Persoonlijk verslag

Muhamed Agic
0946536

April 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 6.

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 6 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 6) 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 6
- 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.

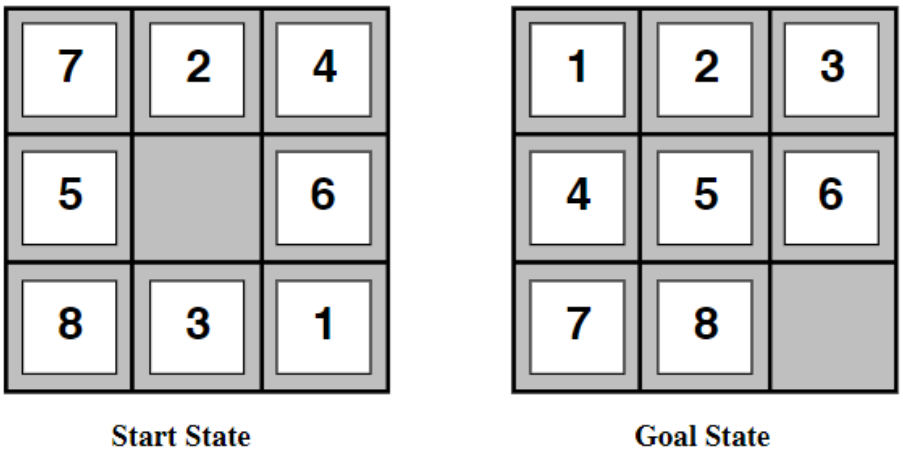

Figure 1: De 8 tile puzzle game

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 6. 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.

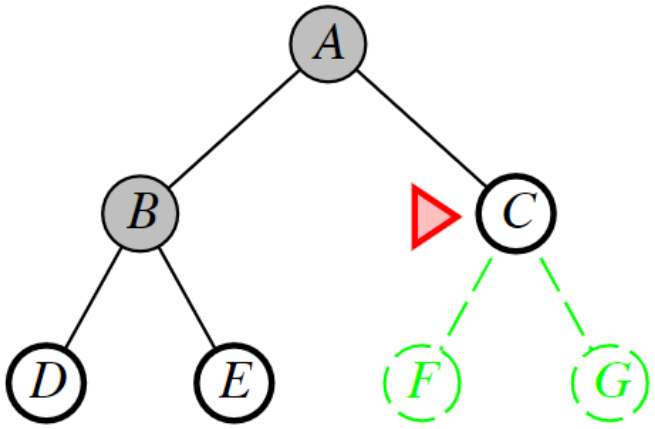

Figure 2: Breadth First Search

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) controlleren. Als blijkt dat B het niet is zal de methode de children van B (dat zijn D en E) uitschrijven. Vervolgens controlleert hij C. Als C het ook niet is dan is de tweede rij compleet gecontrolleerd en gaat de methode naar de derde rij. Dit betekent dat D en E gecontrolleerd worden. Als D en E het niet zijn gaat de methode kijken naar de children van C. Als F en G gecontrollerd zijn is het zoeken klaar.

Deze methode is te zien in figuur 2 waarbij de grijze nodes gecontrolleerd zijn, de wite nodes uitgeschreven zijn en de groene nodes nog niet bekend zijn.

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.

