Persoonlijk verslag
TINLAB Machine Learning

Muhamed Agic
0946536

May 25, 2020

Contents

1 Inleiding

Dit document is een persoonlijk verslag van de stof die is behandeld in de TINLAB Machine Learning. In dit verslag zijn onder andere korte samenvattingen van de papers die beschikbaar zijn gesteld in de TINLAB, over de behandelde stof in de lessen en wat onderwerpen die gaan over de implementatie van het eindproject.

2 Introductie AI

Bij Artificial Intelligence wordt er vaak gedacht over systemen die zelf snelle en goede intelligente beslissingen kunnen maken. Globaal gezien heeft AI twee doelen:

- Scientific goal: Er wordt onderzocht welke ideeën over kennis, leren, het opstellen en/of volgen van regels, zoektechnieken en meer verschillende soorten intelligentie kunnen verklaren en uitleggen.
- Engineering goal: Wordt gebruikt om real world problems op te lossen met AI technieken zoals het toepassen van slimme zoektechnieken, knowlegde representation en meer.

AI kan op de volgende manier beschreven worden. Men kan deze benaderingen hebben:

- Systemen die als mensen **denken**
- Systemen die rationeel **denken**
- Systemen die als mensen **handelen**
- Systemen die rationeel **handelen**

AI heeft als het ware een menselijke kant en een rationele kant. Een benadering is er om AI te laten denken en handelen als mensen. Het nadeel daarvan is dat mensen gevoelens en andere factoren hebben en zich daar mogelijk door kunnen laten leiden, waardoor sommige beslissingen niet altijd correct worden gemaakt. Een andere benadering is dus om rationeel te denken en te handelen, zodat er geen menselijke fouten ontstaan.

2.1 Acting humanly: The Turing test

Een grote vraag die Turing had was of machines kunnen denken en of mensen machines kunnen laten denken als mensen. Er is een test gedaan waarbij een persoon aan de ene kant van een niet doorzichtig scherm zat en een persoon en een machine aan de andere kant. Als de proefpersoon niet door heeft of niet kan achterhalen of het met een mens of machine spreekt, dan heeft de macine de Turing test doorstaan.

Het probleem met de turing test is echter dat de Turing test niet reproduceerbaare, constructief of onverandelijk is voor wiskundige analyse.

2.2 Thinking humanly: Cognitive Science

2.3 Thinking rationally: Laws of Thought

Rationeel denken is een manier van denken die als goed, passend en correct worden gezien. Aristoteles (ongeveer 350 voor christus) vroeg zich af wat correct

denken en redeneren is en hoe men dat proces correct kan laten verlopen. Hoe kan je altijd winnen in een discussie door logica te gebruiken? Zulke vragen werden opgeworpen en behandeld in die tijd.

Er zitten ook gevaren en ethische problemen aan rationeel denken. Wie red je bijvoorbeeld als eerst als er twee mensen bewusteloos op de intensive care liggen terwijl en hulp is voor één persoon? Verder **kan** het zo zijn dat de reflex van de mens soms handiger kan zijn dan een rationeel denkende entiteit. Als iemand zich verbrand aan iets is het beter om zo snel mogelijk actie te ondernemen in plaats van rationeel na te denken en deze stappen te nemen:

- Mijn huid zit op de oven
- De oven is 200 graden Celcius
- Mijn lichaamstemperatuur is ongeveer 37 graden
- Deze situatie kan brandwonden veroorzaken
- Haal je arm van de oven af

Nadelen van rationeel denken zijn dat niet al het intelligente gedrag ontstaat door rationeel na te denken. Verder is er bij rationeel nadenken ook de vraag welke gedachten men **zou moeten hebben** van alle gedachten die men **zou kunnen hebben**.

2.4 Acting rationally

Kort gezegd is rationeel handelen het volgende: Doe hetgeen wat juist/correct is. Wat in deze context juist en correct betekent is hetgeen wat gedaan moet worden om zo goed mogelijk een bepaald doel te halen met de informatie die beschikbaar is. Bij het rationeel handelen hoeven denken en reflexen niet per see aan bod te komen, neem als voorbeeld goede manieren die je vanuit jezelf toepast zonder na te denken.

2.5 Rational agents

Rational agents zijn entiteiten die waarnemen en handelen. Rationele agents kan men ontwerpen. Een rationele agent kan globaal gezien worden als een functie die aan de hand van waarnemingen acties uitvoert, uit te drukken in de volgende formule: $f : P^* \rightarrow A$.

