HoGent
BEDRIJF
EN
ORGANISATIE

Artificiële Intelligentie

Lesnota's

Stijn Lievens

Professionele Bachelor in de Toegepaste Informatica

Inhoudsopgave

HOOFDSTUK 2

Zoekalgoritmes

2.1 Inleiding

In dit hoofdstuk bespreken we hoe een agent een zoekprobleem kan oplossen. Voor een zoekprobleem nemen we aan dat de agent zich bevindt in een eenpersoons omgeving die compleet observeerbaar, deterministisch, statisch en discreet is. De agent bevindt zich steeds in een bepaalde beginpositie en de bedoeling is dat de agent acties onderneemt die hem in een toestand brengen waar één of andere voorwaarde voldaan is.

Het feit dat de omgeving aan alle bovenstaande voorwaarden voldoet betekent dat de agent *op voorhand* de acties kan berekenen die hem naar het doel zullen brengen zonder deze effectief uit te moeten voeren. Op het moment dat de agent dan effectief de acties gaat uitvoeren in de "echte" wereld kan hij de waarnemingen negeren: hij weet immers toch al wat er gaat komen.

Definitie 2.1 Een ZOEKPROBLEEM bestaat uit de volgende elementen:

- Een TOESTANDSRUIMTE S die alle mogelijke toestanden bevat.
- Een verzameling van mogelijke acties A.
- Een TRANSITIEMODEL dat zegt wat het effect is van het uitvoeren van een actie op een bepaalde toestand:

$$T\colon (S, A) \to S\colon (s, a) \mapsto s'.$$

Wanneer s' bereikt wordt door het uitvoeren van een actie a op een toestand s dan wordt s' een *opvolger* van s genoemd[1].

Het uitvoeren van een actie op een bepaalde toestand heeft meestal een bepaalde KOST:

$$C\colon (S, A) \to \mathbb{R}\colon (s, a) \mapsto c.$$

- Een initiële toestand $s_0 \in S$, dit is de toestand van waaruit het zoeken zal vertrekken.
- een DOELTEST. Dit is een functie die voor elke toestand s aangeeft of het doel bereikt is of niet. Een toestand waarvoor de doeltest voldaan is noemen we een DOELTOESTAND. ∎

Opmerking 2.2 De toestandsruimte S kan samen met de transitie- en kostfunctie gebruikt worden om een TOESTANDSRUIMTEGRAAF (Eng. *state space graph*) G op te bouwen. In deze graaf stellen de knopen de toestanden voor en twee knopen zijn verbonden door een (gerichte) boog wanneer de ene knoop de opvolger is van de andere door het uitvoeren van een bepaalde actie. De kost (of het gewicht) van een boog is dan uiteraard de kost van de bijhorende actie.

In een toestandsruimtegraaf komt elke toestand juist één keer voor.

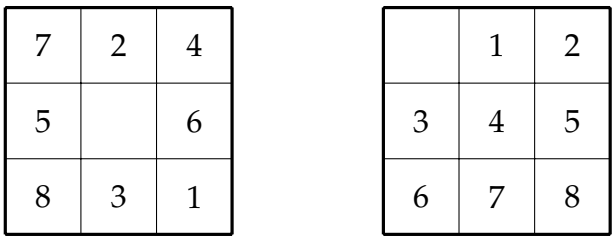
De toestandsruimtegraaf is een abstract concept: het aantal knopen (en bogen) van deze graaf is vaak veel te groot om deze volledig bij te houden in het (hoofd)geheugen van een computer. ∎

Voorbeeld 2.3 (De 8-puzzel) De 8-puzzel is een typisch zoekprobleem. Hierbij is een vierkant rooster gegeven van 9 vakjes: 8 vakjes zijn genummerd van 1 t.e.m. 8 en één vakje is leeg. Men kan een genummerd vakje verschuiven naar het lege vakje, waardoor de originele plaats van het genummerd vakje leeg wordt. De bedoeling is om een vooraf bepaalde "mooie" configuratie te bereiken. In Figuur 2.1 ziet men twee voorbeelden van 8-puzzels.

We beschrijven de verschillende onderdelen van het zoekprobleem:

- De verzameling S bestaat uit alle mogelijke configuraties van de puzzel.

