in Anml; vervolgens wordt de Anml gecompileerd tot een image waarmee te AP wordt geconfigureerd.

Wanneer de AP is ingesteld met een image, kan men een string van symbolen streamen naar de AP. Als uitvoer krijgt men een lijst van momenten dat een reporting state werd geactiveerd.

3.2 Meten van uitvoeringstijden

Er zijn momenteel meerdere modellen van de Micron Automata Processor. Het verschil tussen deze modellen is de manier van aansluiten van de hardware en het aantal Automata Processor chips aanwezig. Het aantal aanwezige chips is een eerste factor voor het meten van uitvoeringstijden.

Het streamen van data gebeurt aan 1 GB/S voor één chip. Men heeft tijdens het instellen van de AP de mogelijkheid om de image te repliceren over meerdere chips; als men dezelfde image configureert over 2 chips dan kan men streamen aan 2 GB/S.

Nog een factor waarmee rekening moet worden gehouden is de tijd die het kost om de outputvectoren van de AP uit te lezen. Deze factor wordt beïnvloed door het aantal reporting states in de automaat; wanneer er veel reporting states worden bereikt zal de outputbuffer snel worden opgevuld met outputvectoren. Indien de outputbuffer vol is moet eerst de output worden uitgelezen van de AP voordat het streamen van de inputstring kan hervatten. De outputbuffer kan momenteel 1024 outputvectoren bevatten. Het uitlezen van een outputvector kan tussen de 91 en 291 symbol cycles liggen naargelang de lengte van de outputvector. De lengte van de outputvector is afhankelijk van het aantal output-regions waar de output van afkomstig is, de programmeur heeft helaas geen controle over welke reporting state in welke output-region wordt gemapt.

De beste strategie is dus om ervoor te zorgen dat het aantal actieve reporting states niet te hoog is per symbol cycle.

3.3 Limieten opgelegd door de hardware

Door de manier waarop de hardware is opgebouwd is er nog een belangrijk aspect waarmee men rekening moet houden bij een strategie voor het ontwikkelen van automaten op de AP. Om te begrijpen waarmee men rekening moet houden, beschouw het hardware model van de AP in Figuur 3. In deze Figuur is het hardware model van één enkele chip weergegeven. Een chip bestaat uit twee half cores, een half core bestaat op zijn beurt uit blocks en block routing lines, in een block is het mogelijk meerdere STE's op de slaan en ofwel een counter ofwel een boolean element. Het is niet enkel nodig om deze elementen te kunnen opslaan

maar ook om te kunnen routeren tussen deze elementen door ze te verbinden. Naargelang men hoger gaat in de hiërarchie is het moeilijker om elementen te verbinden en het is niet mogelijk elementen te verbinden over twee half cores.

Een chip bevat in totaal 49.152 STE's, 768 counter elementen en 2.304 boolean elementen. Indien er de noodzaak is om een zeer grote automaat te configureren op de AP dan is het eventueel niet mogelijk omdat het routeren van de elementen niet mogelijk is. Voor de programmeur betekent dit dus concreet dat men voorzichtiger moet omspringen met verbindingen tussen elementen dan met elementen zelf.

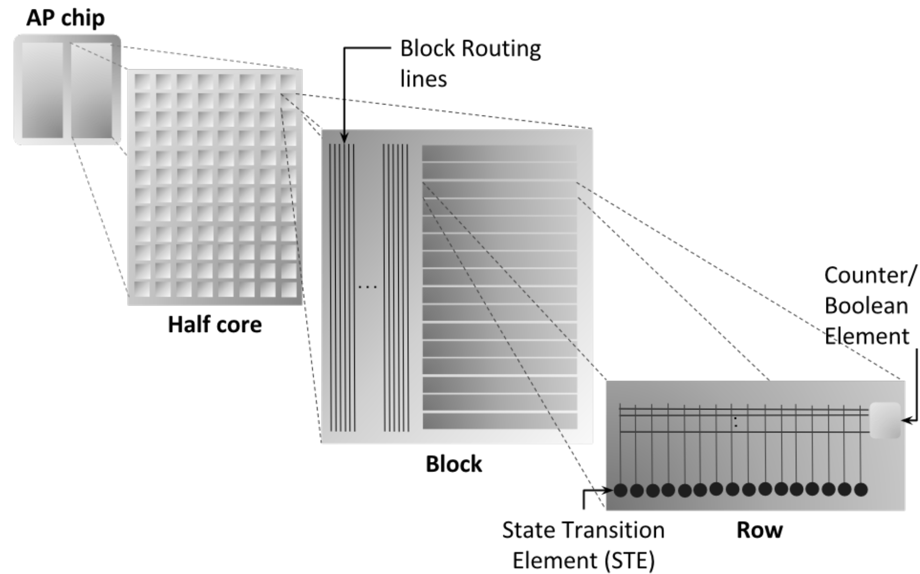


Figure 3: Hardware model van één enkele chip de Micron Automata Processor. Een chip bevat in totaal 49.152 STE's, 768 counter elementen en 2.304 boolean elementen.

3.4 Andere toepassingen

Het is reeds vermeld dat de processor toepassingen heeft in Bioinformatica en Netwerkbeveiliging. Nog een ander vakgebied dat baat heeft bij de AP is Financiën.

In Bioinformatica wordt de AP gebruikt voor **motif search**. Het is een uitdaging om in een grote database van DNA sequences, geconserveerde sequences genaamd motifs te vinden. Deze DNA sequences worden voorgesteld als strings en deze strings worden als input gebruikt voor de AP. De AP is op zijn beurt geprogrammeerd met automaten die bepaalde motifs vinden in een inputstring.

In Netwerkbeveiliging is de AP interessant voor **deep packet inspection**.

In netwerkbeveiliging zijn er signatures van bepaalde attacks en malware bekend. Deze signatures zijn in feite strings die over het netwerk worden verzonden, de AP wordt geprogrammeerd met automaten die deze strings kunnen matchen; zo kan de AP de overeenkomstige attack of malware detecteren.

Het wordt al snel duidelijk dat het belangrijk is om problemen voor te stellen als pattern matching problemen om ze efficiënt op te lossen met de AP. Dit is een uitdaging in het onderzoek omdat afbeeldingen in het 2D domein liggen.

4 Algoritmes voor objectherkenning met automaten

De Micron Automata Processor is ontworpen voor pattern matching. Objecten herkennen in afbeeldingen is analoog aan het matchen van patronen. Het is dan ook een logische stap om te onderzoeken wat de uitdagingen zijn omtrent het zoeken van patronen in afbeeldingen door gebruik te maken van de AP.

4.1 Kennismaking met de tools

Om met de AP te werken is er een SDK beschikbaar. In deze SDK zijn een aantal tools aanwezig die dienen als hulpmiddel voor het ontwikkelen van automatennetwerken die zullen worden geconfigureerd op de AP.

4.1.1 1D hole detection in 1D binaire afbeeldingen

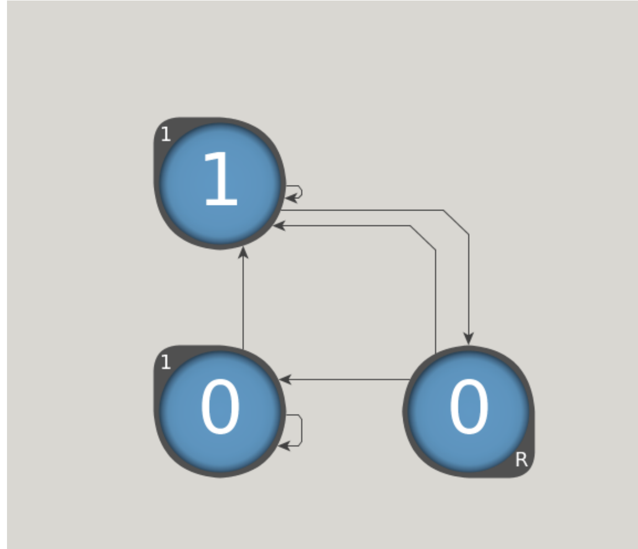
Een 1D binaire afbeelding kan men zien als een 1D reeks van de symbolen '1' en '0'. In dit geval is het dus makkelijk om algoritmes voor te stellen als pattern matching problemen.

Hole detection is het zoeken van gaten in een afbeelding. In een binaire afbeelding betekent dit dat men de witte gebieden, aangeduid met een '1', gaat zoeken. Het algoritme om de witte gebieden in een 1D binaire afbeelding te vinden op een generieke CPU is triviaal; het volgende algoritme dient dus eerder om kennis te maken met het ontwikkelen van algoritmes op de AP.

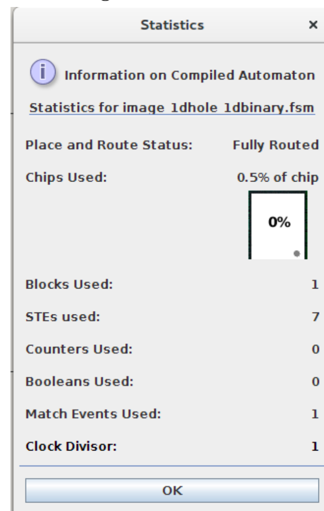
In Figuur 4a zien we de automaat om 1D holes in een 1D binaire afbeelding te vinden. De automaat heeft 2 start elementen uiterst links, dit is omdat het eerste element van een binaire afbeelding '0' of '1' kan zijn. De startelementen worden geactiveerd door het eerste symbool als dit symbool overeenkomt, het is ook mogelijk om een startelement in te stellen dat geactiveerd kan worden bij elk symbool.

Rechts onder is er een reporting symbool '0'. Dit symbool wordt actief nadat er minstens één '1' aan voorafging. Dit symbool wordt dus geactiveerd aan het einde van een hole.

Na het compileren van deze automaat krijgen we de statistieken te zien in Figuur 4b. Dit is een simpele automaat dus het is mogelijk om met 1 AP chip de automaat uit te voeren. Indien we meerdere AP chips ter beschikking zouden



(a) De automaat gemaakt in de AP Workbench.



(b) Compilatie statistieken.

Figure 4: 1D hole detection automaat in een 1D binaire afbeelding.

hebben dan kunnen we de bandbreedte verhogen door de automaat te repliceren over meerdere chips.

Met deze oplossing kunnen we de holes in een 1D binaire afbeelding detecteren; maar de enige informatie die men krijgt van de reporting elements is waar iedere hole eindigt. We kunnen deze implementatie verbeteren door ook een reporting element te configureren aan het begin van de hole. Deze uitgebreide automaat is afgebeeld in Figuur 5.

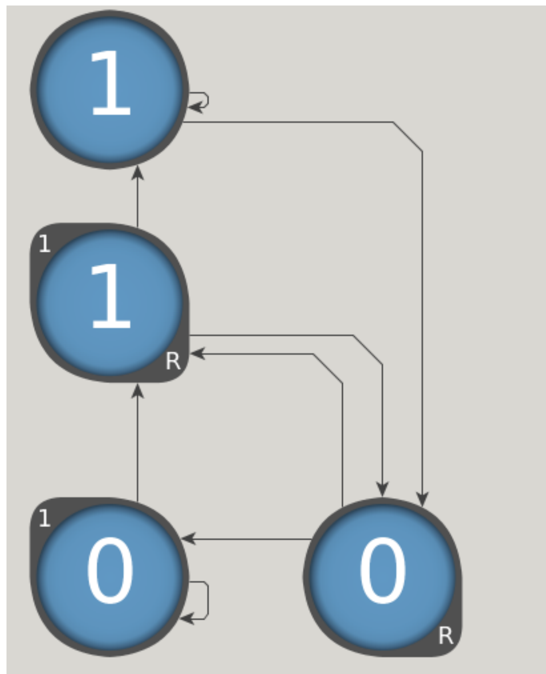


Figure 5: De automaat met extra reporting element.

4.1.2 2D hole in een 2D binaire afbeelding (probleemstelling)

We breiden het vorige probleem uit naar het zoeken van een 2D hole in een 2D binaire afbeelding. Nu we met een 2D afbeelding werken stelt zich het probleem op welke manier we de afbeelding streamen naar de AP. Het is enkel mogelijk om een 1D string van symbolen naar de AP te streamen dus we moeten eerst de afbeelding voorstellen in 1D.

Dit kunnen we doen door de afbeelding rij per rij of kolom per kolom te streamen; we kiezen in dit geval voor rij per rij te streamen. Vervolgens stelt zich een nieuw probleem: het is triviaal om opeenvolgende symbolen met elkaar te vergelijken, maar het is ook nodig om te weten welk symbool “boven” of “onder” het huidige symbool ligt.

Het probleem kunnen we stellen op de volgende manier: “Hoe kunnen we

pattern matching doen in een 2D domein?". Het is nodig om eerst een manier te vinden om dit probleem op te lossen voordat we een algoritme kunnen opstellen voor dit probleem.

4.2 Exact 2D pattern matching

De oplossing voor het probleem "Hoe kunnen we pattern matching doen in een 2D domein?" vinden we in het vakgebied "2D pattern matching". Om dit concept uit te leggen houden we het eerst bij "Exact 2D pattern matching", het zoeken van exacte 2D deelpatronen in een 2D afbeelding.

4.2.1 Algoritme van Baker and Bird

Dit algoritme wordt gebruikt om een 2D deelpatroon eerst te converteren naar een 1D deelpatroon. Eerst wordt een automaat opgesteld volgens het algoritme van Aho Corasick [1].

Het algoritme van Aho Corasick dient om efficiënt woorden te vinden in een text, dit gebeurt door een deterministische automaat op te stellen op basis van een aantal sleutelwoorden; de volledige tekst geeft men als input aan deze automaat, een sleutelwoord is gevonden indien men in een eindstaat terechtkomt.

In deze situatie in het geval van het algoritme van Baker and Bird zijn deze sleutelwoorden ofwel de verzameling van alle rijen ofwel de verzameling van alle kolommen van alle 2D deelpatronen. Hier is gekozen voor alle kolommen te gebruiken.

Wanneer de automaat is opgesteld met alle kolommen van alle 2D deelpatronen kunnen we de 2D deelpatronen reduceren naar 1D deelpatronen. Dit doen we door iedere kolom als input te geven aan de automaat, en de volledige kolom te vervangen met enkel de laatste staat waarin de automaat zich bevindt.

Als laatste rest nog het converteren van de 2D afbeelding. Wederom gebruiken we de automaat waarmee het patroon is gereduceerd naar 1D, als input voor de automaat gebruiken we iedere kolom van de 2D afbeelding. Deze keer vervangen we ieder element van de 2D afbeelding door de huidige staat van de automaat.

4.2.2 Het vinden van letters en cijfers in een binaire afbeelding

Dit onderdeel van het onderzoek is een implementatie van het algoritme van Baker and Bird [1]. Het doel van deze implementatie is het zoeken van patronen in een grotere afbeelding. In dit geval zijn patronen afbeeldingen van letters en cijfers; de 2D afbeelding is een afbeelding van een tekst.

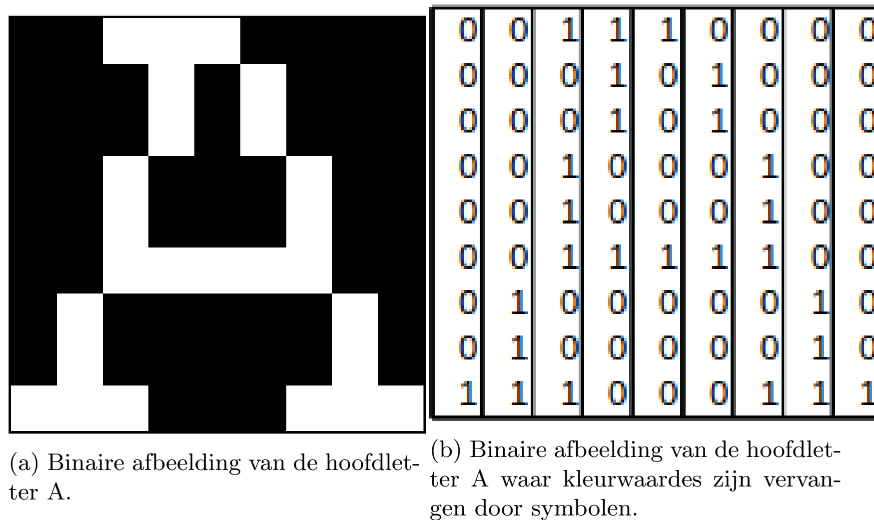
Met deze implementatie kunnen we aantonen hoe men exact pattern matching, compatibel met de AP, kan gebruiken in de context van Beeldverwerking. In deze implementatie beperken we ons tot binaire afbeeldingen. Om dit voorbeeld niet te ingewikkeld te maken zoeken we enkel de letter 'A'.