Voor elke klasse van omgevingen en taken zoekt men de beste performance van een agent of een klasse aan agents. Hier zit wel een disclaimer aan vast, een perfecte agent bestaat niet, omdat dat oneindig veel computation time nodig heeft. Perfecte rationaliteit is door computational limits niet te bereiken.

2.6 Oorsprong en fundamenten van AI

AI is een middel dat voor veel dingen te gebruiken is. Een lijstje met vakgebieden waar AI handig voor te gebruiken is:

- Filosofie
- Wiskunde en computation
- Economie
- Biologie en neurowetenschappen
- Computer engineering

filosofie Vragen die opgeworpen worden bij de wiskundige kan van AI zijn bijvoorbeeld:

- Wat zijn de formele regels om een geldige conclusie te kunnen trekken?
- Wat kan men computen/uitrekenen?
- Hoe redeneert men als bepaalde informatie ontbreekt?

What are the formal rules to draw valid conclusion?What can be computed?How do we reason with uncertain information?

Economie Biologie en neurowetenschappen Computer engineering

3 Problem Sovling Agents

Een problem solving agent is een subtype van een goal agent. Om een goal te realiseren is er immers iets nodig dat bepaalde problemen oplost zodat dat doel behaald kan worden. Om sommige typen problemen en problem solving agents uit te leggen wordt er gebruik gemaakt van het voorbeeld van een map van een land, in dit geval Roemenië, die te zien is in figuur 11.

3.1 Probleem Types

Om bepaalde problemen te kunnen oplossen is het belangrijk om problemen eerst goed te definiëren zodat men weet hoe het probleem aangepakt kan worden. Er zijn verschillende typen problemen die kunnen voorkomen. Enkele daarvan zijn:

- **Single-state probleem**: Dit is **Deterministisch en/of fully observable**. De agent weet precies in welke state het zal zitten. De agent weet so to speak de hele map uit figuur 11 uit zijn hoofd.
- **Conformant probleem**: Dit is een **Non observable** probleem. Het kan zo zijn dat de agent geen idee heeft waar het kan zitten. De agent zal hier meerdere keren moeten proberen een bepaald probleem op te lossen om, in het geval van het Roemenië voorbeeld, de map te verkennen.
- **Contingentie probleem**: Dit is een **Non deterministisch en/of partitially observable** probleem. Bij dit soort problemen kan je maar één stap vooruit denken. Als men van Arad naar Bucharest zou willen (zie figuur 11) dan zou de agent alleen het gebied Arad, Zerind, Sibiu en Timisoara kennen. Als de agent vervolgens in Sibiu zit weet de agent alleen dat het naar Rimnicu Vilcea en Fagaras kan.
- **Exploration probleem**: Dit is een **Unknown state space** probleem. Bij dit probleem weet de agent helemaal niks en moet de agent 'exploren' en door middel van trial en error de oplossing zoeken. Men kan hieruit concluderen dat dit niet tot de meest efficiente oplossing kan leiden ...

Deze problemen kunnen op een nettere manier geformuleerd worden. Het **single-state probleem** is gedefinieerd aan de hand van de volgende 4 elementen:

- De initial state: dat kan een stad zijn in het geval van figuur 11
- De successor function $S(x)$: Dit is een set met action-state paren. Voorbeeld: $S(Arad) = \ (Arad \rightarrow Zerind), \ldots$
- Goal test: Check of het doel is gehaald. Expliciet: $x =$ at Bucharest of impliciet: atBucharest(x)
- Path cost: Dit kan de som van de afstanden zijn als men naar een kortste route wil berekenen of het aantal handelingen dat een machine moet uitvoeren of iets dergelijks.