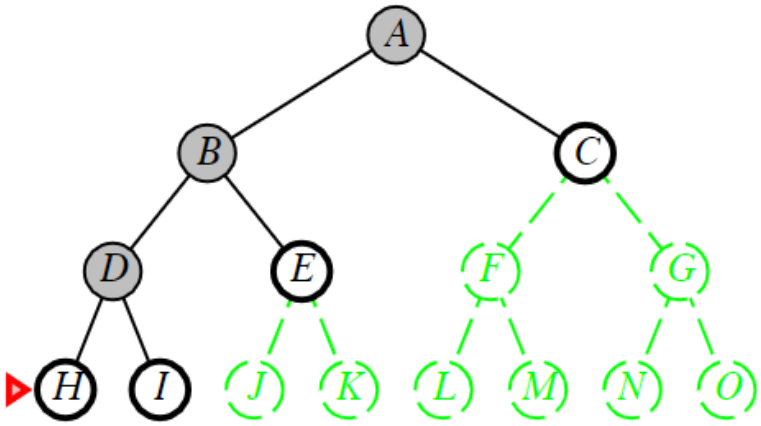

Figure 3: Depth First Search

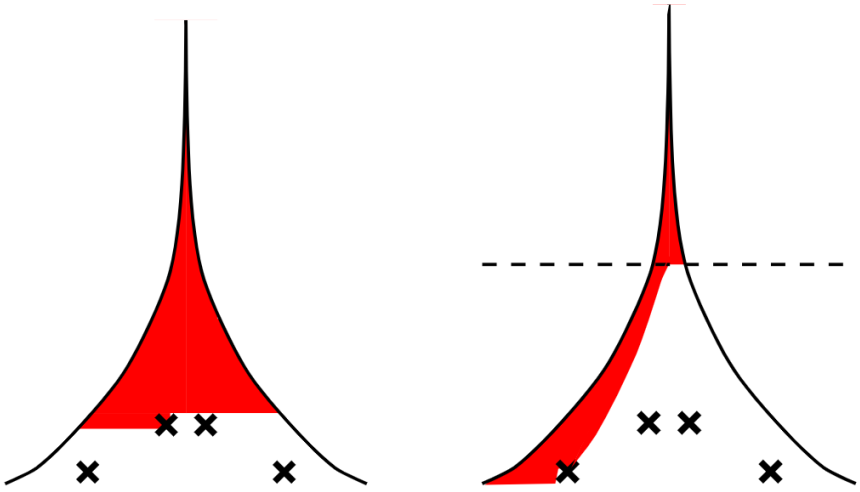

Figure 4: Breadth First Search VS Itterative Deepening Search

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 Papers

In dit gedeelte van het verslag wordt er van elke paper een korte samenvatting geschreven.

4.1 Computing Machinery and Intelligence

4.2 Minds, brains and programs

4.3 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 6 is te zien welk voorbeeld wordt gebruikt voor de uitleg van de zoektechnieken en algoritmes.

4.3.1 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 6 en dat Bucharest de gewilde destinatie is. Bij een informed seach strategie weet men alle plaatsen

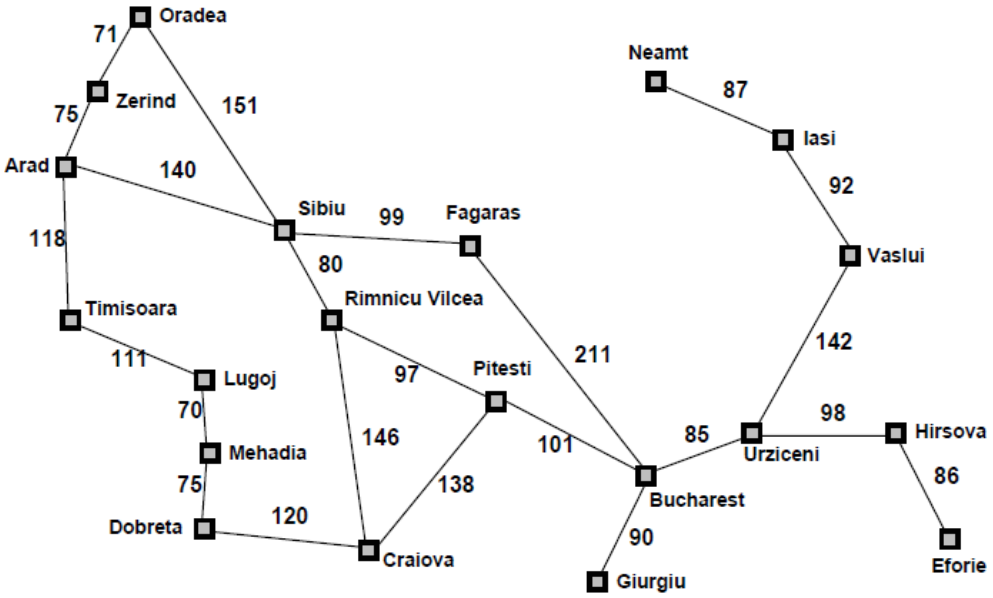

Figure 6: Een map van Roemenië

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.

4.3.2 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	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 7: Afstand in een rechte lijn naar Bucharest

Arad → Sibiu → Faragas → 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 7) 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 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 8 is te zien hoe een functie, in dit geval de recursieve versie an de

```
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 8: Het algoritme voor de recursieve best-first search

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 8 is te zien dat dit een functie is die als parameter een probleem heeft en een oplossing of een failure terug kan geven.

4.3.3 Pattern Databases

4.4

4.5

4.6

References