We bepalen eerst wat de "keywords" zijn die we als invoer gebruiken voor het algoritme van Aho Corasick [1]. Dit zijn de kolommen van alle deelpatronen.

In dit geval is er 1 patroon, weergegeven in Figuur 6a. In Figuur 6b zien we de kolommen van de binaire afbeelding, dit zijn de keywords die we gebruiken als invoer voor het algoritme.

Het resultaat van het algoritme is een deterministische automaat weergegeven in Figuur 7a. De automaat is voorgesteld als een graaf van gewone transitions en een tabel met failure transitions.

De werking van een failure transition gaat als volgt: Stel dat men zich bevindt in state 2; indien het volgende symbool een '0' is gaat men over naar state 3 volgens de normale transition, maar als het volgende symbool een '1' is kan men zich niet beroepen op een normale transition. In dit geval moet het symbool '1' worden geconsumeerd door de failure state van state 2.



(a) Binaire afbeelding van de hoofdletter A.

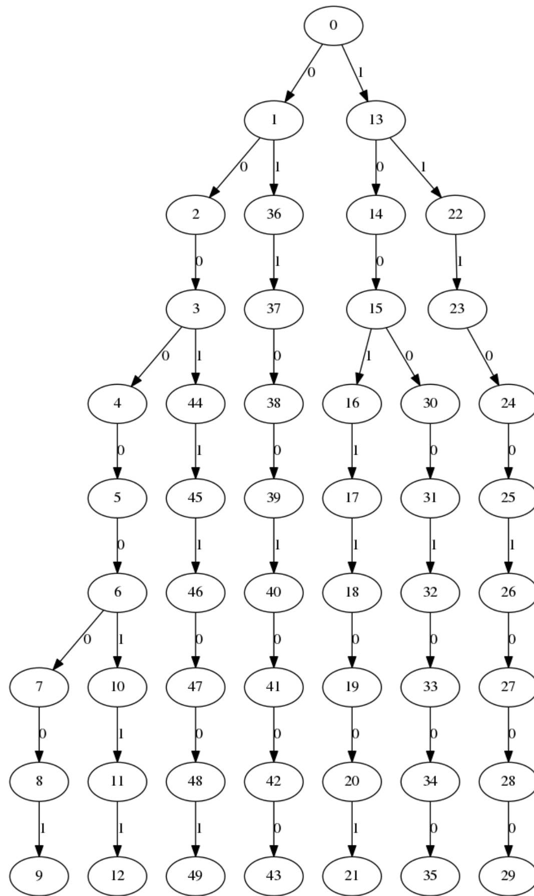
(b) Binaire afbeelding van de hoofdletter A waar kleurwaardes zijn vervangen door symbolen.

Figure 6: Input voor het algoritme van Aho Corasick [1].

Het is belangrijk om op te merken dat deze automaat niet zal worden geconfigureerd op de AP; hiervoor zijn twee redenen; de automaat is deterministisch, een deterministische automaat heeft slechts 1 actieve state en het berekenen van de volgende state kan in constante tijd. Nog een reden is dat we na iedere state transition willen weten wat het huidige state nummer is, indien we dit soort functionaliteit op de AP willen voorzien moet iedere state een reporting state worden en zal het automatennetwerk veel output genereren; dit kan de uitvoertijd op de AP negatief beïnvloeden.

Deze automaat is slechts een hulpmiddel om de afbeelding en het deelpatroon te reduceren naar 1D strings; dit is dan ook de volgende stap.

Met de automaat uit de vorige stap reduceren we de binaire afbeelding van de letter 'A' naar een 1D string. Het resultaat is de 1D string in tabel 8; de symbolen van deze string zijn state numbers uit de automaat. Om deze deelafbeelding terug te vinden in een grotere afbeelding moet de automaat ook worden toegepast op de grotere afbeelding.



(a) Graaf met overgangen van de encoderingsautomaat voor het 2D exact pattern matching voorbeeld.

State	Fail
0	0
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	10
10	44
11	45
12	46
13	0
14	1
15	2
16	36
17	37
18	23
19	24
20	25
21	26
22	13
23	22
24	14
25	15
26	16
27	14
28	15
29	30
30	3
31	4
32	44
33	14
34	15
35	30
36	13
37	22
38	14
39	15
40	16
41	14
42	15
43	30
44	36
45	37
46	23
47	24
48	25
49	26

(b) Tabel met failure overgangen van de encoderingsautomaat voor het 2D exact pattern matching voorbeeld.

Figure 7: Voorstelling van de encoderingsautomaat van het voorbeeld voor 1D exact pattern matching. Iedere state in Figuur 7a heeft een fail overgang die zal worden gebruikt wanneer de huidige state geen overgang heeft voor het volgende symbool. De fail overgang zal dan het volgende symbool comsumeren. De fail overgang voor iedere state is terug te vinden in Figuur 7b.

9	12	21	29	35	43	49	12	9

Figure 8: 1D string verkregen door de automaat in Figuur 7a toe te passen op de kolommen van de binaire afbeelding in Figuur 6b.

De afbeelding waarin we de letter ‘A’ gaan zoeken is weergegeven in Figuur 9. Het huidig algoritme werkt alleen voor exacte patronen; het font en de puntgrootte van de tekst in de afbeelding is daarom hetzelfde als de tekst van de deelafbeelding in Figuur 6a.

Figure 9: Binaire afbeelding waarin een deelafbeelding in zal worden gezocht.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	9	9	9	8	8	8	8	8	8	9	9	9	9	9	9	8	8	8	8	8	8	9	9	9	9	8	9	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	14	11	14	9	8	8	8	8	8	14	11	14	14	14	14	9	8	8	8	8	9	14	14	14	14	9	11	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	15	12	15	11	8	8	8	8	8	15	12	15	15	15	15	11	8	8	8	9	14	15	15	15	15	14	12	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	16	47	30	38	9	8	8	8	8	30	23	30	30	30	30	12	8	8	8	11	15	30	30	30	30	15	47	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	17	48	31	39	11	8	8	8	8	31	23	44	44	44	44	47	8	8	8	12	30	31	31	31	31	30	48	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	18	49	32	40	12	8	8	8	8	5	23	14	14	14	14	36	9	8	8	23	31	5	5	5	5	31	30	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	9	19	27	33	41	47	9	8	8	8	6	23	15	15	15	15	14	11	8	8	23	32	6	6	6	6	5	31	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	11	20	28	34	42	48	11	8	8	8	7	23	30	30	30	30	15	12	8	8	24	45	10	7	7	7	6	32	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	9	12	21	29	35	43	49	12	9	8	8	10	23	44	44	44	44	16	47	8	8	25	38	11	10	10	10	10	33	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	14	47	27	31	31	31	27	47	14	8	8	14	24	14	14	14	14	14	48	8	8	30	39	38	14	14	14	14	34	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	15	48	28	5	5	5	28	48	15	8	8	15	25	15	15	15	15	15	30	8	8	31	30	39	15	15	15	15	35	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	30	30	29	6	6	6	29	30	30	8	8	30	30	30	30	30	30	30	31	8	8	5	31	30	30	30	30	30	31	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	31	31	31	7	7	7	31	31	31	8	8	31	31	31	31	31	31	31	5	8	8	6	5	31	31	31	31	31	5	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	5	5	5	8	8	8	5	5	5	8	8	5	5	5	5	5	5	5	6	8	8	7	6	5	5	5	5	5	6	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	6	6	6	8	8	8	6	6	6	8	8	6	6	6	6	6	6	6	7	8	8	8	7	6	6	6	6	6	7	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	7	7	7	8	8	8	7	7	7	8	8	7	7	7	7	7	7	7	8	8	8	8	8	7	7	7	7	7	8	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8

Figure 10: Geconverteerde waardes van de binaire afbeelding in Figuur 9. De 1D string van Figuur 8 is met een gele kleur weergegeven.

Deze afbeelding wordt niet op dezelfde manier gereduceerd naar 1D. Iedere kolom van de afbeelding is de input voor de automaat, maar alle state numbers worden onthouden in plaats van alleen de laatste state number; dit resultaat is weergegeven in Figuur 10.

Nu rest enkel nog het vinden van de 1D string van de deelafbeelding in de geconverteerde waardes van de grote afbeelding; merk op dat op het probleem nu is gereduceerd naar een 1D pattern matching probleem. In Figuur 10 is de 1D string van de deelafbeelding aangeduid met een gele kleur. Het is dus niet langer nodig een 2D patroon te vinden maar een 1D string die een representatie is van het patroon.

4.2.3 Uitvoering op de Micron AP

Nu het probleem is geformuleerd als pattern matching probleem kan het worden opgelost door de Micron AP.

Eerst moeten alle 1D strings van alle patronen worden geconfigureerd in een automatennetwerk. Voor iedere 1D string maken we een automaat waarvan de start state actief is op ieder symbool van de inputstring; het laatste symbool van de 1D string wordt de reporting state. In dit geval is er slechts één 1D string en zal er dus maar één automaat zijn.

Het automatennetwerk is weergegeven in Figuur 11. Merk op dat state numbers met meerdere cijfers moeten worden opgesplitst in meerdere states want een automaat kan maar 1 symbool tegelijk consumeren. Niets belet de gebruiker echter om de volledige range van ondersteunde symbolen als state numbers te gebruiken in plaats van enkel de symbolen 0-9. Hier worden enkel de symbolen 0-9 gebruikt om het algoritme beter te illustreren. In dit geval zou het mogelijk zijn om numerieke symbolen te definiëren. In Anml is dit mogelijk door het symbool te omgeven met '{}' in plaats van de standaard '[ ]', op deze manier hoeft men getallen tussen 0 en 255 niet op te splitsen.

Zoals eerder vermeld is de start state, links boven, actief bij ieder symbool van de inputstring. De final state, rechts onder, is een reporting state. Wanneer de reporting state actief wordt is het patroon gevonden in de grotere afbeelding. Omdat de inputstring te lang is voor in de AP workbench te simuleren is de command line tool gebruikt voor simulatie.

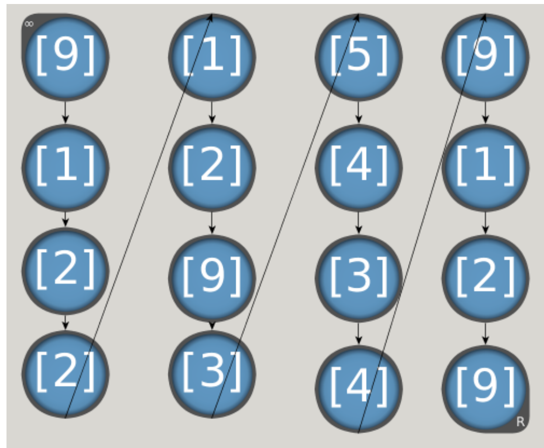


Figure 11: Automaat voor het vinden van de 1D string in Figuur 8, weergegeven in de AP workbench.

De output van de command line tool is weergegeven in Figuur 12. Het patroon is gevonden bij offset 1027 met reporting element __16__. Het einde van de 1D string van de patroon is dus gevonden bij het 1027ste symbool van de inputstring. Indien er meerdere automaten voor patronen zijn kan men met

```
1 apemulate -v -m A_detect_anml.emap A_detect_anml.fsm
      textImage_encoded.txt
2
3 Emulate 1 FSMs
4   Automaton A_detect_anml.fsm
5   Match result:
6   Offset 1027   Reporting element: __16__
7 Total time is 0 seconds
8 Emulation completed.
9
```

Figure 12: Emulatie van de AP met de command line tool *apemulate*.

de identifier __16__ te weten komen van welke automaat het reporting element is. Nog een andere mogelijkheid is om de reporting state een unieke code toe te kennen die het patroon identificeert, deze code zou dan ook bij deze output worden vermeld.

Bij het uitvoeren van de command line emulator worden 3 files als argumenten gegeven. **A_detect_anml.emap** is een file die mappings bevat van state ids op de AP naar state ids zoals in de source Anml; dit is nuttig om de output beter leesbaarder te maken omdat id's van elementen dan overeenkomen met de zelf gekozen id's in plaats van de id's die zijn toegekend tijdens het compileren.

Het bestand **A_detect_anml.fsm** is de binary image die is verkregen door het automatennetwerk te compileren. Ten slotte is er ook **textImage_encoded.txt**; dit is een tekstbestand dat de inputstring bevat die zal worden gestreamd naar de AP. In dit geval bevat dit bestand de afbeelding die is geëncodeerd met de automaat in Figuur 7.

4.2.4 Tijdscomplexiteit

Het doel van het algoritme is om patronen te vinden in een grotere afbeelding, het algoritme is gebaseerd op bestaande technieken en we hebben nu een praktische implementatie van het algoritme die men kan uitvoeren op de AP.

Een overzicht van het algoritme is weergegeven in Figuur 13. De werkwijze van het algoritme is opgesplitst in een "Voorbereiding" en een "Uitvoering" fase. Het algemeen doel van de voorbereidingsfase is om een automatennetwerk op te stellen voor de AP die de 1D patronen zal matchen voor een gegeven inputstring. Dit automatennetwerk kan worden hergebruikt voor het zoeken van patronen in andere afbeeldingen, de enige voorwaarde is dat de afbeeldingen wordt geëncodeerd met dezelfde encoderingsautomaat als de patronen; dit is de reden waarom de stappen uit de voorbereidingsfase niet bijdragen tot de tijdscomplexiteit. In de voorbereidingfase stellen we dus een encoderingsautomaat op en encoderen we hiermee alle deelpatronen.

In de uitvoeringsfase worden de kolommen van een afbeelding geëncodeerd en wordt de afbeelding rij per rij gestreamd naar de AP. Het encoderen van de

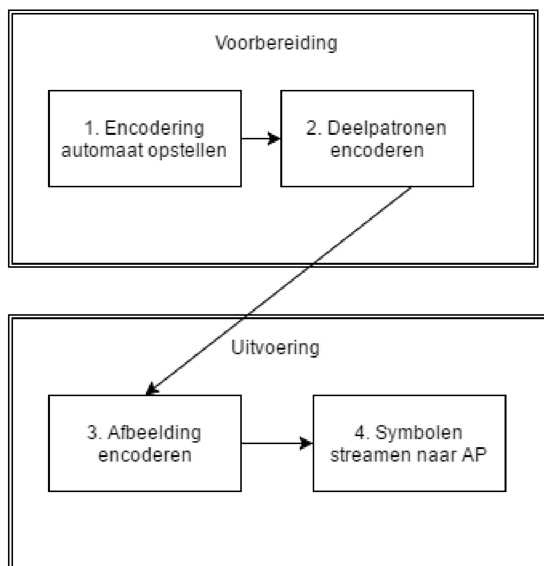


Figure 13: Overzicht van het exact pattern matching algoritme.

afbeelding zien we niet als voorbereiding omdat deze stappen uniek zijn voor iedere afbeelding. De bottleneck van deze implementatie ligt bij het encoderen van de afbeelding omdat de encoderingsautomaat wordt gesimuleerd op een generieke CPU. Het voordeel is dat de encoderingsautomaat deterministisch is, er is slechts 1 state actief en het berekenen van de volgende state werkt in constante tijd. Dus stel dan het aantal kolommen van de afbeelding gelijk is aan m en het aantal rijen gelijk is aan n dan is de tijdscomplexiteit $O(mn)$. Het is mogelijk iedere kolom in parallel te encoderen, indien men iedere kolom parallel encodeert dan is de tijdscomplexiteit $O(n)$.

4.2.5 Conclusie

De implementatie van het algoritme kan efficiënt worden toegepast op de AP want men kan in $O(mn)$, of in $O(n)$ met parallellisatie, 2D patronen vinden in een 2D afbeelding waarbij m de breedte is en n de hoogte is van de afbeelding. Het nadeel van exact pattern matching is dat enkel exacte patronen kunnen worden gevonden in een afbeelding. In het geval van het zoeken van letters en cijfers betekent dit dat het font en de puntgrootte exact hetzelfde moeten zijn; dit kan men echter deels oplossen door ook deelpatronen van dezelfde letters en cijfers van verschillende fonts en puntgrootten te configureren in het automatennetwerk. Stel dat men daadwerkelijk een afbeelding van een bepaalde tekst heeft. Zoals reeds vermeld moet het font en de puntgrootte van de tekst hetzelfde zijn als die van de deelpatronen. Het is ook noodzakelijk dat de afbeelding geen ruis bevat.