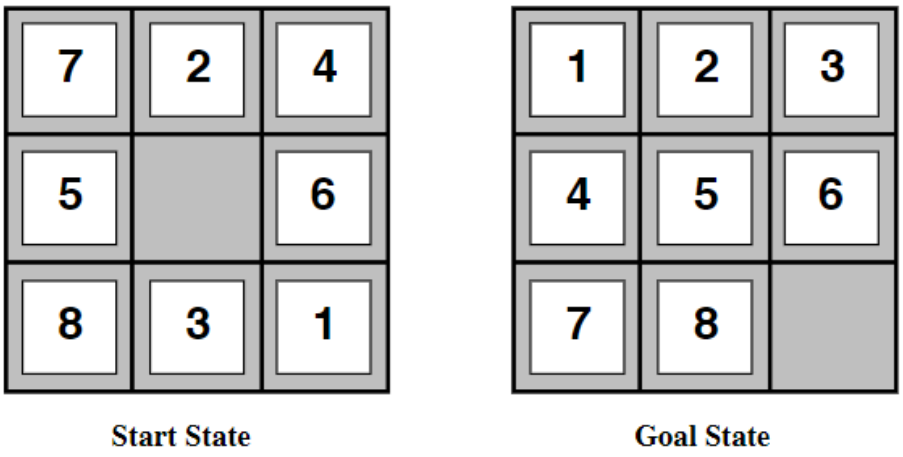

Figure 1: De 8 tile puzzle game

Een belangrijk onderscheid dat gemaakt moet worden is het onderscheid tussen een probleem met verschillende nodes waarbij de oplossing is om een bepaald doel te halen en een probleem met een totaal plaatje so to speak. Het probleem in figuur 1 is in essentie anders dan het probleem in figuur 11. Dit verschil is belangrijk om te weten bij de implementatie van agents.

3.2 Search strategies

Er zijn verschillende soorten search strategies die men kan gebruiken voor agents. Enkele eigenschappen die een search strategie heeft zijn:

- Completeness: Vind het altijd een oplossing als een een oplossing is?
- Tijd complexiteit: Heeft te maken met het aantal nodes dat wordt gegenereerd
- Ruimte complexiteit: Het maximum aantal nodes in het geheugen
- Optimaliteit: vind het altijd een oplossing met de minste kosten?

Tijd en ruimte complexiteit worden gemeten in termen van b, d en m. Die betekenen respectievelijk:

- b: De maximale branching factor van een search tree
- d: diepte van de oplossing met de laagste kosten
- m: de maximale diepte van de state space wat in theorie ∞ kan zijn

Hieronder volgen enkele van de meeste gebruikte search strategieën.

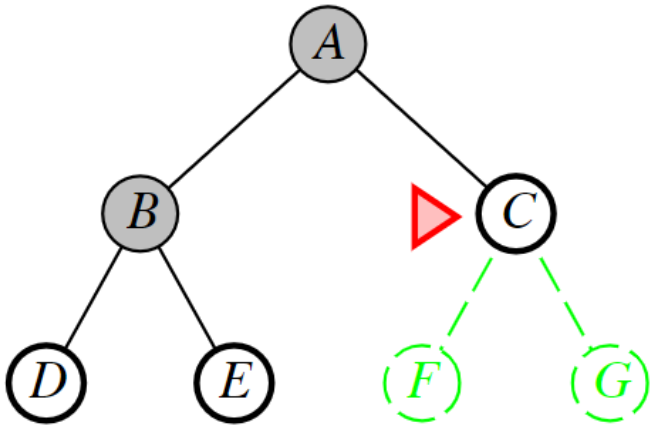

Figure 2: Breadth First Search

3.2.1 Breadth First Search

De naam verklapt al een beetje hoe deze methode zoekt. Deze methode zoekt in de breedte. op het moment dat men alle nodes in figuur 2 moet onderzoeken zal deze methode als volgt te werk gaan.

De initial state is A en de children zijn B en C. Vanuit A zal de methode eerst B (of C afhankelijk van welke kant men op gaat) controller en. Als blijkt dat B het niet is zal de methode de children van B (dat zijn D en E) uitschrijven. Vervolgens controleert hij C. Als C het ook niet is dan is de tweede rij compleet gecontroleerd en gaat de methode naar de derde rij. Dit betekent dat D en E gecontroleerd worden. Als D en E het niet zijn gaat de methode kijken naar de children van C. Als F en G gecontroleerd zijn is het zoeken klaar.

Deze methode is te zien in figuur 2 waarbij de grijze nodes gecontrolleerd zijn, de witte nodes uitgeschreven zijn en de groene nodes nog niet bekend zijn. Let wel dat de uitgeschreven children op het einde van de wachtrij moeten wachten (FIFO) Als B is gecontroleerd komen D en E aan het einde van de wachtrij te staan en wordt eerst C gecontroleerd.

De voordelen van breath first search is dat de zoekmethode compleet en optimaal kan zijn. Verder kost het vergeleken met andere zoekmethoden minder ruimte en kan men redelijk snel een oplossing vinden, al hangt dat wel af van de grootte en de complexiteit van het probleem. Het nadeel is echter dat het veel ruimte kost, omdat er veel children nodes gegenereerd kunnen worden tijdens het zoekproces.