[1] Hier zie je dat we te maken hebben met een deterministische omgeving: wanneer s en a gekend zijn is er juist één opvolger s'.



Figuur 2.1: Een voorbeeld van een 8-puzzel. Links ziet men een typische beginconfiguratie (initële toestand) en rechts ziet men de doeltoestand.

- Er zijn vier acties, nl. `Boven`, `Onder`, `Links` en `Rechts`. Deze worden beschreven in termen van het lege vakje.
- Het transitiemodel is een eenvoudige vertaling van wat er gebeurt in de fysieke puzzel. Wanneer men bv. de actie `Rechts` uitvoert op de linkerpuzzel in Figuur 2.1 dan bekomt men een nieuwe puzzel waarbij het lege vakje en vakje 6 omgewisseld zijn. De kost van elke actie is één.
- De initiële toestand is een willekeurige configuratie van de puzzel[2].
- De doeltest bestaat uit verifiëren of de vooraf vastgelegde configuratie werd bereikt.

Het aantal toestanden voor de 8-puzzel is $9! = 362880$ wat voor een moderne computer nog beheersbaar is. Men kan de puzzel echter ook uitbreiden naar grotere borden, bv. 4 op 4 (de 15-puzzel) met $16! \approx 2.1 \times 10^{13}$ of 5 op 5 (de 24-puzzel) met $25! \approx 1.6 \times 10^{25}$ toestanden. ∎

Voorbeeld 2.4 (Het 8-koninginnenprobleem) Het doel van het 8-koninginnenprobleem is om 8 koninginnen op een standaard 8 op 8 schaakbord te plaatsen zodanig dat geen enkel paar koninginnen elkaar aanvalt volgens de standaard schaakregels waarin een koningin horizontaal, verticaal en diagonaal kan bewegen. In Figuur 2.2 ziet men een oplossing van het 8-koninginnenprobleem.

[2]Echter, slechts de helft van de puzzels kan omgezet worden in een gegeven doelconfiguratie.

We bekijken nu een eerste manier om dit probleem voor te stellen als een zoekprobleem, waarbij we met een leeg bord beginnen en telkens een koningin toevoegen aan het bord.

- De toestandsruimte S bevat alle mogelijke configuraties waarbij er tussen de 0 en 8 koninginnen op het bord staan.
- Wanneer er minder dan 8 koninginnen op het bord staan dan kunnen we op een willekeurig leeg vakje een koningin toevoegen; dit zijn de acties. Het transitiemodel wordt gegeven door de fysica van het model en de kost van de acties is voor dit probleem irrelevant en kan dus gelijk aan nul worden genomen.
- De initiële toestand is het lege bord.
- De doeltest bestaat uit nagaan of er inderdaad 8 koninginnen op het bord staan én uit controleren dat er geen paar koninginnen is dat elkaar aanvalt. In dit geval is de doeltest effectief een functie en niet zomaar een eenvoudige opsomming van de doeltoestanden.

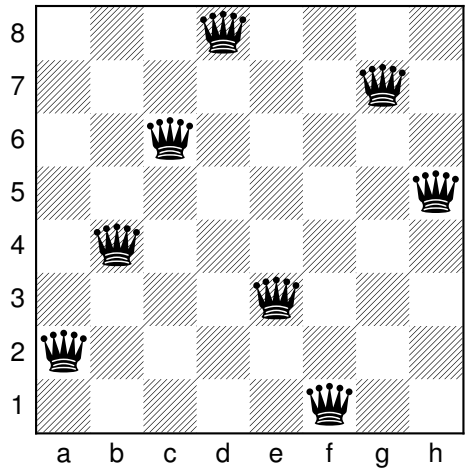
Met deze representatie is het aantal toestanden in de toestandsruimte

$$\binom{64}{0} + \binom{64}{1} + \cdots + \binom{64}{8} \approx 5.13 \times 10^9,$$

wat met 8 bytes per toestand reeds 40 GB aan hoofdgeheugen vraagt! ∎

Voorbeeld 2.5 (Het 8-koninginnenprobleem (bis)) Een efficiëntere representatie bestaat erin om toestanden waarin twee koninginnen elkaar aanvallen te verbieden. In de vorige representatie worden eerst alle acht de koninginnen geplaatst en pas dan wordt gecontroleerd of deze configuratie voldoet aan de doeltest. Om het aantal toestanden verder te beperken gaan we koninginnen kolom per kolom toevoegen, startend vanaf links. We bekomen de volgende formulering als een zoekprobleem.