Deze strenge voorwaarden maken het algoritme moeilijk bruikbaar in de

praktijk. Het illustreert wel dat het mogelijk is om de AP te gebruiken voor Beeldverwerking.

4.3 Approximate pattern matching

In het vorige deel werd onderzocht hoe exact pattern matching kan worden toegepast op de AP. Er werd geconcludeerd dat de AP hiervoor goed geschikt is mits een aantal voorbereidende fases op de input zoals het opstellen van een encoderingsautomaat en het encoderen van de input. In dit onderdeel van het onderzoek breiden we exact pattern matching uit naar approximate pattern matching. Dit gebeurt door een bepaalde error threshold te introduceren bij het zoeken naar patronen.

De werkwijze voor 2D approximate pattern matching in [3] is analoog aan het exact pattern matching algoritme. Eerst genereren we een encoderingsautomaat waarmee we de gegeven patronen encoderen en waarmee vervolgens ook de grote afbeelding wordt geëncodeerd.

De encoderingsautomaat voor 2D approximate pattern matching is anders dan de encoderingsautomaat van 2D exact pattern matching. hiervoor gebruiken we niet meer de automaat uit het algoritme van Aho Corasick [1]; in plaats daarvan gebruiken we een SFFRCO automaat. SFFRCO is een classificatie voor een automaat die een 1D pattern matching probleem oplost. Het volgende onderdeel bevat informatie over classificaties van automaten voor 1D string matching; deze classificaties zijn belangrijk voor de uitleg van het algoritme. Indien de lezer reeds bekend is met deze classificaties kan het volgende deel worden overgeslaan.

4.3.1 Types automaten voor 1D pattern matching

Het is nodig om bepaalde types automaten voor 1D pattern matching te begrijpen voor het algoritme voor 2D approximate pattern matching.

Alle 1D pattern matching problemen zijn sequentiële problemen; dit wil zeggen dat ze kunnen worden opgelost met automaten. Melichar en Holub [4] hebben een 6 dimensionele classificatie geformuleerd voor iedere soort van 1D pattern matching voor een eindig alfabet. Tijdens het onderzoek naar 2D approximate pattern matching zullen we refereren naar bepaalde automaten volgens deze classificaties. Hieronder is een overzicht van de classificaties:

1. Aard van het patroon: **S**tring of Se**q**uence.
2. Integriteit van het patroon: **F**ull pattern of **S**ubpattern.
3. Aantal patronen: **O**ne, **F**inite number of **I**nfinite number.
4. Manier van matching: **E**xact, Hamming distance **R**, Levenshtein **D** and Damerau **T** distance, approximate matching met Δ **G**, Γ **L** of Δ en Γ distance gecombineerd **H**.

5. Belang van ieder symbool: take **C**are of all symbols, **D**on't take care of some symbols.

6. Aantal instanties van patronen: **O**ne of Finite **S**equence.

Een classificatie van automaat die reeds vermeld is is een SFFRCO automaat. Dit is dus een automaat die meerdere string patronen zal matchen volgens een bepaalde Hamming distance.

De Hamming distance tussen twee strings van dezelfde lengte is het aantal symbolen op dezelfde positie die niet hetzelfde zijn.

4.3.2 Het opstellen van de encoderingsautomaat

Er is reeds vermeld dat de encoderingsautomaat voor 2D approximate pattern matching een SFFRCO automaat is. Dit is een automaat die meerdere strings zal matchen op basis van een inputstring, voor iedere string is er een bepaalde Hamming distance die niet mag worden overschreden; door het introduceren van een Hamming distance wordt het dus mogelijk om fouten toe te laten bij het matchen van strings.

Om een SFFRCO automaat te bouwen moeten we eerst voor iedere string een SFORCO automaat bouwen. Een SFORCO automaat is vergelijkbaar met een SFFRCO automaat, het verschil is dat een SFORCO automaat slecht één string zal matchen.

De strings waarvoor we een SFORCO automaat gaan bouwen zijn dezelfde strings als bij het exact 2D pattern matching algoritme, namelijk de kolommen van ieder 2D patroon. De Hamming distance die we gebruiken voor iedere kolom is 1, er moet dus slechts één symbool overeenkomen om de SFORCO automaat te laten matchen. Dit is een bewuste keuze, het doel van iedere SFORCO automaat is om het aantal foute symbolen te weten te komen bij iedere match. De Hamming distance gebruiken we in dit geval dus niet om het aantal matches te limiteren; dit zal later pas gebeuren in het algoritme.

In dit geval speelt ook het labelen van iedere final state van de automaat een belangrijke rol. We labelen iedere final state van de SFORCO automaten met een tuple van de index van de string en het aantal fouten.

We gebruiken wederom de binaire afbeelding van de letter 'A' in Figuur 6 als voorbeeld voor dit algoritme. De SFORCO automaat van de eerste kolom van de binaire afbeelding A is weergegeven in Figuur 14. Een SFORCO automaat wordt snel groot naargelang de lengte van de string, daarom is enkel de eerste kolom van de binaire afbeelding van de letter A geïllustreerd.

In deze automaat zien we gewone overgangen maar ook overgangen met het label "Fail". De "Fail" overgang wordt gebuikt wanneer in de huidige state het volgende symbool niet kan worden geconsumeerd. De final states zijn weergegeven met een dubbele rand. Wanneer men in een final state terechtkomt kan men via het label van de state te weten komen om welke string het gaat en hoeveel fouten er voorkomen in de match.

De volgende stap is om de verzameling van SFORCO automaten te converteren naar een SFFRCO automaat die we kunnen gebruiken in ons algo-

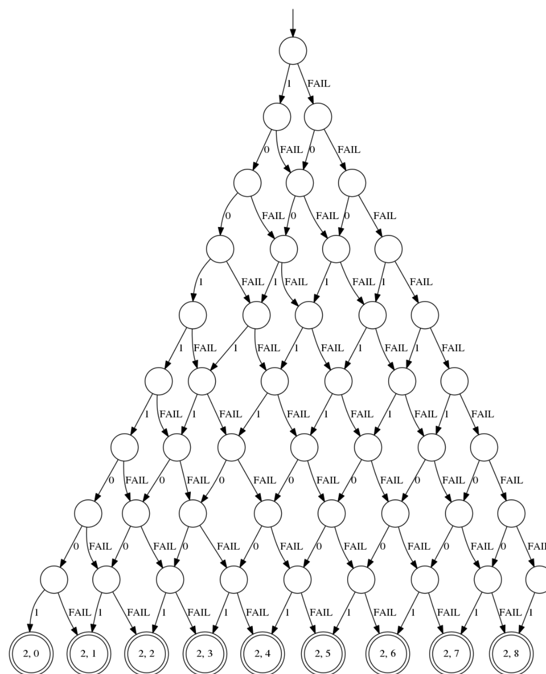


Figure 14: SFORCO automaat van de eerste kolom van binaire afbeelding A in Figuur 6a.

ritme; hieraan zijn twee voorwaardes verbonden. Bij het matchen van een string moet duidelijk zijn van welke kolom de string afkomstig is en hoeveel fouten voorkomen in de string. Dit probleem wordt typisch opgelost door een speciaal label te gebruiken voor iedere final state. Het is interessant om te vermelden dat het voorzien van speciale labels mogelijk is op de AP. In dit geval kunnen we gebruik maken van het feit dat de Micron AP inwendig id's toekent aan iedere state, het is dus al mogelijk om erachter te komen bij welk patroon een bepaalde final state hoort. Nu rest enkel nog het achterhalen van het aantal fouten. Hiervoor kunnen we de extra code gebruiken die we kunnen toekennen aan iedere final state.

Figure 15: SFFRCO automaat van de binaire afbeelding van de letter A in Figuur 6a. De volledige versie van de eerste kolom is weergegeven in Figuur 14

De SFFRCO automaat voor de afbeelding A is weergegeven in Figuur 15. De automaat is weergegeven in een verkleinde versie voor een betere weergave. Zoals eerder vermeld is er een SFORCO automaat voor iedere kolom van het patroon, deze vormen samen de SFFRCO automaat. Bij iedere start state van iedere kolom is een niet-deterministische self loop toegevoegd, deze states zullen dus actief zijn bij ieder inputsymbool. Voor iedere kolom zijn de twee uiterste final states weergegeven. De labels van deze final states zijn van de vorm 'x, y' waarbij x gelijk is aan de identifier voor de kolom en y het aantal fouten in de gevonden match.

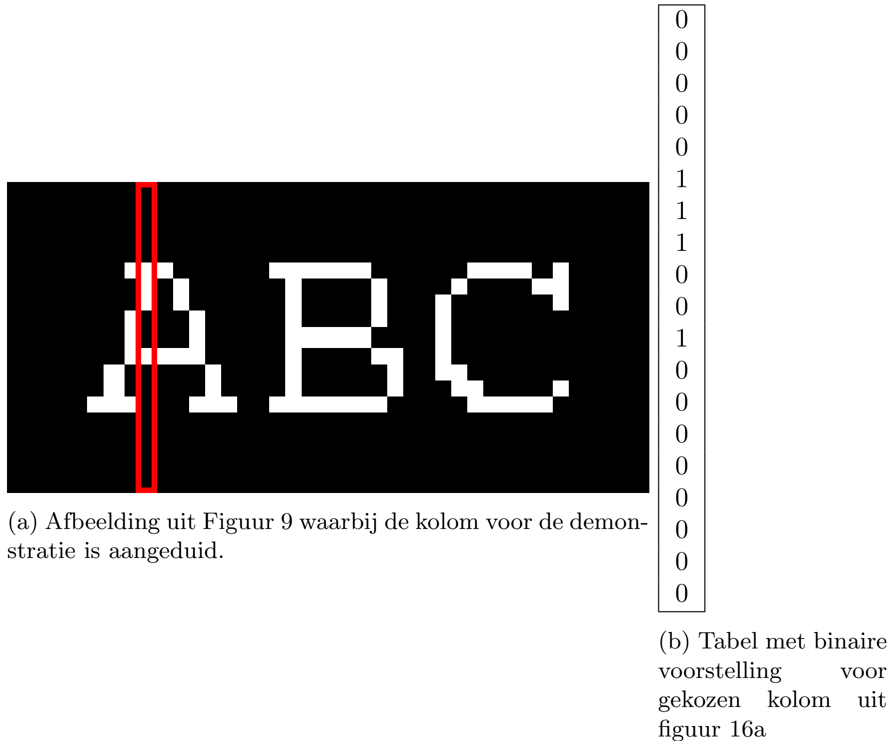
De encoderingsautomaat voor 2D approximate pattern matching en 2D exact pattern matching zijn zeer verschillend. Ter herinnering, de encoderingsautomaat voor 2D exact pattern matching is opgesteld met het algoritme van Aho Corasick [1]. Het resultaat van het algoritme van Aho Corasick [1] is een deterministische automaat, een belangrijke eigenschap is dat in deze automaat iedere string van ieder patroon zit verwerkt. Dit is niet het geval bij 2D approximate pattern matching want ieder patroon heeft zijn eigen SFFRCO automaat en iedere string in de SFFRCO automaat heeft op zijn beurt zijn eigen SFORCO automaat.

4.3.3 Patroon encodering

De volgende stap is om de patronen te encoderen met de encoderingsautomaat. In de encoderingsautomaat hebben we voor iedere aparte kolom een identifier toegewezen, het volstaat om iedere kolom te vervangen door zijn identifier.

4.3.4 Encodering van de afbeelding

In deze stap doorlopen we de grote afbeelding op dezelfde manier als bij exact pattern matching, maar het is nodig om meer informatie bij te houden voor iedere tussenstap. Voor iedere tussenstap is het nodig om voor iedere kolom te onthouden hoeveel foute symbolen er zijn voorgekomen indien er een match is. Er wordt dus een geheel nieuwe dimensie aan data toegevoegd bij het encoderen, dit staat in contrast met het algoritme voor exact pattern matching. Om dit te demonstreren nemen we opnieuw de afbeelding in Figuur 9 als input en encoderen we slecht één kolom met de SFFRCO automaat in Figuur 15. De gekozen kolom is weergegeven in Figuur 16a; in Figuur 16b is deze kolom voorgesteld met de concrete input symbolen voor de SFFRCO automaat.



(a) Afbeelding uit Figuur 9 waarbij de kolom voor de demonstratie is aangeduid.

0
0
0
0
0
1
1
1
0
0
1
0
0
0
0
0
0
0
0

(b) Tabel met binaire voorstelling voor gekozen kolom uit figuur 16a

Figure 16: Input voor demonstratie van encodering met SFFRCO automaat.

De kolom in Figuur 16b wordt doorlopen van boven naar onder en we onthouden bij ieder symbool het aantal fouten voor iedere kolom van ieder patroon indien er een match is. In deze demonstratie hebben we 19 inputsymbolen, er is 1 patroon met 9 kolommen.

Het resultaat van de uitvoer van de SFFRCO automaat en de gekozen kolom is weergegeven in Tabel 1. De tabel moet worden geïnterpreteerd op de volgende manier: in de kolom ‘Symbol’ is weergegeven welk symbool is geconsumeerd, in

<table>
<tr><th>Symbol</th><th colspan="10">Matches</th></tr>
<tr><td>0</td><td colspan="10">∅</td></tr>
<tr><td>0</td><td colspan="10">∅</td></tr>
<tr><td>0</td><td colspan="10">∅</td></tr>
<tr><td>0</td><td colspan="10">∅</td></tr>
<tr><td>0</td><td colspan="10">∅</td></tr>
<tr><td>1</td><td colspan="10">∅</td></tr>
<tr><td>1</td><td colspan="10">∅</td></tr>
<tr><td>1</td><td colspan="10">∅</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>4</td><td>2</td><td>6</td><td>5</td><td>3</td><td>4</td><td>5</td><td>2</td><td>4</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>4</td><td>4</td><td>4</td><td>5</td><td>3</td><td>4</td><td>3</td><td>4</td><td>4</td></tr>
<tr><td rowspan="2">1</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>3</td><td>5</td><td>1</td><td>6</td><td>4</td><td>5</td><td>0</td><td>5</td><td>3</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>5</td><td>5</td><td>5</td><td>6</td><td>6</td><td>5</td><td>4</td><td>5</td><td>5</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>5</td><td>5</td><td>7</td><td>4</td><td>6</td><td>3</td><td>6</td><td>5</td><td>5</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>5</td><td>7</td><td>5</td><td>0</td><td>2</td><td>1</td><td>6</td><td>7</td><td>5</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>4</td><td>6</td><td>4</td><td>3</td><td>3</td><td>4</td><td>5</td><td>6</td><td>4</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>3</td><td>5</td><td>3</td><td>4</td><td>2</td><td>5</td><td>4</td><td>5</td><td>3</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>2</td><td>4</td><td>6</td><td>3</td><td>3</td><td>2</td><td>5</td><td>4</td><td>2</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>2</td><td>4</td><td>6</td><td>3</td><td>3</td><td>2</td><td>5</td><td>4</td><td>2</td></tr>
<tr><td rowspan="2">0</td><td>Kolom</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>Errors</td><td>2</td><td>4</td><td>4</td><td>3</td><td>1</td><td>4</td><td>5</td><td>4</td><td>2</td></tr>
</table>

Table 1: Resultaat van het uitvoeren van de SFFRCO automaat in Figuur 15 met de gekozen input uit Figuur 16b. In de kolom 'Matches' is weergegeven bij welke kolom van het patroon een match is vastgesteld en hoeveel foute symbolen er voorkomen in de match. Indien in de kolom 'Matches' het symbool ∅ staat zijn er geen matches voorgekomen bij het consumeren van het symbool. Perfecte matches zijn gemarkeerd met een gele kleur.

de kolom 'Matches' is weergegeven bij welke kolom van het patroon een match is vastgesteld en hoeveel foute symbolen er voorkomen in de match. Indien in de kolom 'Matches' het symbool $\emptyset$ staat zijn er geen matches voorgekomen bij het consumeren van het symbool.