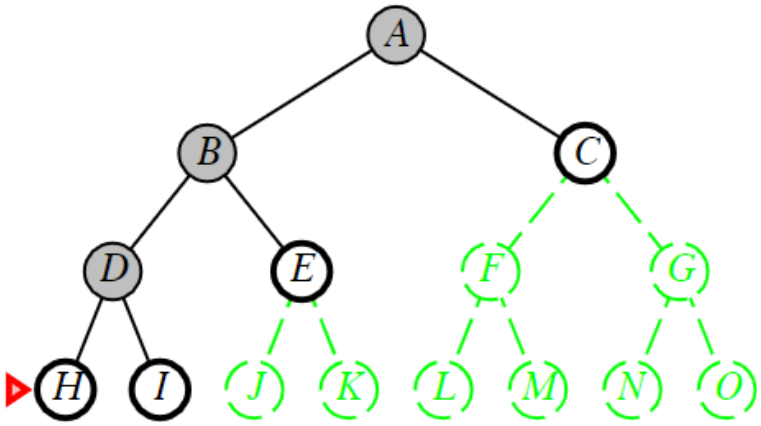

Figure 3: Depth First Search

3.2.2 Depth First Search

Depth first search werkt net iets anders dan breadth first search. Deze zoekmethode gaat niet eerst alle mogelijkheden in de breedte af, maar in de **diepte**, vandaar de naam depth first search. Een illustratie van de depth first search is te zien in figuur 3.

Depth first search heeft als voordeel dat het minder ruimte inneemt dan de breadth first search. Het grote nadeel is echter dat het veel tijd kost om een oplossing te zoeken. Ook is de zoekmethode niet compleet bij een oneindig diep aantal takken met nodes en de zoekmethode is ook niet helemaal optimaal, zie Depth-First in figuur 5.

3.2.3 Itterative Deepening Search

De itteratieve versie van de depth first search is een interessante zoekmethode, omdat het de voordelen heeft van de breadth first search en de depth first search. Deze itteratieve versie is compleet en optimaal en heeft het voordeel als het gaat om tijd van de breadth first search en het voordeel van de hoeveelheid ingenomen ruimte in het geheugen van de depth first search, zie figuur 5. Verder is in figuur 4 dat is dat geval de een oplossing sneller is gevonden bij de itterative deepening search dan bij de breadth first search.

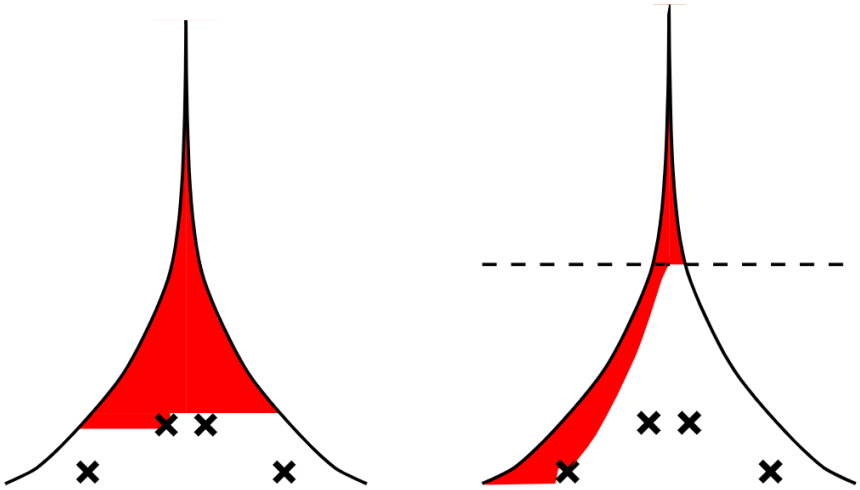

Figure 4: Breadth First Search VS Itterative Deepening Search

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Figure 5: Eigenschappen van zoekalgoritmen

4 Constraint Statisfaction Problems (CSP's)

4.1 Inleiding CSP's

In AI is het **plannen** van een sequence aan acties erg belangrijk. Het pad naar het doel, de kosten voor een bepaald pad en de diepte van een pad zijn erg belangrijk om te weten. Heuristics zijn er om probleem specifieke guidance te geven voor een bepaald probleem.

Bij standaard problemen die worden opgelost is het gebruikelijk om een functie te hebben die kijkt of een bepaald doel is behaald. Constraint Statisfaction Problems is een subset van search problems. Waar normale problemen wellicht kunnen worden opgelost met functies die hard bepalden of een bepaald doel is behaald of niet, worden constraint statisfaction problems opgelost met 'handleidingen' of 'een boek met regels' waar het zich aan moet houden om de constraints te statisfyen.

4.2 Voorbeelden van CSP's

4.2.1 Kleur de provincies

In dit voorbeeld is het de bedoeling om de provincies van een land in te kleuren, in dit geval Australië. Een regel bij het inkleuren van het land is dat er geen provincies mogen zijn die elkaar grenzen met dezelfde kleur. Een voorbeeld van een goede oplossing is te zien in figuur 6.

De manier van aanpak van zo'n probleem is als volgt. Men definieerteerst de variabelen die er zijn. In dit geval zijn dat de staten van het land.

Deze **variabelen** zijn: WA, NT, Q, NSW, V, SA, T

Vervolgens definieert men het domein van het probleem. De staten mogen in dit geval kiezen tussen de kleur rood, groen en blauw. Dit levert ons:

Het **domein D** = $Rood, Groen, Blauw$

De eis is dat adjacente staten niet dezelfde kleuren mogen hebben. Deze constraints die staten hebben kunnen op twee manieren geschreven worden:

Impliciet: WA $\neq$ NT
Expliciet: (WA, NT) $\in$ {(Rood, Groen), (Groen, Blauw), (Blauw, Rood) ...}

De impliciete manier verteld alleen dat de staten WA en NT **niet** dezelfde kleur mogen hebben. De expliciete manier is gedetailleerder en geeft alle mogelijke geldige combinaties die men mag maken.

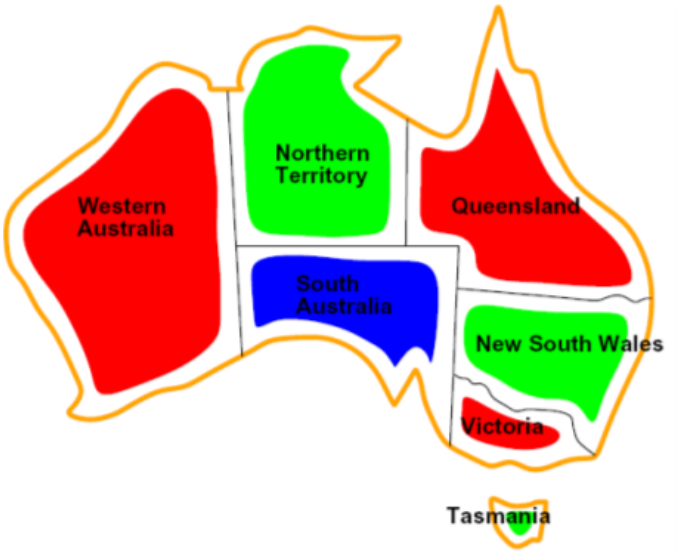

Figure 6: Australië ingekleurd

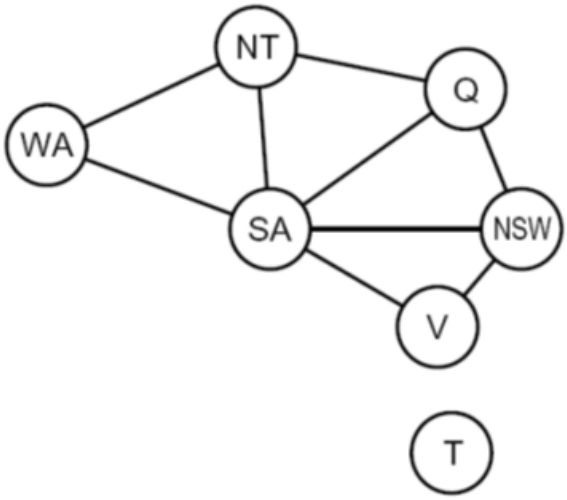

Figure 7: Een grafiek met constrains van de staten van Australië in figuur 6

Figure 8: Schaakbord met koninginnen

4.2.2 Plaats de koninginnen

Een ander voorbeeld van CPS's zijn het plaatsen van koninginnen op een bepaalde plek op een schaakbord. De bedoeling is om de koninginnen zo op het schaakbord te plaatsen dat niemand elkaar kan aanvallen. Een bepaalde formulatie van het probleem kan zijn:

Men heeft voor elk vak in figuur 8 een variabele waar wordt bijgehouden wat er gebeurt op dat vak (staat er een koningin op is is het vak leeg). Het **domein** is {0,1} omdat er op een bepaald vak ofwel een koningin kan staan (1) of niets (0). Verder is het de vraag wat de constrains zijn in dit probleem. De constrains zijn weergegeven in figuur 9.

De constrains kunnen als volgt worden uitgelegd: Voor elk rij i met als kolom j en diagonaal k zijn de **geldige** constrains {(0,0),(0,1),(1,0)}. Dit betekent dat er in dezelfde rij, kolom of diagonaal de volgende situaties geldig zijn:

- Er zijn géén koninginnen (0,0)
- Er is één koningin (0,1),(1,0)

De combinatie (1,1) is hier niet geldig omdat de koninginnen elkaar kunnen aanvallen en dat is niet toegestaan in deze situatie.