We merken op dat pas vanaf het negende symbool matches worden gevonden; dit komt omdat iedere kolom van het patroon lengte 9 heeft. Na het negende symbool worden er altijd matches gevonden (in deze situatie). Dit is niet per toeval, om een match te verkrijgen moet minstens 1 symbool overeenkomen, de kans is zeer groot dat dit gebeurt omdat het alfabet enkel bestaat uit twee symbolen; dit is te wijten aan het feit dat we werken met binaire afbeeldingen. De enige situatie waarbij geen match wordt gevonden voor een kolom nadat minstens 9 inputsymbolen geconsumeerd zijn is wanneer de inputstring het inverse is van de kolom. Dit is dus bvb. wanneer de kolom gelijk is aan '101010101' en de input gelijk is aan '010101010'. Deze situatie kan men ook zien in Figuur 14 waar de uiterst rechtse state geen FAIL overgang heeft. Indien we wel een FAIL overgang zouden toewijzen aan deze state door een nieuwe final state toe te voegen met het label '0, 9' dan zou dit betekenen dat we een Hamming distance toelaten die gelijk is aan de lengte van de string, met andere woorden, alle strings zouden worden toegelaten.

Er zijn twee perfecte matches gevonden tijdens de uitvoer, deze zijn in de tabel met een gele kleur gemarkeerd. Er is een perfecte match bij het veertiende symbool met kolom 3, deze match was al verwacht omdat de gekozen kolom overeenkomt met deze kolom uit het patroon. Een andere perfecte match komt eerder voor bij het elfde symbool. Deze match komt voor omdat de laatste 6 symbolen uit kolom 6 een substring zijn van kolom 3 en de eerste 3 symbolen komen uit de zwarte rand rondom de letter.

4.3.5 Uitvoering van het zoeken naar 2D patronen

In de laatste stap van het algoritme gebruiken we output van de SFFRCO automaten (in dit geval slechts één automaat) om de 2D patronen terug te vinden in de grote afbeelding met een bepaalde error threshold. Een geschikte automaat hiervoor is een SFOLCO automaat. Een SFOLCO automaat is een automaat die gelijkaardig is met een SFORCO automaat, het verschil is dat bij een SFOLCO automaat de Levenshtein distance wordt gebruikt bij het matchen van strings in plaats van de Hamming distance. Het verschil tussen de Hamming distance en Levenshtein distance bij automaten ligt bij het input alfabet van de automaten. Bij de Hamming distance heeft men twee overgangen; men kan ofwel het symbool consumeren ofwel niet, in het tweede geval wordt de 'FAIL' overgang gebruikt. Bij de Levenshtein distance heeft iedere state een k overgangen waarbij k een vooraf bepaalde error threshold is; het volgende symbool is altijd een cijfer in [0, k].

De werking van de SFOLCO automaat die gebruikt zal worden in dit algoritme is anders dan wat er tot nu toe aan bod kwam; dit is omdat we niet slechts 1 symbool consumeren maar een tabel van symbolen zoals de tabellen in de matches kolom in Tabel 1. Het idee achter de werking van de SFOLCO

automaat op de output van de SFFRCO automaten is dat er telkens op een niet deterministische manier een symbool wordt gekozen voor iedere actieve state. Met andere woorden, op basis van de input tabel wordt gekozen welke actieve states overgaan naar nieuwe actieve states. Stel dat de states voor kolom 1,4 en 8 actief zijn dan moeten de error waardes voor kolom 2, 5 en 9 in de volgende cycle worden geconsumeerd. De reden voor deze opeenvolging is dat het vereist is om de opeenvolgende kolommen van het patroon met elkaar te vergelijken zodat het patroon kan worden gevonden met de SFOLCO automaat. Bovenop deze functionaliteit zal de errorwaarde van de eerste kolom altijd worden geconsumeerd omdat hier geen kolom aan voorafgaat.

Deze functionaliteit is simpel te implementeren in code, indien we dit willen uitvoeren op de AP dan zal een speciaal mechanisme moeten worden gehanteerd. Dit is echter niet nodig om het algoritme te begrijpen dus dit zal pas worden opgelost bij de implementatie van het algoritme op de AP in sectie 4.3.7.

Bij het opbouwen van de SFOLCO automaat moeten we ook eerst specifiëren wat k is. Bij het uitvoeren van deze automaat moeten we cijfers als input geven in [0, k]; deze cijfers hebben we reeds berekend door het uitvoeren van de SFFRCO automaat in de vorige stap.

Er moet een SFOLCO automaat zijn voor iedere SFFRCO automaat. We hebben nu slechts één SFFRCO automaat dus er is nu slechts één SFOLCO automaat. Indien we meerdere patronen tegelijk zoeken zijn er meerdere SFFRCO automaten, één automaat voor ieder patroon, dus zullen er ook meerdere SFOLCO automaten zijn.

De SFOLCO automaat om de patronen te matchen in de grote afbeelding is weergegeven in Figuur 17. In deze automaat is een error threshold van 2 gekozen om de automaat compact te houden voor weergave maar de belangrijke principes van deze automaat zijn nog steeds aanwezig. Een error threshold van 2 in de SFOLCO automaat wil concreet zeggen dat er maximaal 2 foute symbolen mogen voorkomen tussen de oorspronkelijke 2D voorstelling van het patroon en het gevonden patroon in de grote afbeelding.

In de start state is ook een niet-deterministische self loop toegevoegd zodat de start state actief is op ieder inputsymbool. De automaat is bewust voorgesteld in 3 kolommen, de states in de eerste kolom stellen de situatie voor wanneer er geen fouten zijn. De tweede kolom is de situatie wanneer er al 1 fout is voorgekomen, deze states hebben 1 overgang minder dan de states in de eerste kolom omdat het hier niet meer is toegestaan om 2 fouten te maken omdat men dan de error threshold overschrijdt; indien het symbool 2 toch moet worden geconsumeerd in één van deze states dan is de huidige state niet meer actief en er wordt ook geen nieuwe state geactiveerd. Ten slotte is er de derde kolom, hier zijn al 2 fouten gemaakt en er is alleen een overgang voor 0 fouten over. Dit is strict genomen geen echte SFOLCO automaat omdat niet ieder state k overgangen heeft, maar door de behoefte van deze implementatie is dit niet nodig.

De final states zijn aangegeven met een dubbele rand, indien men hier terechtkomt dan is er een match gevonden. Er is ook een speciaal label toegekend aan deze states dat aantoont hoeveel fouten er aanwezig zijn in de match.

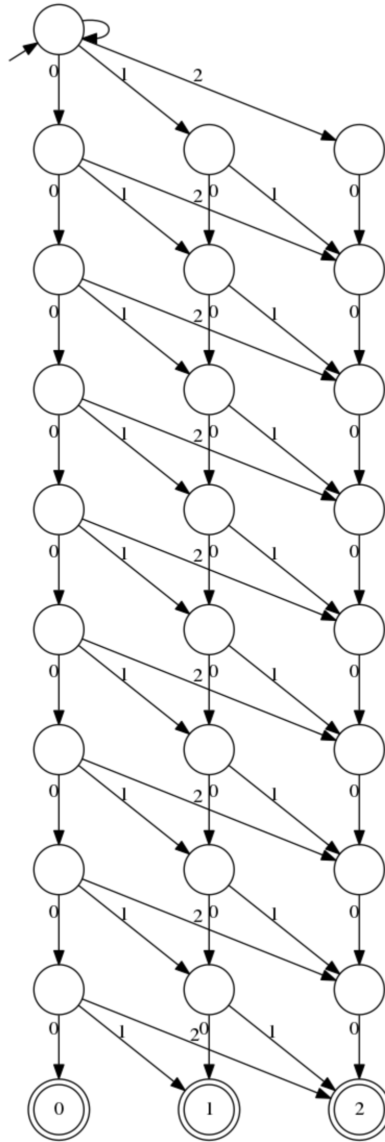


Figure 17: SFOLCO automaat voor een string met lengte 9 met error threshold (k) 2. De string lengte is gelijk aan de hoogte van het patroon.

Nu is het mogelijk om patronen binnen de grote afbeelding te vinden door de tabellen met error waardes van de grote afbeelding rij per rij door te geven aan de SFOLCO automaten.

4.3.6 Samenvatting van het algoritme

Het algoritme voor approximate 2D pattern matching in de context van binaire afbeeldingen is volledig toegelicht in de vorige onderdelen. Dit onderdeel heeft als doel om kort en bondig het algoritme samen te vatten zonder veel aandacht te besteden aan hoe bepaalde keuzes die zijn gemaakt tijdens de uitleg van het algoritme. Hier zal ook duidelijk worden gemaakt wat de gelijkenissen en verschillen zijn met het exact 2D pattern matching algoritme. Een overzicht van de classificaties van 1D string matching automaten is te vinden in 4.3.1.

Eerst moet een encoderingsautomaat worden opgesteld, in dit algoritme is deze automaat een SFFRCO automaat die wordt opgebouwd met SFORCO automaten waarbij de start state van iedere SFORCO automaat een niet-deterministische self loop heeft. De strings die gebruikt worden voor het bouwen van de SFORCO automaten zijn de kolommen van ieder patroon.

In deze SFFRCO automaat heeft iedere final state een speciaal label, met dit label is het mogelijk om te weten te komen hoeveel fouten er aanwezig zijn bij de match; ook is het mogelijk om te weten van welke kolom en van welk patroon de match is gevonden.

De volgende stap is om van ieder patroon een 1D voorstelling te maken door iedere kolom te vervangen door zijn id. Bij exact 2D pattern matching vinden we de id van iedere kolom door de encoderingsautomaat uit te voeren op iedere kolom en de id van de final state te gebruiken, dit hoeft niet bij approximate 2D pattern matching omdat we reeds een unieke id hebben toegekend in de vorige stap.

De voorlaatste stap is het encoderen van de grote afbeelding. In beide algoritmes gebeurt dit door de encoderingsautomaat uit te voeren met de kolommen van de grote afbeelding. Bij exact pattern matching onthouden we in iedere stap slechts de huidige state id van de encoderingsautomaat. Bij approximate pattern matching onthouden we andere informatie, voor iedere kolom onthouden we ofwel niets indien er geen match ofwel onthouden we het aantal fouten van de match als er wel een match is. Voor ieder symbool van de grote inputafbeelding weten we dus voor iedere kolom van de patronen of er een match is, en indien er een match is het aantal fouten.

De laatste stap is het zoeken van de patronen in de grote afbeelding. We hebben een geëncodeerde voorstelling van ieder patroon en van de grote afbeelding. Nu rest nog het opstellen van een automaat die de patronen kan matchen in de geëncoodeerde grote afbeelding. Bij exact pattern matching is dit triviaal omdat de 2D patronen en de grote 2D afbeelding zijn gereduceerd zijn naar 1D; dit is dus een 1D pattern matching probleem geworden.

Bij approximate 2D pattern matching is dit niet het geval. Er wordt een SFOLCO automaat opgesteld voor ieder patroon. In de vorige stap verkregen we een tabel van errorwaardes voor ieder symbool van de grote afbeelding,

een SFOLCO automaat in dit geval consumeert niet één error waarde maar een tabel van errorwaardes, de automaat moet in staat zijn om de enkel de relevante actieve states te laten overgaan naar nieuwe states.

Net zoals in exact pattern matching wordt de grote afbeelding nu rij per rij doorgegeven aan de SFOLCO automaten, wanneer een final state wordt bereikt bij een SFOLCO automaat dan is er een patroon gevonden en het totaal aantal fouten binnen dit patroon.

4.3.7 Implementatie op de Micron AP

Het algoritme van 2D approximate patten matching is nu volledig toegelicht, in tegenstelling tot het algoritme van 2D exact pattern matching is het niet direct duidelijk hoe het mogelijk is om dit algoritme uit te voeren op de AP. In dit onderdeel zal duidelijk worden gemaakt wat precies de problemen zijn bij de implementatie en welke overwegingen kunnen worden gemaakt voor deze problemen.

Probleem 1: Niet deterministische encoderingsautomaat

Het eerste probleem is dat de encoderingsautomaat niet langer deterministisch is. Bij 2D exact pattern matching wordt de encoderingsautomaat opgesteld met het algoritme van Aho Corasick [1] en verkrijgen we een deterministische automaat die efficiënt kan worden gesimuleerd op een CPU om vervolgens de grote afbeelding te encoderen. Bij 2D approximate pattern matching wordt de encoderingsautomaat opgesteld door een SFFRCO automaat op te stellen op basis van de kolommen van ieder patroon. De SFFRCO automaat bestaat uit SFORCO automaten die ieder een niet deterministische self loop hebben op de start state. Dit betekent dus dat de start state van iedere SFORCO automaat actief is bij ieder inputsymbool van de grote afbeelding. De vraag is dus hoe we efficiënt de grote afbeelding kunnen encoderen met de niet deterministische SFFRCO automaat.

Het is mogelijk om niet deterministische automaten te converteren naar deterministische automaten via standard subset construction [5]. Dit kan echter resulteren in automaten die zeer groot zijn naargelang het aantal states in de niet deterministische automaat, namelijk tot $2^{|NFA|}$. Dit is niet praktisch voor SFORCO automaten want een SFORCO automaat groeit sterk naargelang de lengte van de string. Ook als de strings voor de SFORCO automaten niet lang zijn dan zullen overeenkomende substrings er alsnog voor zorgen dat het aantal states hoog is [3]. Het blijft dus nodig om de niet deterministische versie van de automaat te simuleren.

Het lijkt logisch om dit probleem op te lossen door de grote afbeelding te encoderen door de AP te configureren met de SFFRCO automaat, maar dit is helaas ook geen goede werkwijze. Dit is geen goede oplossing omdat de SFFRCO te veel output zal genereren dus het uitvoeren van de automaat op de AP zal dus niet efficiënt zijn; de reden waarom het genereren van veel output inefficiënt is, is beschreven in 3.2. Dit zien we bij de output van de demonstratie in Tabel

1, bij ieder symbool behalve de eerste 8 wordt steeds een match vastgesteld voor iedere kolom.

Er is nu vastgesteld dat de niet deterministische versie van de automaat niet efficiënt kan worden gesimuleerd op de AP en dat de deterministische versie van de automaat te groot zal zijn om te simuleren op de CPU. De niet deterministische versie zal dus op een andere manier moeten worden gesimuleerd.

Voor de implementatie in dit onderzoek wordt de automaat gesimuleerd met de CPU. De CPU zal dus voor veel actieve states de volgende states moeten berekenen, omdet er enkel een niet deterministische self loop is bij de start state zal er steeds één of geen nieuwe state worden berekend voor iedere actieve state, er zal dus geen state explosion optreden. Door de simplistische structuur van de SFORCO automaat is het ook mogelijk om het maximum aantal actieve states te berekenen; dit is gelijk aan het aantal “lagen” van de automaat, het aantal lagen van een SFORCO automaat is gelijk aan de lengte van de string + 1; de SFORCO automaat in Figuur 14 bijvoorbeeld heeft 10 lagen. De SFFRCO automaat in Figuur 15 heeft 9 SFORCO automaten, op iedere gegeven moment zullen er dus worst case 90 nieuwe states moeten worden berekend. Het berekenen van nieuwe states kan volledig parallel gebeuren. Een voorbeeld van de code voor een SFORCO automaat te simuleren is weergegeven in Listing 1

```
1  activeStates = []
2  newActiveStates = []
3
4  for symbol in inputString:
5    #initial state is always active
6    activeStates.append(initialState)
7
8    for state in activeStates:
9      if symbol in state.NextStates:
10       nextState = state.NextStates[symbol]
11       if nextState in FinalStates:
12         reportMatch(nextState)
13       else:
14         newActiveStates.append(state.NextStates[symbol)
15
16   activeStates = newActiveStates
17   newActiveStates = []
```

Listing 1: Pseudo code in Python om een automaat te simuleren met een niet deterministische self loop op de CPU.

Probleem 2: Simuleren van de SFOLCO automaat op de AP

In de laatste stap van het algoritme wordt een SFOLCO automaat opgesteld. De input voor deze automaat is een tabel van error waardes voor ieder symbool van de grote afbeelding, een SFOLCO automaat zoals beschreven in het algoritme consumeert niet één error waarde tegelijk maar een tabel van errorwaardes, de automaat moet in staat zijn om enkel de relevante actieve states te laten overgaan naar nieuwe states. Het probleem is dat het niet mogelijk is met de AP om te kiezen welke actieve states ‘relevant’ zijn.