Er zijn meerdere manieren om Constraint Statisfaction Problems te formuleren en afhankelijk van een bepaalde situatie kan de afweging gemaakt worden om de ene of de andere manier van formuleren te hanteren. Verder worden bij CSP's gebruik gemaakt van constraint grafieken. Deze worden in het algemeen gebruikt om sneller een oplossing te zoeken. Zo kan men in figuur 7 zien dat T en independent subprobleem is die makkelijk opgelost kan worden.

4.3 CPS's en zoekalgoritmen

In het voorbeeld van Australië (figuur 6) was het de bedoeling dat grenzende staten niet dezelfde kleur hebben. Hier worden vergelijkingen van zoekalgo-

$$\forall i, j, k \;\; (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \;\; (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

Figure 9: De constraints in het koninginnen probleem in figuur 8

ritmes gemaakt om te kijken welke zoekalgoritmes het best passen bij bepaalde problemen.

Wat zou BFS (Breadth First Search) doen in deze situatie? De initial state is een state waarin niks assigned is, het is een lege state so to speak. De children van de initial state zijn nodes die allemaal één assignment hebben, bijvoorbeeld $WA = rood$. Die children hebben weer één extra assignment en ga zo maar door.

De oplossing van dit probleem zit helemaal in de onderste rij en het is dus helemaal niet handig om BFS te gebruiken in deze situatie, omdat de oplossing pas op zijn laatst opgelost wordt. Een betere keuze in dit geval zou de DFS (Depth First Search) zijn, omdat die direct naar het onderste deel van de search tree gaat en in dit geval veel sneller tot een mogelijke oplossing komt.

Een mooie oplossing voor dit soort inkleur problemen is **Backtracking Search**. Backtracking search is een vorm van DFS met twee extra eigenschappen en houdt als het ware live bij of er wordt voldaan aan de voorwaarden en kan een significante rol spelen bij de performance van het oplossen van het probleem. De extra's die backtracking search heeft zijn:

- De volgorde van variabele assignments maakt niet uit: Dus $WA = Rood\ then\ NT = Groen$ en $NT = Groen\ then\ WA = Rood$ zijn hetzelfde
- Checkt de constrains terwijl het bezig is

Er zijn technieken om backtracking search te verbeteren. Enkele technieken daarvoor zijn:

- Algemene versies van een algoritme kunnen een grote rol spelen bij de snelheid
- Het ordenen van de assignments die uitgevoerd moeten worden
- Het eruit filteren van onvermijdelijke failures, zodat die niet weer nagelopen hoeven te worden
- Het kijken naar de constraint grafieken om een beter beeld te krijgen van het probleem en mogelijk sneller een oplossing te vinden zoals in de map van Australië in figuur 6.

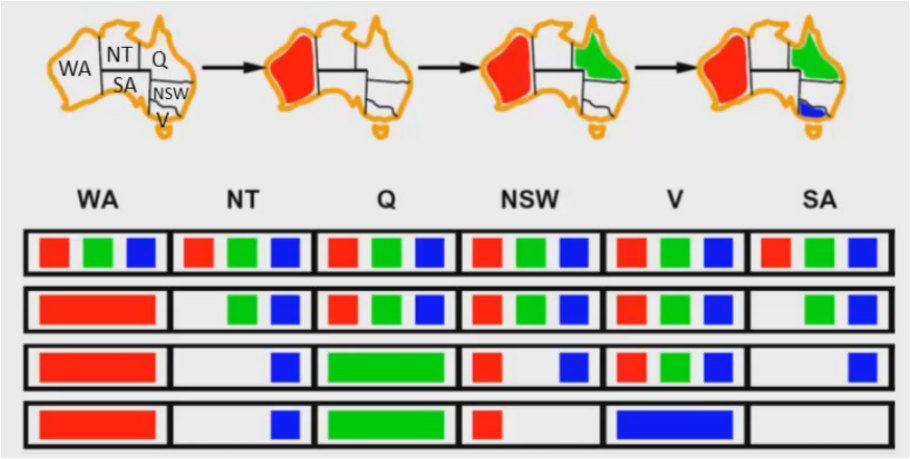

Figure 10: Filtering voor betere performance

Bij het filteren wordt het domein van de variabelen gecheckt en worden de mogelijke opties binnen het domein weggestreept die niet meer geldig zijn om de voldoen aan de eisen die gelden tussen de relaties (constraints). Dat is goed in dit voorbeeld in figuur 10 te zien.