Het is niet mogelijk om direct op deze manier automaten uit te voeren op de AP dus er moet een techniek worden gevonden die wel compatibel is met de AP en die equivalent is met het beschreven algoritme.

In code is het makkelijker om deze functionaliteit te verkrijgen. In listing 2 is de code weergegeven waarmee het mogelijk is om de SFOLCO automaat op de beschreven manier uit te voeren. Merk op dat deze weinig verschilt van de code in listing 1. Het meest essentiële verschil ligt bij het bijhouden van actieve states, bij de SFORCO simulator code houden we slechts referenties naar states bij terwijl bij de SFOLCO simulator code we een tuple bijhouden van een state en een index. De index in de tuple wordt gebruikt om te beslissen welk symbool in de huidige tabel het relevante symbool is.

```
1  activeStates = []
2  newActiveStates = []
3
4  for table in inputTables:
5    #initial state is always active
6    activeStates.append((initialState, 0))
7
8    for state,tableIndex in activeStates:
9      symbol = table[tableIndex]
10     if symbol in state.NextStates:
11       nextState = state.NextStates[symbol]
12       if nextState in FinalStates:
13         reportMatch(nextState)
14       else:
15         newActiveStates.append((nextState, tableIndex+1))
16
17   activeStates = newActiveStates
18   newActiveStates = []
```

Listing 2: Pseudo code in Python om een SFOLCO automaat te simuleren die op een niet deterministische manier een tabel van symbolen consumeert in ieder cyclus. Het algoritme dat is weergegeven is gebaseerd op algoritme 4.6 uit [3]

Een techniek die men kan gebruiken is door de structuur van de SFOLCO automaat aan te passen en een vaste volgorde van kolommen aan te houden tijdens het streamen naar de AP. De grote afbeelding wordt rij per rij gestreamd. Ter herinnering, een element van de grote afbeelding is een tabel van error waardes. De tabellen worden van links naar rechts gestreamd. Indien er geen match is streamen we het symbool ‘x’.

Vlak voor iedere state, behalve de final states voegen we een serie van ‘dummy states’ toe. Een ‘dummy state’ zal altijd overgaan naar de volgende ‘dummy state’ ongeacht het symbool.

Neem weer het voorbeeld in Tabel 1, we hebben 9 kolommen dus iedere reeks van ‘dummy states’, behalve de self loop in de start state, zal gelijk zijn aan 9, de reeks van ‘dummy states’ in de self loop van de start state is gelijk aan 8. De reden voor deze getallen wordt duidelijk na het illustreren van de techniek. Om te techniek te illustreren wordt de flow van een automaat beschreven met tokens, hierdoor zal de essentie van de techniek duidelijker worden. Een state is actief wanneer er zich een token bevindt. Initiëel is er een token aanwezig in

de start state.

Wanneer we de eerste error waarde voor kolom 0 consumeren dan splitst de token op in twee tokens door de niet deterministische overgang. Het eerste token verplaatst zich van de start state naar een eerste 'dummy state', dit token noemen we 'token0'. Het tweede token wordt verplaatst naar de eerste 'dummy state' van de self loop van de start state, dit token noemen we 'token1'. Bij het consumeren van de error waarde van kolom 1 worden de twee tokens verplaatst, 'token0' en 'token1' verplaatsen naar hun respectievelijke volgende 'dummy state'. We gaan verder met de rest van de kolommen tot de eerste tabel van error waardes volledig is geconsumeerd.

We merken op dat de tokens nu 8 keer verplaatst zijn. 'token1' bevindt zich nu terug in de start state en 'token0' zit in de laatste 'dummy state' van de reeks. Nu consumeren we de de error waarde voor kolom 0 in de tweede tabel, 'token1' wordt gesplitst zoals beschreven in het vorige deel en 'token0' bevindt zich terug in een 'echte' state, dit is gewenst want het volgende token is de error waarde voor kolom 1 in tabel 2. De volgende keer dat 'token0' zich in een 'echte' state bevindt dan is het volgende symbool de error waarde van kolom 2 in tabel 3. Met andere woorden, de flow die 'token0' beschrijft is het matchen van een patroon beginnend bij kolom 0 uit de eerste tabel.

De beschreven techniek werkt op de AP omdat nu de errorwaardes één voor één kunnen worden gestreamd in plaats dat een hele tabel moet kunnen worden gesconsumeerd. Een nadeel is dat het totaal aantal 'dummy states' die worden toegevoegd snel zeer groot kan worden. het totaal aantal 'dummy states' is gelijk aan

$$[(|SFOLCO| - |finalstates|) * |columns|] - 1 \tag{1}$$

waarbij

$$|SFOLCO| = 1 + \sum_{i=0}^{k}(|columns| - \lfloor\frac{|i-1|}{|strings|}\rfloor)[3]$$

$$|finalstates| = k + 1$$

. In het voorbeeld is het aantal 'dummy states' gelijk aan $[(28 - 3) * 9] - 1 = 224$ dit is een groot aantal nieuwe states aangezien de automaat oorspronkelijk 28 states bevat.

De techniek is toegelicht, deze is equivalent met de theoretische techniek en het is mogelijk om deze techniek uit te voeren op de AP. Deze techniek schaalt echter slecht naargelang de breedte van patronen en de error threshold, de AP heeft immers een limiet aan het aantal STE's, namelijk 49.152 per chip. De 'dummy states' hebben echter slechts 1 uitgaande overgang en dit zorgt ervoor dat het routeren van de 'dummy states' goed mogelijk blijft op de AP, de uitleg over het routeren van elementen op de AP kwam aan bod in sectie 3.3. Daarom wordt een alternatieve techniek toegelicht die niet vereist dat de structuur van de SFOLCO automaat wordt veranderd.

De alternatieve techniek is gebaseerd op het belangrijkste aspect van de vorige techniek. Het belangrijkste bij de vorige techniek was dat wanneer 'token0' terug in een 'echte' state terechtkwam dat het volgende symbool de error waarde is van de volgende kolom uit de volgende tabel. Wat hiermee duidelijk wordt is dat dit probleem eerder een kwestie is van de volgorde van symbolen in plaats van de structuur van de SFOLCO automaat. Zoals eerder was vermeld hebben we geen controle over welke states 'relevant' zijn, maar we hebben wel controle over de volgorde waarin we symbolen streamen.

Deze techniek bestaat er dus in om de error waardes van al de error tabellen uit de geëncodeerde afbeelding te ordenen zodat de 'dummy states' niet langer nodig zijn, zonder 'dummy states' wordt het mogelijk om grotere patronen te gebruiken voor het algoritme en de tijd die nodig is om een automaat te compileren wordt verminderd. We illustreren de techniek weer met het voorbeeld in Tabel 1. In de vorige situatie zouden de tabellen rij per rij worden gestreamd naar de AP, maar zonder 'dummy states' te gebruiken zouden de opeenvolgende waardes van de tabellen met elkaar worden vergeleken en dit is niet gewenst.

Stel dat het eerste symbool van de eerste tabel wordt gestreamd, het volgende symbool moet de tweede waarde van de tweede tabel zijn zoals bij de uitleg met het voorbeeld van de tokens. Vervolgens moet de derde waarde van de derde tabel worden gestreamd en dit gaat verder tot alle kolommen van het patroon zijn vergeleken.

Hetgeen wat er precies gebeurde in de bovenstaande uitleg is dat het patroon werd gezocht beginnend bij de eerste tabel met error waardes. De volgende stap is dus om het patroon te zoeken beginnend bij de tweede error tabel. Wanneer het patroon is gezocht beginnend bij iedere error tabel van de geëncodeerde afbeelding dan is het zoeken naar het patroon in de afbeelding voltooid. Merk op dat bij het doorlopen van de geëncodeerde afbeelding sommige symbolen niet voorkomen, dit zijn de symbolen die nooit relevant zijn; een voorbeeld van dit soort symbolen zijn alle waardes van de eerste error tabel behalve de eerste.

```
1 symbolAmount = len(encodedImage)
2
3 for tableStartIndex in range(0, symbolAmount, patternWidth):
4   for columnIndex in range(0, patternWidth):
5     symbolIndex = tableStartIndex + columnIndex * (patternWidth+1)
6       if symbolIndex<symbolAmount:
7         symbol = encodedImage[symbolIndex]
8         Stream(symbol)
```

Listing 3: Pseudo code in Python om de volgorde van symbolen aan te passen om het gebruikt van 'dummy states' te vermijden. De error tabellen zijn aaneengesloten opgeslagen in een lineaire structuur door rij per rij de geëncodeerde afbeelding te doorlopen.

Het algoritme voor deze techniek is weergegeven in Listing 3. Nu we een manier hebben om de lineaire structuur te streamen naar de AP zodat de 'dummy states' niet langer nodig zijn lijkt dit op het eerste zicht een goede oplossing voor het probleem. Het moet nog steeds mogelijk zijn om op basis van informatie van de reporting state te weten te komen wat de positie is van

het patroon wat is gevonden. Dit werd eerder gedaan door naar de index te kijken van het symbool waar de match zich voordeed; omdat we eerder steeds de grote afbeelding rij per rij streamden was het triviaal om met deze index te weten waar het patroon zich bevind in 2D.

Door de volgorde aan te passen kunnen we niet langer op dezelfde manier deze positie achterhalen van het patroon in de afbeelding maar het blijft nog steeds mogelijk. Het is mogelijk om te achterhalen uit welke error tabel de match komt met $tableIndex = \lfloor \frac{matchPos}{patternWidth} \rfloor$, nu de positie van de error tabel gekend is kan op dezelfde manier als bij het rij per rij streamen achterhaald worden wat de positie is binnen de afbeelding.

Probleem 3: wat precies te configureren op de AP bij meerdere patronen?

Om één patroon te vinden in de geëncodeerde afbeelding zal een SFOLCO automaat worden geconfigureerd op de AP voor het algoritme van 2D approximate pattern matching, er is nu wel een methode gevonden om een SFOLCO automaat te simuleren op de AP maar wat als er meerdere patronen en dus meerdere SFOLCO automaten zijn? Bij het algoritme van 2D exact pattern matching was het mogelijk om simpelweg een aantal automaten te configureren in hetzelfde automatennetwerk die bepaalde 1D strings matchen, maar nu is het nodig om meerdere SFOLCO automaten te simuleren die ieder een bepaalde volgorde van kolommen aanhouden voor ieder patroon.

Stel dat het doel is om iedere SFOLCO automaat te configureren op de AP in één automatennetwerk; het is eventueel mogelijk om opnieuw een systeem gelijkaardig aan het systeem met de 'dummy states' op te stellen zodat de timing van ieder symbool juist is voor iedere SFOLCO automaat. Dit zou nog meer 'dummy states' introduceren in de automaten en zou ook slecht schalen naargelang het aantal patronen.

De AP heeft in dit geval een belangrijke eigenschap die is uit te buiten om dit probleem op te lossen, namelijk dat het herconfigureren van de AP met een nieuw automatennetwerk snel gebeurt indien dit automatennetwerk reeds is gecompileerd. Een oplossing die geen nieuwe 'dummy states' introduceert is dus om de AP voor ieder patroon de herconfigureren en vervolgens de grote afbeelding te streamen die is geëncodeerd met enkel de SFFRCO automaat van dat patroon.

De vraag is dan of dit geen negatieve invloed heeft op de efficiëntie van het algoritme omdat de grote afbeelding meerdere keren zal moeten worden geëncodeerd en de data meerdere keren moet worden gestreamd. In sectie 4.3.2 is aangehaald dat de SFFRCO automaten van ieder patroon volledig losstaan van elkaar, er is dus geen voordeel bij het encoderen van de grote afbeelding wanneer dit gelijktijdig gebeurt voor alle patronen of wanneer het encoderen apart gebeurt. Op het gebied van encoderen heeft dit dus geen negatieve invloed. Op het gebied van data die moet worden gestreamd heeft dit ook geen negatieve invloed want het aantal symbolen in de geëncodeerde afbeelding (n x n') is gelijk aan $(m_1 + m_2 + m_3 + ...)nn'$ waarbij m gelijk is aan het aantal kolommen van

een patroon. Indien men de grote afbeelding apart per patroon encodeert is het aantal symbolen gelijk aan $m_1nn'+m_2nn'+m_3nn'+... = (m_1+m_2+m_3+...)nn'$.

De oplossing is dus om de geëncodeerde afbeelding voor ieder patroon apart te encoderen en apart te streamen naar de AP. Op deze manier worden onnodige complexe automaten vermeden terwijl dit nog steeds een efficiënte oplossing is.

4.3.8 Tijdscomplexiteit

Het algoritme van 2D approximate pattern matching kan net zoals 2D exact pattern matching in twee onderdelen worden opgesplitst. Eerst is er een voorbereidende fase waarin een encoderingsautomaat wordt opgesteld en waarin ieder patroon wordt gereduceerd naar een 1D string. In de volgende fase moet de grote afbeelding worden geëncodeerd en moet ieder geëncodeerd element van de grote afbeelding worden gestreamd naar de AP.

Ook is de weer de voorbereidende fase niet uniek voor iedere uitvoer van het algoritme dus is het alleen interessant om te kijken naar de tijdscomplexiteit van de uitvoerende fase. Het encoderen van de grote afbeelding is een probleem dat is toegelicht in sectie 4.3.7; bij dit probleem is er geconcludeerd dat het encoderen zal gebeuren door de niet deterministische encoderingsautomaat te simuleren op de CPU. Hier ligt dus net zoals bij 2D exact pattern matching de bottleneck voor het algoritme.

Volgens [3] is de tijdscomplexiteit voor het simuleren van de encoderingsautomaat $O(mm'nn')$, waarbij m de lengte is van iedere string (of de hoogte van het patroon), m' is het aantal strings (of de breedte van het patroon), n is de hoogte van de grote afbeelding en n' de breedte.

Het matchen van de patronen werkt wel op de AP. De automaat die wordt geconfigureerd op de AP is een SFOLCO automaat, maar om de speciale manier om deze uit te voeren compatibel te maken op de AP moet deze ofwel eerst structureel worden aangepast door 'dummy states' toe te voegen ofwel moet de volgorde van de symbolen van de geëncodeerde afbeelding worden aangepast. Het toevoegen van deze 'dummy states' en precies de reden waarom het nodig is deze al dan niet toe te voegen is beschreven in 4.3.7.

Vervolgens moet ieder element van de grote afbeelding worden gestreamd naar de AP, dit betekent dat mnn' symbolen naar de AP worden gestreamd voor ieder patroon in tegenstelling tot nn' symbolen é'enmalig bij 2D exact pattern matching.

4.3.9 Conclusie

Het algoritme van 2D approximate pattern matching is uitvoerbaar op de AP maar slechts bepaalde onderdelen van het algoritme worden efficiënt uitgevoerd. Bij dit algoritme is de fase van het encoderen van de grote afbeelding een bottleneck ook al is deze fase geformuleerd met het uitvoeren van automaten.

Dit algoritme heeft als voordeel dat patronen kunnen worden gevonden ook als ze niet exact voorkomen in een grote afbeelding. In het geval van het vinden van letters en cijfers in een afbeelding betekent dit dat elementen zoals ruis en

aliasing aanwezig kunnen zijn en de correcte patronen nog steeds kunnen worden gevonden omdat er een bepaalde error threshold is. Het font en de puntgrootte van de letters en cijfers hoeven dus niet langer exact hetzelfde te zijn en hoeven slechts gelijkaardig te zijn. Het nadeel van deze threshold is dat het nu ook mogelijk wordt om false positives te matchen.

Het algoritme van 2D approximate pattern matching verzacht de voorwaardes die zijn gesteld door het algoritme van 2D exact pattern matching. Dit is ten koste van de efficiëntie van het encoderen van de grote afbeelding; er moeten ook meer symbolen worden gestreamd naar de AP.

4.4 2D hole detection in 2D binaire afbeeldingen (deel 2)

Voordat het algoritme van 2D exact pattern matching en 2D approximate pattern matching aan bod kwamen was in sectie 4.1.2 de probleemstelling voor 2D hole detection in 2D binaire afbeeldingen toegelicht. Dit onderdeel heeft als doel de 2D hole detection in 2D binaire afbeeldingen terug opnieuw te onderzoeken en een methode te vinden om dit probleem op te lossen op basis van algoritmes die reeds zijn toegelicht. Verder zal naar dit algoritme verwezen worden simpelweg als “hole detection”.