In het voorbeeld in figuur 10 is te zien dat na de vierde stap in het proces de staat SA geen geldige opties binnen het kleurendomein heeft om te kiezen. Er zal dus met behulp van backtracking of andere methoden een paar stappen terug gedaan moeten worden zodanig dat er kan worden voldaan aan de eisen.

Een handige manier om het kleurprocess snel te laten verlopen is door middel van **forward checking**. Forward checking is een manier van **filteren** waarbij de mogelijke geldige opties van het domein van de variabelen gevolgd en bepaalde stappen terug gedaan worden als een variabele geen domein meer over heeft. Het bijhouden van het domein is goed te zien in de balken van figuur 10.

Nog een filtertechniek in de AI is het checken van **arc consistency**. Arc consistency is sneller in het detecteren van failures dan forward checking. Bij Arc cosnsistency worden de domeinen vaker gecheckt op problemen die mogelijk kunnen onstaan als bepaalde beslissingen worden genomen. In het derde balkje in figuur 10 is te zien dat het domein van de staten NT en SA gereduceerd is tot blauw. De staten kunnen niet allebei blauw nemen omdat ze met elkaar grenzen, dit is dus in conflict met de constrains. Het belangrijke bij het checken van arc constrains is dat de buren van een bepaalde variabele, in dit geval een staat, opnieuw moeten worden gecheckt op het domein voordat een beslissing wordt genomen. Dit resulteert in een snellere detectie van een mogelijke failure.

Tot slot is het handig om een bepaalde ordening te hebben van welke staten

eerst ingekleurd worden. **Minimum remaining values (MRV)** is een ordering techniek die eerst de moeilijkste probleem aanpakt, in het geval van map coloring kiest deze ordening methode voor de variabele met het kleinste domein. Door deze ordening techniek toe te passen zijn oplossingen soms veel sneller te vinden dan normaal ...

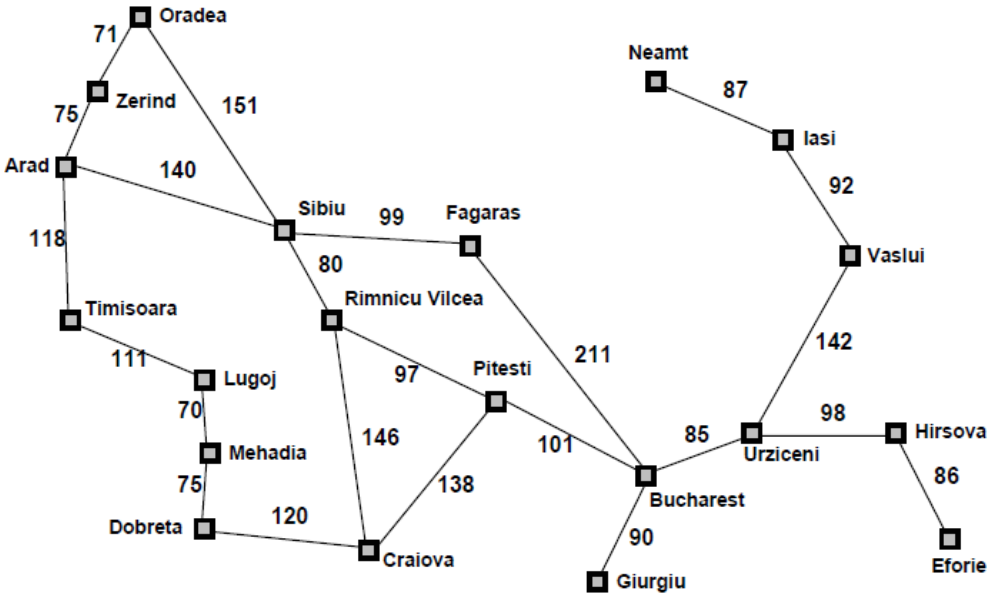

Figure 11: Een map van Roemenië

5 Papers

In dit gedeelte van het verslag wordt er van elke paper een hele korte samenvatting geschreven. De papers die als boekenclub host gedaan hadden moeten worden worden wat uitgebreider samengevat.

5.1 Computing Machinery and Intelligence

5.2 Minds, brains and programs

5.3 AIMA - (informed) Heuristic search

In deze paper worden verschillende zoektechnieken besproken. Verder worden er nog andere algoritmen genoemd en besproken. Hieronder zijn ze kort een voor een terug te vinden. In deze paper worden zoektechnieken en algoritmes uitgelegd aan de hand van graven die betrekking hebben tot het land Roemenië. In figuur 11 is te zien welk voorbeeld wordt gebruikt voor de uitleg van de zoektechnieken en algoritmes.