In de probleemstelling werden de volgende vragen gesteld: hoe kunnen we een 2D afbeelding voorstellen in 1D en hoe weten we welk symbool “boven” of “onder” het huidige symbool ligt? Deze twee vragen komen in principe op hetzelfde neer, een meer algemene vraag is: hoe kunnen we een afbeelding voorstellen in 1D zonder dat nodige spatiäle informatie verloren gaat?

Deze vraag wordt beantwoord in 2D exact pattern matching en 2D approximate pattern matching door een encoderingsautomaat op te stellen en deze te gebruiken om op een bepaalde manier dimensionale reductie toe te passen. Het type automaat en de output van deze automaten is in beide algoritmes anders, maar ze hebben beide als doel dimensionele reductie toe te passen op de grote afbeelding. De strategie in dit onderdeel van het onderzoek is om hetzelfde principe toe te passen voor het probleem van hole detection.

De vraag is nu hoe precies de dimensionale reductie toe te passen; een groot verschil met de vorige algoritmes is dat ‘holes’ een variabele breedte en hoogte hebben. Bij de vorige algoritmes worden er steeds patronen gezocht in een grote afbeelding waarbij de patronen een vaste breedte en hoogte hebben. Een hole is dus eerder een abstracte beschrijving van een patroon terwijl bij vorige algoritmes patronen steeds vast waren gedefiniëerd.

De encoderingsautomaat bij 2D exact pattern matching zorgt ervoor dat iedere kolom van een patroon een unieke identifier krijgt, op basis van deze unieke identifier is het vervolgens mogelijk iedere kolom te reduceren naar één symbool door de kolom te vervangen door zijn identifier. Op deze manier wordt een reeks van kolommen van een patroon een 1D string zonder dat er spatiäle informatie verloren gaat. De grote afbeelding wordt met dezelfde encoderingsautomaat geëncodeerd door ieder element te vervangen met een label uit de encoderingsautomaat, vervolgens wordt de geëncodeerde afbeelding rij per rij gestreamd. Bij ieder symbool in de grote afbeelding weet men dus welk sym-

bool “boven” het huidige symbool ligt omdat het huidige symbool een unieke identifier is van een kolom.

Voor het hole detection algoritme stellen we dus de vraag welke informatie nodig is bij ieder symbool van de geëncodeerde grote afbeelding. Dit is afhankelijk van de volgorde waarin de grote afbeelding wordt gestreamd naar de AP maar ook wanneer precies een match moet worden vastgesteld. Voor dit voorbeeld streamen we de afbeelding rij per rij zoals bij de vorige algoritmes en we stellen een match vast op het uiterst rechtse symbool onderaan een hole. De match beschrijft dus alleen ruwweg de positie van een ‘hole’. Het is dus nuttig om te weten bij ieder symbool wat het symbool “onder” het huidige symbool is, want dan is het mogelijk vast te stellen of men onderaan een hole zit of niet.

De strategie is dus om de grote afbeelding zodanig te encoderen dat men bij ieder symbool weet wat het onderliggende symbool is. Er kunnen 4 situaties voorkomen die we ieder een label kunnen geven:

1. Het huidige symbool is een ‘0’ (symbool buiten een hole) en het symbool er onder is ook een ’0’.
2. Het huidige symbool is een ‘0’ en het symbool er onder is een ‘1’.
3. Het huidige symbool is een ‘0’ en dit symbool ligt op de onderste rij van de afbeelding.
4. Het huidige symbool is een ‘1’ en het symbool er onder is een ‘0’.
5. Het huidige symbool is een ‘1’ en het symbool er onder is een ‘1’.
6. Het huidige symbool is een ‘1’ en dit symbool ligt op de onderste rij van de afbeelding.

Nu moet een keuze worden gemaakt welke situaties nodig zijn om te encoderen in de grote afbeelding, de grote afbeelding wordt kolom per kolom van boven naar onder geëncodeerd en wordt vervolgens rij per rij gestreamd naar de AP. Tijdens het rij per rij streamen willen we weten of we aan de onderkant van een hole zitten en of we aan het uiterst rechts symbool zitten. In situatie 4 en 6 zit men aan de onderkant van een hole, deze situaties moeten dus voorkomen in de geëncodeerde afbeelding. Ook situatie 5 is belangrijk omdat we dan kunnen uitsluiten dat we op de rand van een hole zitten. Op de onderste rij van de afbeelding nemen we aan dat het symbool onder het huidige symbool gelijk is aan ‘0’, situatie 1 en 3 en situatie 6 en 4 zijn dan hetzelfde.

Op basis van de bovenstaande keuzes onderscheiden we 3 klassen van situaties die we willen encoderen in de grote afbeelding; ofwel hebben we situatie 1, 2 of 3, ofwel hebben we situatie 4 of 6, ofwel hebben we situatie 5. We geven deze klasses respectievelijk de labels ‘0’, ‘1’ en ‘2’.

Het is nu bepaald wat het resultaat moet zijn van de encodering, namelijk dat ieder symbool van de grote afbeelding moet worden vervangen door het label van één van de klassen die bovenstaand zijn beschreven. Nu zal deze methode worden getest, in Figuur 18 staat de volledige testsituatie geïllustreerd. Om de

(a) Afbeelding die als voorbeeld dient voor hole detection in sectie 4.4.

(b) Regio in Figuur 18a aangeduid met een rode kader.

```
2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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 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 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 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 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 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 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 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 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 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 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(c) Geëncodeerde versie van de afbeelding in Figuur 18b

Figure 18: Voorbeelddata voor hole detection.

encodering duidelijk te maken is een klein stuk uit afbeelding 18a gehaald en weergegeven in Figuur 18b en de geëncodeerde versie hiervan is weergegeven in afbeelding 18c.

Er zijn verschillende manieren waarop de grote afbeelding in een implementatie kan worden geëncodeerd. Net zoals bij 2D exact pattern matching kan dit worden gedaan door een encoderingsautomaat op te stellen en die vervolgens toe te passen op de grote afbeelding door de automaat te simuleren met de CPU. De encoderingsautomaat is dan een simpele automaat met een lookahead. Het is echter niet nodig in deze situatie om een encoderingsautomaat op te stellen aangezien de grote afbeelding, net zoals in de vorige algoritmes, zal worden geëncodeerd met de CPU; men kan een eenvoudig CPU programma schrijven die dezelfde taak verricht. De enige reden waarom het eventueel interessant kan zijn dit te doen met een automaat is wanneer de AP in de toekomst het toelaat om veel output te genereren zonder verlies van efficiëntie.

Nu de grote afbeelding is geëncodeerd moeten we een automaat opstellen die het patroon van een hole kan matchen en deze automaat configureren op de AP. De automaat zal de string moeten matchen die aantoont dat we aan de onderkant van een hole zitten op het uiterst rechtse symbool. Logisch gezien zal de onderkant van een hole in de geëncodeerde afbeelding een reeks van '1' symbolen zijn waarbij een '0' voorkomt vlak voor de eerste '1' en vlak na de laatste '1', dit is ook te zien in Figuur 18c. In deze reeks mogen enkel '1' symbolen voorkomen, indien er één of meerdere '2' symbolen in de string voorkomen dan weet de automaat dat we niet op de onderste rand zitten. In Figuur 19 is de automaat weergegeven die deze functionaliteit voorziet, deze automaat is gemaakt in de AP Workbench.

Wanneer we deze automaat uitvoeren met de geëncodeerde versie van de afbeelding uit Figuur 18a dan wordt de hole detection uitgevoerd; het resultaat hiervan is weergegeven in Figuur 20. In Figuur 20 is er op iedere positie waarbij de reporting state uit de automaat in figuur 19 actief wordt een rode markering voorzien.

Er is een nadeel bij deze aanpak, in het beste geval is het aantal matches gelijk aan het aantal holes die voorkomen in de afbeelding maar het is ook mogelijk dat het aantal matches groter zal zijn. Dit komt voor wanneer de onderste rand van een hole niet convex is, er zullen dus meerdere reeksen van '1' symbolen zijn voor dezelfde hole. Dit nadeel is een gevolg van het feit dat de kolommen van een hole niet allemaal dezelfde lengte hebben en de kolommen worden van boven naar onder geëncodeerd. Deze methode kan dus beter gezien worden als hulpmiddel waarbij deze false positives en eventueel andere metrieken van de holes zoals de oppervlakte worden verwerkt in een later stadium. Een voorbeeld van dit nadeel is weergegeven in Figuur 21.

4.4.1 Tijdscomplexiteit

Dit algoritme is niet opgesplists in fases zoals bij de vorige algoritmes omdat het niet nodig is om patronen te reduceren naar 1D. De tijdscomplexiteit van het hole detection algoritme is analoog aan de uitvoerende fase van 2D exact pattern

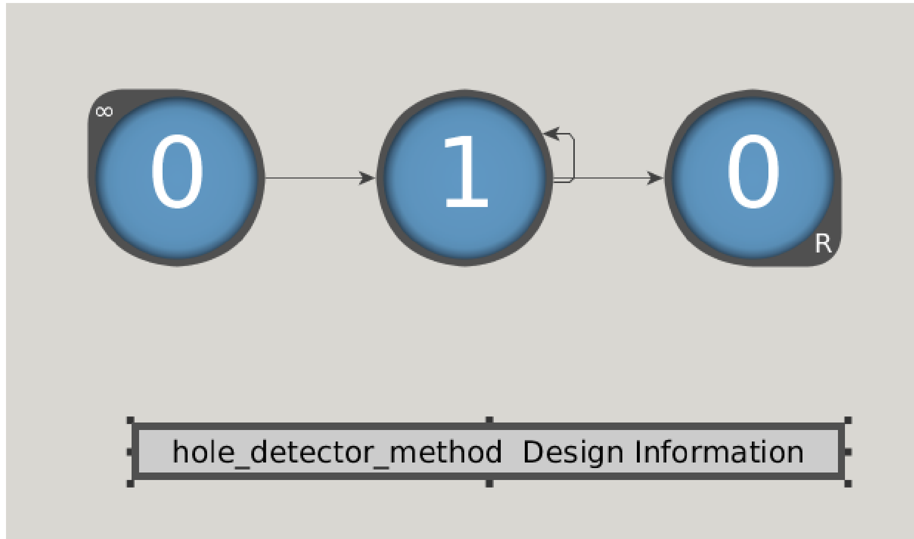


Figure 19: Automaat die hole detection uitvoert op een afbeelding die is geëncodeerd volgens de methode beschreven in sectie 4.4. Deze automaat is gemaakt in de AP Workbench uit de Micron AP SDK.

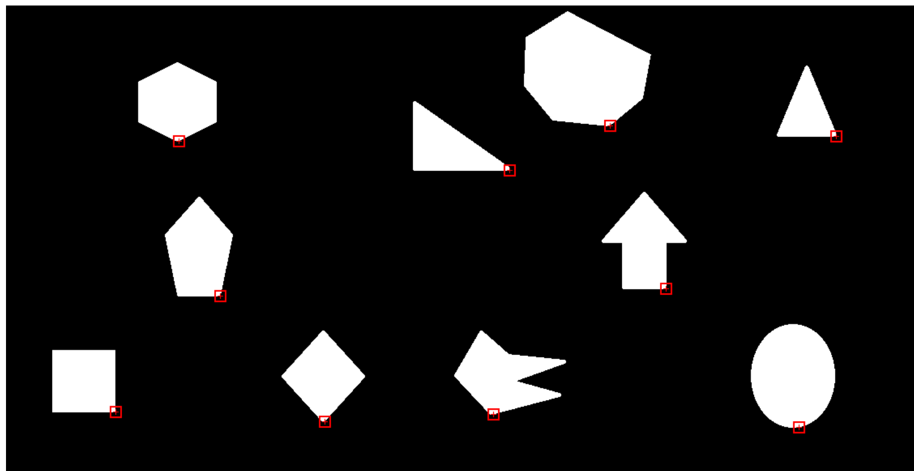
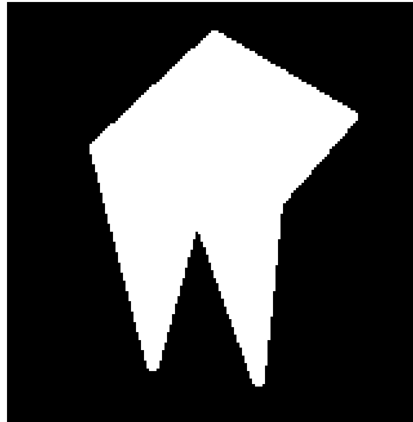
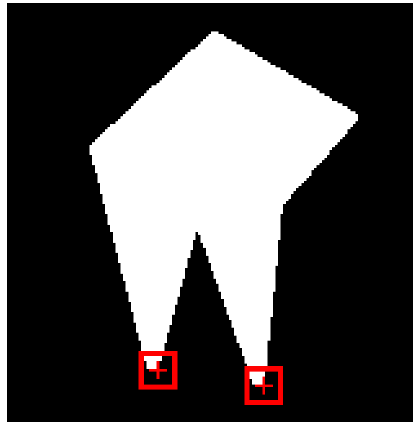


Figure 20: Afbeelding uit Figuur 18a; iedere positie met een rode kader is de positie waarin de automaat in Figuur 19 een match vaststelt. De automaat wordt niet direct uitgevoert op de afbeelding, de afbeelding wordt eerst geëncodeerd volgens de methode beschreven in sectie 4.4.



(a) Afbeelding van een hole met een niet convexe onderrand.



(b) Afbeelding uit Figuur 21a waarop de hole detection uit sectie 4.4 is toegepast. Hierin wordt duidelijk wat het nadeel is van deze methode aangezien er twee matches zijn voor één hole.

Figure 21: Voorbeeldsituatie waarin de hole detection methode uit sectie 4.4 false positives verkrijgt.

matching waarbij de grote afbeelding wordt geëncodeerd en wordt gestreamd naar de AP.

De bottleneck is wederom het encoderen van de grote afbeelding, dit is dan ook $O(mn)$ voor een afbeelding met m rijen en n kolommen. Bij het 2D exact pattern matching algoritme verkrijgt men een betere tijdscomplexiteit wanneer men parallellisatie gebruikt omdat het encoderen van iedere kolom in parallel kan gebeuren. In dit geval is het zelfs mogelijk om ieder symbool van de afbeelding in parallel te encoderen, dit wil dus zeggen dat het encoderen in parallel in constante tijd kan. In de praktijk heeft men hier op generieke CPU's geen baat bij vanwege het beperkt aantal cores. Op GPU architectuur is het wel mogelijk dit uit te buiten vanwege het grote aantal cores.

4.4.2 Conclusie

Het beschreven hole detection algoritme kan efficiënt worden uitgevoerd met de AP. Tot nu toe is bij ieder algoritme de bottleneck voorgekomen bij het encoderen van de grote afbeelding; bij het hole detection algoritme is dit niet niet anders omdat de beschreven methode is afgeleid van de vorige algoritmes. De tijdscomplexiteit voor het encoderen van de grote afbeelding is hetzelfde als bij 2D exact pattern matching. Met parallellisatie kan deze stap veel efficiënter worden gemaakt, dit kan zelfs als "embarrassingly parallel" worden beschreven omdat ieder element van de grote afbeelding apart kan worden geëncodeerd.

Het nadeel is dat dit algoritme niet krachtig genoeg is om ook niet convexe holes op een juiste manier te detecteren. In het geval van het voorbeeld uit Figuur 18 was dit geen probleem maar indien een hole een niet convexe onderrand heeft zoals in Figuur 21 dan treden er false positives op. De reden waarom slechts de onderrand convex moet zijn is vanwege de richting waarin de grote afbeelding wordt geëncodeerd, namelijk van boven naar onder.

5 Praktische toepassing op de AP

In dit deel van het onderzoek zal een algoritme worden toegelicht op basis van een praktische toepassing. Hieruit zal blijken of bepaalde algoritmes praktisch toepasbaar zijn. Voor dit onderzoek is de hardware voor de Micron Automata Processor niet gebruikt. Hier zal vooral worden gelet op betrouwbaarheid van het algoritme, uitvoertijden zullen simpelweg worden afgeleid van het aantal symbolen die zouden worden gestreamd naar de AP.

In iedere stap van de procedure vindt een discussie plaats over de bevindingen, op deze manier worden de methodes in hoofdstuk 4 geënvalueerd en komen eventueel bepaalde problemen aan het licht die niet meteen af te leiden zijn van enkel de theorie.

5.1 Het vinden van letters en cijfers in een afbeelding van een document

De algoritmes uit hoofdstuk 4 hadden steeds als voorbeeld dat een afbeelding van een letter zal worden gezocht in een grotere afbeelding. Dat voorbeeld zal hier worden uitgebreid naar het zoeken van meerdere letters en cijfers binnen een afbeelding van een document die men zou verkrijgen door een document in te scannen.

Het doel van deze implementatie is om te weten te komen of de algoritmes uit hoofdstuk 4 nuttig zijn binnen de context van het vinden van letters en cijfers in een afbeelding.

5.2 Strategie voor de implementatie

Dit is een eerste naïeve strategie die zal worden toegepast voor de implementatie. Deze strategie is puur gebaseerd op de theoretische informatie uit hoofdstuk 4.

Het algoritme van 2D approximate pattern matching is het meest geschikt voor deze implementatie omdat in een realistische situatie de grote afbeelding een document is, verkregen uit bijvoorbeeld een scanner. Hierdoor zal in zekere mate ruis aanwezig zijn in het document en er is mogelijk ook anti aliasing toegepast op de tekst zelf.

De afbeelding uit de scanner zal ofwel een kleur afbeelding zijn of een zwart wit afbeelding, om het algoritme toe te passen zoals het is beschreven in hoofdstuk 4 zal de afbeelding eerst worden geconverteerd naar een binaire afbeelding. De verwachting is dat letters en cijfers nog steeds voldoende herkenbaar zijn zodat het nog steeds mogelijk is voor het algoritme om de juiste letters en cijfers te matchen.

De stappen die worden uitgevoerd op de CPU zullen met een Python script worden uitgevoerd. De stappen op de AP zullen met de *apemulate* binary worden uitgevoerd. Dit betekent dat het encoderen van de grote afbeelding gebeurt door SFFRCO automaten te simuleren op de CPU en het zoeken van matches gebeurt door de SFOLCO automaten te simuleren met de AP. De classificaties van de genoemde automaten zijn terug te vinden in 4.3.1.
Het stappenplan voor het algoritme uit te voeren gaat als volgt:

1. Een afbeelding van een document verkrijgen en converteren naar een binaire afbeelding.
2. Afbeeldingen van patronen verkrijgen.
3. Een SFFRCO automaat genereren voor ieder patroon.
4. De binaire afbeelding van het document voor ieder patroon encoderen met een SFFRCO automaat
5. Een SFOLCO automaat genereren voor ieder patroon.
6. De geëncodeerde afbeelding streamen naar de AP voor ieder patroon.
7. Resultaten interpreteren.

5.3 De afbeelding van het document

De afbeelding is een simpele binaire afbeelding met 3 paragrafen van de "Lorem Ipsum" tekst. De afbeelding wordt gegenereerd zonder anti aliasing en het font en de puntgrootte zijn identiek aan de patronen die worden gebruikt. Door eerst deze simpele afbeelding te gebruiken voor het algoritme is de verwachting dat bepaalde nuances in de implementatie die niets te maken hebben met de input afbeelding worden blootgesteld. De afbeelding is weergegeven in Figuur 22

Iedere afbeelding van een document wordt geconverteerd naar een string door een 0 te vervangen met het karakter '0' en waardes groter dan 0 worden vervangen door het karakter '1'. Naar deze string zullen we verder verwijzen als de documentstring. Voor deze implementatie bepalen we dat de achtergrond 0 is en de tekst zelf 1 is.

5.4 De afbeeldingen die als patronen dienen van alle letters en cijfers

De patronen die zullen worden gezocht in de grote afbeelding zullen eerst beschikbaar moeten worden gemaakt. De patronen zijn afbeeldingen van letters en cijfers. De afbeeldingen zijn allemaal binaire afbeeldingen en zullen worden gegenereerd met een script, ieder patroon wordt dus synthetisch gemaakt. Iedere letter en cijfer zal zonder anti aliasing worden gegenereerd omdat voor de implementatie met binaire afbeeldingen wordt gewerkt. Voor deze implementatie zal worden gewerkt men het "FreeMono" font en een puntgrootte van 16.

De letters en cijfers die zullen worden gezocht zijn: 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'. Iedere letter en cijfer wordt op een afbeelding gerendered en er wordt een bounding box berekend zodat het patroon geen buitenrand heeft. Dit zorgt dus voor 62 patronen van verschillende afmetingen. In Tabel 2 zijn de afmetingen voor ieder patroon terug te vinden.

5.5 Het genereren van een SFFRCO automaat voor ieder patroon

Voor ieder patroon moet een SFFRCO automaat worden gegenereerd, iedere SFFRCO automaat dient als encoderingsautomaat waarmee de input afbeelding zal worden geëncodeerd. Dit wordt gedaan zoals beschreven in 4.3.2. Na het uitvoeren van deze stap is er een SFFRCO automaat beschikbaar gemaakt voor ieder patroon. In tabel 2 kan men voor ieder patroon het aantal states in de SFFRCO automaat terugvinden. Merk op dat automaten met dezelfde afmetingen evenveel states hebben in hun SFFRCO automaat.

Patroon	Afmetingen (pixels)	Aantal states
0	6x10	390
1	5x10	325
2	6x10	390
3	7x10	455

4	6x10	390
5	7x10	455
6	6x10	390
7	6x10	390
8	6x10	390
9	7x10	455
A	9x9	486
B	8x9	432
C	8x9	432
D	8x9	432
E	8x9	432
F	8x9	432
G	9x9	486
H	8x9	432
I	5x9	270
J	8x9	432
K	9x9	486
L	7x9	378
M	10x9	540
N	8x9	432
O	8x9	432
P	7x9	378
Q	8x11	616
R	8x9	432
S	7x9	378
T	8x9	432
U	8x9	432
V	9x9	486
W	10x9	540
X	8x9	432
Y	9x9	486
Z	6x9	324
a	8x7	280
b	7x10	455
c	8x7	280
d	8x10	520
e	7x7	245
f	5x10	325
g	7x10	455
h	8x10	520
i	5x9	270
j	4x12	360
k	7x10	455

l	5x10	325
m	8x7	280
n	8x7	280
o	8x7	280
p	8x10	520
q	8x10	520
r	6x7	210
s	6x7	210
t	7x9	378
u	6x7	210
v	9x7	315
w	9x7	315
x	8x7	280
y	8x10	520
z	6x7	210

Table 2: Gegevens over de gegenereerde SFFRCO automaten voor de praktische toepassing van 2D approximate pattern matching. Deze gegevens blijven onveranderd voor iedere uitvoer van het algoritme omdat steeds dezelfde patronen zullen worden gebruikt.

5.6 Het encoderen van documentstring voor ieder patroon met een SFFRCO automaat

In deze stap zal iedere SFFRCO automaat uit de vorige stap worden gesimuleerd met de documentstring als input. Het resultaat van deze stap is een geëncodeerde versie van de documentstring voor ieder patroon. Deze stap is de bottleneck van de hele procedure omdat hier meerdere niet deterministiche automaten moeten worden gesimuleerd op de CPU. De code uit Listing 1 zal worden toegepast op iedere SFORCO automaat binnen de SFFRCO automaat.

Door deze stap uit te voeren wordt ieder karakter van de documentstring vervangen door een lijst van error waardes. De lengte van iedere lijst is gelijk aan de breedte van het patroon, het aantal symbolen in de geëncodeerde documentstring is dus gelijk aan het aantal symbolen in de documentstring maal de breedte van het patroon.

Omdat er bepaalde patronen zijn met een hoogte van 10 of meer kan het voorkomen dat in de SFOLCO automaat een overgang moet zijn voor 10 fouten. Omdat het cijfer '10' bestaat uit meerdere karakters is het in dit geval beter om de Anml op te bouwen met numerieke STE's, het is dan mogelijk om een STE te maken met een overgang voor de waarde '10'. Numerieke STE's hebben echter een limiet van 255, deze oplossing is dus niet geschikt voor overgangen van 256 fouten of meer maar dit komt niet voor in deze implementatie.

Naar de volgende stap toe, het genereren van een SFOLCO automaat, moet beslist worden op welke manier de geëncodeerde documentstring moet worden

Lorem ipsum dolor sit amet, consectetur adipiscing
 elit. Pellentesque interdum risus vitae turpis ma
ximus maximus non bibendum lectus. Nunc sit amet t
urpis et dui mollis scelerisque. Praesent pretium
luctus facilisis. Phasellus interdum mi a sem phar
etra, quis vulputate sapien rhoncus. Nunc ut ex te
llus. Fusce lacinia euismod orci, ac faucibus ipsu
m tincidunt at. Aliquam in efficitur libero. Sed e
get orci leo. Ut dignissim ullamcorper ipsum et la
oreet. Morbi vel neque arcu. Curabitur bibendum ju
sto a molestie molestie. Vestibulum sit amet ante
ut augue interdum accumsan in luctus dolor.

Curab
itur condimentum diam nisl, in porta tortor blandi
t in. Curabitur et orci neque. Fusce mattis sit am
et quam ac blandit. Nulla ut lobortis mi, vitae so
dales dolor. Fusce mollis aliquam metus tincidunt
viverra. Nulla eget finibus lorem. Sed at velit fa
cilisis, aliquet lectus et, vehicula urna. Interdu
m et malesuada fames ac ante ipsum primis in fauci
bus.

Quisque ipsum mauris, pulvinar et vehicula q
uis, vulputate vel orci. Integer gravida, ligula n
ec dapibus ultrices, dui ligula aliquet dolor, ac
bibendum nisi nibh vel felis. Sed vitae fermentum
odio. Quisque at nisi quis neque cursus accumsan.
Duis enim mauris, volutpat vel pulvinar et, placer
at ut lorem. Curabitur eget suscipit sem, at tinci
dunt ante. Suspendisse ut laoreet ligula. Nullam m
olestie, neque eu sodales ornare, erat tortor curs
us lectus, lacinia dictum dolor est ac turpis. Eti
am in quam ut lorem euismod ullamcorper et a neque
. In non est ut magna consectetur venenatis sit am
et et mauris. Vivamus vehicula luctus tristique. S
ed finibus nisi suscipit quam semper dapibus. Duis
 vitae lectus ultricies nibh facilisis condimentum
 ut faucibus lorem. Ut a risus suscipit, imperdiet
 ex at, scelerisque ex. Duis hendrerit neque ultri
ces, suscipit erat a, tristique mi.

Figure 22: De eerste afbeelding waarop de praktische implementatie van 2D approximate pattern matching zal worden toegepast. De afbeeling is binair, de tekst heeft geen anti aliasing. De afmetingen van de afbeelding zijn 509x703.

geordend; dit hangt af van welke techniek zal worden gebruikt uit probleem 2 van 4.3.7. Ofwel blijft de volgorde onveranderd en zal er gebruik worden gemaakt van 'dummy states', ofwel moet de volgorde van de symbolen van de documentstring worden veranderd.

Het aantal STE's in een SFOLCO automaat groeit sterk naargelang de hoogte van een patroon en de error threshold, wanneer 'dummy states' worden toegevoegd groeit dit aantal states ruwweg met een factor van de breedte van het patroon. Er wordt dus gekozen om 'dummy states' te vermijden en de volgorde van de geëncodeerde documentstring aan te passen.

5.7 Een SFOLCO automaat genereren voor ieder patroon

Een SFOLCO automaat voor een patroon wordt opgesteld op basis van de afmetingen van het patroon en de gekozen error threshold. De afmetingen van een patroon zijn reeds gekend, de error threshold moet nog op een bepaalde manier worden beslist. Voor deze implementatie wordt voor ieder patroon een aparte error threshold berekend op basis van een vooraf gekozen percentage. De betekenis van het percentage is het maximaal aandeel van karakters dat niet overeenkomt. De reden voor de gekozen werkwijze is om geen bias te hebben voor bepaalde patronen.

Een bijkomende opmerking bij deze stap is dat deze ook vroeger kan voorkomen in de procedure. De enige gegevens die nodig zijn voor deze stap zijn de afmetingen van ieder patroon en een error threshold, de inhoud van de patronen is niet belangrijk.