5.3.1 Informed Search

In de paper wordt onder andere laten zien dat een **informed search** strategie oplossingen efficienter kan vinden dan een uninformed strategie. Een informed

search strategie is een strategie dat specifieke kennis over het probleem **en daarbuiten** gebruikt.

Neem als voorbeeld dat men begint bij Arad in figuur 11 en dat Bucharest de gewilde destinatie is. Bij een informed seach strategie weet men alle plaatsen en afstanden tussen de steden (nodes). Op die manier kan men makkelijk de kortste route vinden van Arad naar Bucharest. Bij deze strategie is men al informed van alle mogelijkheden om het zo te zeggen.

5.3.2 Best-First Search

De **best-first search** is een instance van de generale tree-search en graph-search algoritmen waarbij een opvolgende node wordt gekozen aan de hand van een **evaluation function** $f(n)$. De evaluation functie wordt gezien als een cost estimate, dus de route die het minste 'kost' wordt gekozen. In het voorbeeld van het kiezen van de kortste route betekent het dat vanaf Arad de kortste route wordt gekozen bij best-first search.

De keuze voor f bepaald de gebruikte zoekstrategie. De meeste best-first algoritmen includen een **heuristic functie** als een component van f wat wordt opgeschreven als $h(n)$.

$h(n)$ = de verwachter kosten voor het goedkoopste pad van een state op node n naar een goal state. Als n een goal node is dan is $h(n) = 0$.

Greedy best-first search probeert om nodes te kiezen die het dichtst bij het doel zijn omdat het waarschijnlijk sneller leidt naar het doel. Hier wordt gebruik gemaakt van de heuristic functie, dus $f(n) = h(n)$. Als men begint bij Arad en het doel Bucharest is, dan wordt de volgende route genomen:

Arad $\rightarrow$ **Sibiu** $\rightarrow$ **Faragas** $\rightarrow$ **Bucharest**

Dit is **niet** de kostste route, want die gaat via Rimnicu Vilcea en Pitesti. De greedy best first search gaat, als men hier op het eerste gezicht naar kijkt, via zo min mogelijk nodes naar het doel. Dat is ook de reden waarom hij greedy wordt genoemd.

Deze zoekmethode kan echter voor problemen zorgen. Neem de situatie dat men begint in Iasi en naar Faragas moet. De greedy best-first search zal dan kiezen voor Neamt, omdat de afstand in een rechte lijn (zie figuur 12) tussen Neamt en Faragas kleiner is dan tussen Vaslui en Faragas. Als er is gekozen voor Neamt loopt het dood en springt het algoritme weer terug naar Iasi. Vanuit Iasi zal hij weer kiezen voor Neamt en zo is er een oneindige loop gecreëerd waar het algoritme niet uit kan.

De worst-case tijd en ruimte complexiteit voor de tree versie is $O(b^m)$, waar

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 12: Afstand in een rechte lijn naar Bucharest

m staat voor de maximale diepte aan zoekruimte. Met een goede heuristic functie kan de complexiteit goed naar beneden worden gebracht. Hoeveel het naar beneden kan worden gebracht ligt aan het probleem en de kwaliteit van de heuristische functie.

A* Search, uitgesproken als 'A-star search', is de meest bekende vorm van best-fist search. Het kiest bepaalde nodes door te kijken naar de kosten die gemaakt moeten worden om een node te bereiken en de kosten om van de node naar het doel te gaan: $f(n) = g(n) + h(n)$.

Omdat $g(n)$ de kosten geeft voor het pad van de huidige node naar node n en $h(n)$ ons de verwachte kosten geeft voor het goedkoopste pad van n naar het doel, kan men zeggen dat:

$f(n)$ = de verwachte kosten voor de goedkoopste oplossing via n.

In figuur 13 is te zien hoe een functie, in dit geval de recursieve versie an de best first search, in code kan worden opgebouwd. De code is als het ware half in een programmeertaal en half in het engels, zodat men beter begrijpt wat er gebeurt.

Op regel 1 in figuur 13 is te zien dat dit een functie is die als parameter een probleem heeft en een oplossing of een failure terug kan geven.

5.3.3 Pattern Databases

Pattern databases kunnen uitstekend gebruikt worden voor problemen zoals te zien is in figuur 1. Het complete probeem, het oplossen van de puzzel, bestaat uit kleine sub problemen. Pattern databases bevatten sub oplossingen voor bepaalde patronen. Door sub problemen op te lossen, zoals bijvoorbeel het oplossen van kleinere onderdelen van de puzzel ((1,2,3,4),(2,3,4,5),(3,4,5,6) ...) wordt uiteindelijk het hele probleem opgelost.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

Figure 13: Het algoritme voor de recursieve best-first search

References