In de vorige stap is beslist om geen 'dummy states' te gebruiken. De SFOLCO automaten worden eerst opgebouwd als klassieke NFA, vervolgens worden deze geconverteerd naar hun Anml equivalent. Dit gebeurt door de overgangen in de klassieke NFA te converteren naar STE's. In Figuur 23 zijn de statistieken van deze implementatie over de gegenereerde SFOLCO automaten terug te vinden. Het meest opmerkelijk aspect in deze gegevens is de sterke groei van STE's bij het stijgen van de error threshold. Deze groei is zeer sterk bij patronen met het grootste aantal pixels. Dit is een gevolg van het gebruiken van een percentage van het totaal aantal pixels als error threshold, de bekomen error threshold k groeit dan sterker als bij patronen met een kleiner totaal aantal pixels.

Dit gevolg is goed te zien bij de patronen met afmetingen '5x10', '6x10' en '7x10'. De grootte van de automaat is afhankelijk van de hoogte van het patroon en k, dus als we voor al deze patronen dezelfde k zouden gebruiken dan zijn alle resulterende automaten even groot. Door het totaal aantal pixels te gebruiken om k te berekenen wordt ook de breedte van het patroon in rekening gebracht en groeien de automaten van bredere patronen sneller dan andere patronen met dezelfde hoogte.

In de Figuur is ook een gegeven 'Max STE' aanwezig die constant is voor de waarde 24.576; dit is de helft van het maximaal aantal STE's die aanwezig kunnen zijn op één enkele chip. Het 'Max STE' gegeven is hier een referentie voor het maximaal aantal STE's dat mag voorkomen in een SFOLCO automaat.

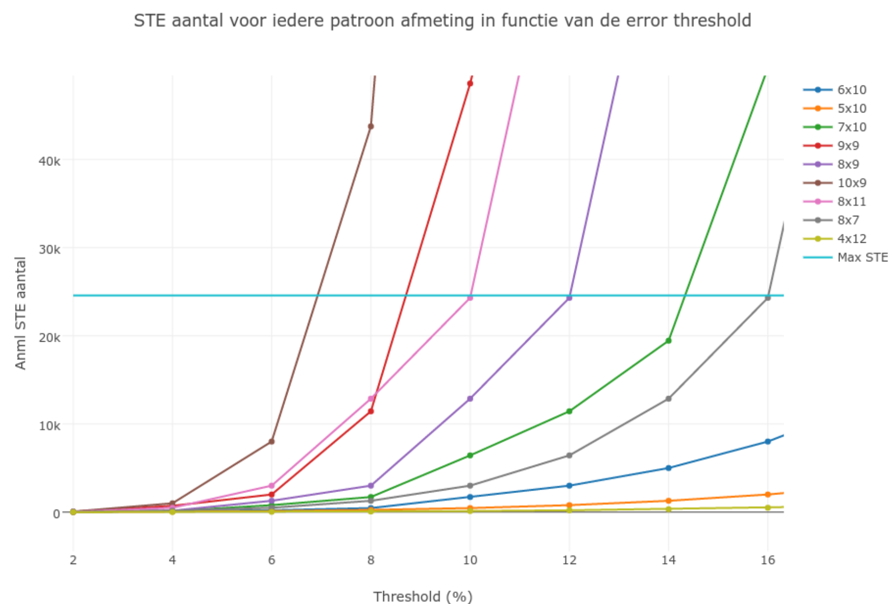


Figure 23: De grootte van iedere SFOLCO automaat uitgedrukt in het aantal STE's. De grootte van een SFOLCO automaat is afhankelijk van de afmetingen van het patroon en de gekozen error threshold. De 'Max STE' data is het maximaal aantal STE's dat een SFOLCO automaat kan hebben om op 1 AP chip te passen en dient als referentie voor de andere data. Op basis van deze data is het duidelijk dat iedere automaat exponentieel groeit.

De redenering achter dit maximum is dat er geen routering mogelijk is tussen twee half cores van een chip terwijl alle STE's van een SFOLCO automaat steeds verbonden zijn via een bepaald pad, de automaat moet dus kunnen passen op één enkele half core.

De SFOLCO automaat voor de afmeting '10x9' overschrijdt het maximaal aantal STE's al bij een error threshold van ongeveer 7%. Indien nog steeds geen enkele bias voor bepaalde patronen mag voorkomen moet deze maximale error threshold ook voor de andere patronen gelden. Dit is zeer limiterend omdat andere patronen nog veel ruimte hebben bij deze threshold. Ook indien grotere patronen moeten worden gebruikt zal snel dit limiet worden bereikt.

Om deze automaten de configureren op de AP is het ook nodig deze eerst te compileren, de tijd die het kost om zeer grote automaten zoals hier voorkomen te compileren is dan ook groot. Een goede strategie is om pre processing te doen en zodat iedere benodigde SFOLCO automaat op voorhand is gecompileerd, dit is enigszins mogelijk omdat enkel de afmetingen van de patronen gekend moeten zijn en niet de inhoud.

Op basis van deze gegevens wordt vastgesteld dat de manier van het opbouwen van de SFOLCO automaat zeer slecht schaalt. Een oplossing hiervoor ligt eventueel bij het gebruik van counter of boolean elementen uit Anml; deze dienen immers om automaten compacter te maken.

5.8 De geëncodeerde documentstring streamen naar de AP voor ieder patroon

Voor ieder patroon is er nu een geëncodeerde documentstring en een SFOLCO automaat beschikbaar. Om nu het patroon terug te vinden in het oorspronkelijk document moet de AP worden geconfigureerd met de juiste SFOLCO automaat van het patroon en vervolgens moet de respectievelijke geëncodeerde documentstring worden gestreamd naar de AP. Indien er een match is gevonden dan is het huidige patroon gevonden in het document. Bij een match wordt vermeldt op welke positie van de input de match voorkomt en wat de reportcode is, aan de hand van de positie is het mogelijk te achterhalen waar het patroon zich bevindt in de afbeelding van het document, de report code is het aantal fouten die voorkomen bij de match ten opzichte van het patroon.

In Tabel 3 zijn de resultaten van de berekening van de uitvoertijden weergegeven. De uitvoertijden zijn sterk afhankelijk van het aantal symbolen dat wordt gestreamd naar de AP en het aantal gebruikte chips van de AP; men kan namelijk dezelfde automaat configureren op meerdere chips om de stream capaciteit te verhogen. In de Tabel is de stream capaciteit 1 Gbps. Merk op dat de uitvoertijden van dezelfde grootte orde zijn, dit staat in contrast met de resultaten verkregen bij het genereren van de SFOLCO automaten in Figuur 23 waar bepaalde patronen veel meer STE's bevatten en dus veel groter zijn.

Dit toont een interessante eigenschap van de AP, namelijk dat wanneer de automaat groter wordt dat de uitvoertijd niet noodzakelijk ook groter wordt. De uitvoertijd wordt vooral beïnvloed door het aantal chips waarop de automaat kan worden gerepliceerd en door de mate waarin reporting states actief worden.

De mate waarin reporting states actief worden is belangrijk omdat de buffer met outputvectoren moet worden leeggemaakt indien deze volledig wordt opgevuld.

Het aantal symbolen dat moet worden gestreamd is niet hetzelfde voor ieder patroon terwijl steeds een geëncodeerde documentstring van hetzelfde document wordt gestreamd. Dit is een gevolg van het encoderen van de documentstring waarbij ieder element van de documentstring wordt vervangen door een tabel van error waardes, het aantal error waardes is gelijk aan de breedte van het patroon.

De totale uitvoertijd in de laatste rij van de tabel is de som van de uitvoertijden van alle patronen. Er is bij dit totaal geen rekening gehouden met de overhead die men krijgt wanneer men van SFOLCO automaat wisselt door de AP te herconfigureren. Het is verwacht dat deze overhead klein blijft omdat een binaire image configureren op de AP efficiënt is.

Patroon	Aantal symbolen	Uitvoertijd (seconden)
0	2146947	0.0019995002076
1	1789125	0.00166625250131
2	2146947	0.0019995002076
3	2504768	0.00233274698257
4	2146947	0.0019995002076
5	2504768	0.00233274698257
6	2146947	0.0019995002076
7	2146947	0.0019995002076
8	2146947	0.0019995002076
9	2504768	0.00233274698257
A	3220407	0.00299923773855
B	2862588	0.00266599282622
C	2862588	0.00266599282622
D	2862588	0.00266599282622
E	2862588	0.00266599282622
F	2862588	0.00266599282622
G	3220407	0.00299923773855
H	2862588	0.00266599282622
I	1789125	0.00166625250131
J	2862588	0.00266599282622
K	3220407	0.00299923773855
L	2504768	0.00233274698257
M	3578225	0.00333248171955
N	2862588	0.00266599282622
O	2862588	0.00266599282622
P	2504768	0.00233274698257
Q	2864912	0.00266815721989
R	2862588	0.00266599282622
S	2504768	0.00233274698257
T	2862588	0.00266599282622

U	2862588	0.00266599282622
V	3220407	0.00299923773855
W	3578225	0.00333248171955
X	2862588	0.00266599282622
Y	3220407	0.00299923773855
Z	2146947	0.0019995002076
a	2862588	0.00266599282622
b	2504768	0.00233274698257
c	2862588	0.00266599282622
d	2862588	0.00266599282622
e	2504768	0.00233274698257
f	1789125	0.00166625250131
g	2504768	0.00233274698257
h	2862588	0.00266599282622
i	1789125	0.00166625250131
j	1463110	0.00136262737215
k	2504768	0.00233274698257
l	1789125	0.00166625250131
m	2862588	0.00266599282622
n	2862588	0.00266599282622
o	2862588	0.00266599282622
p	2862588	0.00266599282622
q	2862588	0.00266599282622
r	2146947	0.0019995002076
s	2146947	0.0019995002076
t	2504768	0.00233274698257
u	2146947	0.0019995002076
v	3220407	0.00299923773855
w	3220407	0.00299923773855
x	2862588	0.00266599282622
y	2862588	0.00266599282622
z	2146947	0.0019995002076
TOTAAL	162843923	0.151660221629

Table 3: Tabel die de uitvoertijden bevat voor het streamen van de geëncodeerde afbeelding van ieder patroon. Het streamen gebeurt aan 1 Gbps. De totale uitvoertijd is de totale som van de uitvoertijden voor ieder patroon exclusief de overhead van het herconfigureren van de AP voor ieder patroon. In deze situatie wordt verondersteld dat de buffer met outputvectoren niet volledig opgevuld wordt.

5.9 Conclusie

Het algoritme van 2D approximate pattern matching zoals beschreven in sectie 4.3 is nu toegepast in de situatie van het vinden van letters en cijfers binnen een afbeelding van een document. Het doel van de toepassing was om het algoritme te evalueren en eventuele problemen te ontdekken.

Het grootste probleem dat is ontdekt is de exponentiële groei van de SFOLCO automaten in functie van een gekozen error threshold. Dit zorgt voor een gelimiteerde keuze voor een error threshold. Een voorstel voor dit probleem op te lossen is door de opbouw van de SFOLCO automaat niet langer te doen volgens de klassieke manier maar eerder gebruikmakend van de Anml componenten die bedoelt zijn om automaten compacter te maken. Dit is echter geen probleem indien men een error threshold kan kiezen die onder dit limiet valt. In de toepassing was het nog steeds mogelijk om een error threshold van 7% te kiezen.

Aan de uitvoertijden werd een belangrijke eigenschap van de AP duidelijk. De uitvoertijden voor ieder patroon zijn van dezelfde grootte orde terwijl de SFOLCO automaten van sommige patronen veel groter zijn dan die van andere patronen.

Het is een goede strategie om de SFOLCO automaten te genereren en te compileren in een pre processing fase. Indien men dit doet tijdens het uitvoeren van het algoritme dan kan men geen gebruik maken van de efficiëntie die men verkrijgt bij het gebruik van de AP.

6 Algemene conclusie en toekomstig werk

De Micron Automata Processor is gebouwd als processor die zeer efficiënt pattern matching kan verrichten. De pattern matching waar de AP zich in onderscheidt is vooral 1D pattern matching waarbij men naar bepaalde strings op zoek gaat in een gigantisch domein van input. Dit is duidelijk aan de huidige populaire toepassingen van de AP zoals signature analysis bij Network Security of motif search bij Bioinformatica. Het is daarom ook een uitdaging geweest binnen dit onderzoek om toepassingen te vinden binnen beeldverwerking, een vakgebied waarbij men werkt met twee dimensionale data.

Op basis van de toegelichte algoritmes in dit onderzoek kan worden vastgesteld dat het mogelijk is om beeldverwerking te doen met de AP ten koste van een stap waarbij de twee dimensionale data wordt gereduceerd naar een 1D string. Deze stap waarin de dimesionale reductie wordt toegepast bepaalt telkens de tijdscomplexiteit van het gevonden algoritme omdat dit de bottleneck vormt voor het algoritme.

Deze extra stap die bij ieder algoritme wordt toegepast kan gezien worden als een workaround om beeldverwerking mogelijk te maken op de AP in zijn huidige staat. De vereiste waarmee het mogelijk zal worden om de algoritmes uit te voeren zonder deze stap is wanneer de AP ondersteuning biedt voor twee dimensionale data te hebben als input of als er een bepaald mechanisme wordt

geïmplementeerd waarmee men meer controle krijgt over welk symbool door welke automaat wordt verwerkt. Dit tweede kan ervoor zorgen dat het mogelijk wordt om twee dimensionale data te streamen als een 1D string en te verwerken als twee dimensionale data op de AP zelf.

Nog een mogelijkheid waarbij deze bottleneck kan worden opgelost is wanneer deze extra stap kan worden uitgevoerd op de AP zelf. De extra stap is steeds geformuleerd als de simulatie van een automaat maar deze is niet geschikt om te simuleren met de AP omdat de automaat veel output moet kunnen genereren. Indien de AP betere ondersteuning krijgt voor het genereren van output wordt het mogelijk de dimensionale reductie toe te passen met de AP en wint men veel efficiëntie bij ieder algoritme in deze thesis.

6.1 Synopsis

In deze thesis is er op zoek gegaan naar toepassingen binnen het vakgebied beeldverwerking met de Micron Automata Processor. Het onderzoek is gestart bij de kennismaking met de tools die beschikbaar zijn voor onderzoekers om applicaties te ontwikkelen voor de Micron Automata Processor. Een onderdeel van deze kennismaking was het uitproberen van simpele concepten in beeldverwerking zoals hole detection waarvoor pas later in het onderzoek een werkwijze is gevonden. De reden waarom niet direct een manier van werken was gevonden is omdat de Micron Automata Processor, net zoals klassieke automaten, enkel werkt met één dimensionale data. Hier ligt het meest fundamentele limiet dat moet worden overwonnen om Beeldverwerking haalbaar te maken op de Micron Automata Processor.

Er zijn drie algoritmes aan bod gekomen die een mogelijke toepassing kunnen hebben in Beeldverwerking, namelijk 2D Exact pattern matching, 2D approximate pattern matching en 2D hole detection. In ieder van deze algoritmes is het vereist om dimensionale reductie toe te passen op de input afbeelding. De input afbeelding wordt op een bepaalde manier geëncodeerd zodat de afbeelding wordt gereduceerd naar een één dimensionale string. Eens dat de afbeelding is voorgesteld als een 1D string dan wordt het mogelijk om op een efficiënte manier informatie te halen uit de afbeelding met de Micron Automata Processor. Dit was steeds in grote lijnen de werkwijze bij ieder algoritme.

In het laatste gedeelte van het onderzoek is de theorie voor 2D approximate pattern matching uitgetest in een praktische toepassing. Het algoritme werd getest door letters en cijfers te zoeken in een testafbeelding. Iedere letter en cijfer is een patroon dat wordt gezocht in de testafbeelding met een bepaalde error threshold.

6.2 Toekomstig werk

In dit onderdeel zijn een aantal onderwerpen die verder gaan op het het onderzoek in deze thesis. Ieder onderwerp heeft als doel om de algoritmes die aan bod komen in deze thesis ofwel efficiënter te maken ofwel uit te breiden.

6.2.1 Micron Automata Processor met 2D ondersteuning

Onderzoek naar de mogelijkheid om de Micron Automata Processor ondersteuning te geven voor twee dimensionale data. Dit zou eventueel de mogelijkheid bieden om de algoritmes beschreven in deze thesis uit te voeren zonder de bottleneck van het encoderen van de input data.

6.2.2 Rotatie en schaal invariante patronen

Bij de algoritmes waar specifieke patronen worden gezocht in een grotere afbeelding neemt het algoritme aan dat de patronen onder dezelfde hoek staan als in de grote afbeelding en dezelfde schaal hebben. Dit kan enigzins opgelost worden met een error threshold maar een hogere error threshold zorgt voor meer false positives.

6.2.3 Micron Automata Processor met ondersteuning voor veel output

De dimensionale reductie wordt bij 2D Approximate pattern matching toegepast op een afbeelding door een niet deterministische automaat te simuleren. Deze automaat wordt bewust niet uitgevoerd op de Micron Automata Processor omdat deze na iedere cyclus veel output genereert. Dit zorgt ervoor dat het simuleren van deze automaat inefficiënt is op de Micron Automata Processor. Indien het mogelijk is om veel output te genereren zonder efficiëntie te verliezen dan wordt het mogelijk dimensionale reductie toe te passen op een efficiënte manier.

6.2.4 Het zoeken naar woorden in een afbeelding van een document

De praktische toepassing in deze thesis was het zoeken naar letters en cijfers binnen een afbeelding van een document. Een uitbreiding hierop is het zoeken naar woorden waarbij men nog steeds gebruik maakt van de AP voor op basis van de gevonden letters en cijfers woorden kan matchen.

References

[1] Alfred Vaino Aho and Margaret John Corasick: Efficient string matching: An aid to bibliographic search. Commun. ACM, 18(6):333340, 1975.

[2] Indranil Roy: ALGORITHMIC TECHNIQUES FOR THE MICRON AUTOMATA PROCESSOR, 2015

[3] Jan Žd'árek Two-dimensional Pattern Matching Using Automata Approach, 2010

[4] Bořivoj Melichar and Jan Holub: 6D classification of pattern matching problems. In Jan Holub, editor, Proceedings of the Prague Stringology Club Workshop 97, pp. 2432, Czech Technical University in Prague, Czech Republic, 1997. Collaborative Report DC9703.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman: Introduction to Au- tomata Theory, Languages, and Computation. Addison-Wesley, Reading, MA, 2001. Second edition, 535 pages.

[6] Jan Holub: Simulation of nondeterministic finite automata in pattern matching. Dissertation thesis, Czech Technical University in Prague, Czech Republic, 2000.

[7] Peter H. Sellers: The theory and computation of evolutionary distances: Pattern recognition. J. Algorithms, 1(4):359373, 1980.

[8] Sun Wu and Udi Manber: Fast text searching allowing errors. Commun. ACM, 35(10):8391, 1992.

[9] Maxime Crochemore and Christophe Hancart: Automata for matching patterns. In Grzegorz Rozenberg and Arto Salomaa, editors, Handbook of Formal Languages, pp. 399462. Springer-Verlag, Berlin, 1997.

[10] Richard S. Bird: Two-dimensional pattern matching. Inf. Process. Lett., 6(5):168 170, October 1977.

[11] Theodore P. Baker: A technique for extending rapid exact-match string matching to arrays of more than one dimension. SIAM J. Comput., 7(4):533541, November 1978.

[12] Esko Ukkonen: Finding approximate patterns in strings. J. Algorithms, 6(13):132 137, 1985.

[13] Bořivoj Melichar: Approximate string matching by finite automata. In Václav Hlav and Radim Šára, editors, Computer Analysis of Images and Patterns, No. 970 in Lecture Notes in Computer Science, pp. 342349. Springer-Verlag, Berlin, 1995.

[14] "The Micron Automata Processor" http://www.micronautomata.com/. Accessed: May, 2017.

[15] D LUGOSCH , P., B ROWN , D., G LENDENNING , P., L EVENTHAL , M., and N OYES , H., "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing", IEEE Transactions on Parallel and Distributed Systems, vol. 99, p. 1, 2014.

[16] "Anml Documentation" https://www.micronautomata.com /documentation/anml_documentation/. Accessed: May, 2017.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Beeldverwerking met de Micron Automatic Processor

Richting: **master in de informatica-multimedia**
Jaar: **2017**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Goyens, Frank

Datum: **9/06/2017**