- De toestandsruimte bevat toestanden waarbij de eerste n (met $0 \leq n \leq 8$) kolommen van het bord juist één koningin bevatten. Bovendien is er geen paar koninginnen dat elkaar aanvalt.
- De mogelijke acties op een gegeven bord bestaan erin om een koningin toe te voegen in de eerste vrije kolom op een vakje dat niet wordt



Figuur 2.2: Eén van de 92 oplossingen van het 8-koninginnenprobleem.

aangevallen door één van de reeds geplaatste koninginnen. Het transitiemodel is zoals men verwacht en de kost van elke actie is nog steeds gelijk aan nul.

- De initiële toestand is het lege bord.
- De doeltest is zoals voorheen.

In dit geval bevat de toestandsruimte 2057 toestanden en wordt het triviaal om een oplossing te vinden.

Men ziet dus dat de formulering van de toestanden en acties een grote invloed kan hebben op de mogelijkheid om al dan niet een oplossing te vinden. ∎

Voorbeeld 2.6 (Route- en rondreisproblemen) Veel "real-life" hebben te maken met het vinden van een route tussen twee locaties. Veronderstel dat er n steden zijn: in het *routeprobleem* wil men een weg vinden tussen twee van deze steden.

- Er zijn n toestanden, de toestand i duidt aan dat "we in stad i zijn".
- Vanuit een bepaalde stad kan men acties ondernemen om een andere stad te bereiken, bv. de `RijdNaar` actie.

- Wanneer men in toestand i de `RijdNaar`(j) actie onderneemt, dan is de volgende toestand die waarin men zich in locatie j bevindt. Dit is het transitiemodel. De kost van de actie kan bv. de afstand in km zijn tussen i en j, of de gemiddelde reistijd of het gemiddeld benzineverbruik, ...
- De initiële toestand is een willekeurige stad.
- De doeltest bestaat uit verifiëren dat men in de gewenste stad is aangekomen.

Voor het routeprobleem is het aantal toestanden dus gelijk aan het aantal steden.

Bij een *rondreisprobleem*, waarbij men elke stad minstens éénmaal moet bezoeken en terug moet keren naar de eerste stad, zijn de toestanden fundamenteel anders. Het is immers niet langer voldoende om te weten in welke locatie men *nu* is, men moet ook onthouden *waar men reeds geweest is*.

- Een toestand bestaat uit de huidige stad, en de verzameling van de reeds bezochte steden. Het aantal toestanden is dus $n \times 2^n$. Nu is het aantal toestanden exponentieel in het aantal steden.
- De acties zijn dezelfde als voorheen.
- In het transitiemodel moet men niet enkel de huidige stad aanpassen maar ook de verzameling van reeds bezochte steden uitbreiden met de stad waar men net vandaan komt. De kost van de acties is zoals voorheen.
- Initieel zijn we in een willekeurige stad en hebben we geen enkele andere stad bezocht.
- De doeltest bestaat uit verifiëren dat we opnieuw in de startstad zijn en dat alle steden minstens éénmaal bezocht zijn.

Bij het *handelsreizigersprobleem* moet elke tussenliggende stad juist éénmaal worden bezocht. ∎

We hebben nu enkele voorbeelden gezien van de definitie van een zoekprobleem. We bekijken nu wat het betekent om een zoekprobleem op te lossen.

Definitie 2.7 Een OPLOSSING VAN ZOEKPROBLEEM bestaat uit een sequentie van acties zodanig dat startend vanuit de initiële toestand een doeltoestand wordt bereikt.

De KOST van een oplossing is de som van de kosten van de individuele acties.

Een OPTIMALE OPLOSSING is een oplossing waarvoor de kost minimaal is onder alle mogelijke oplossingen. ∎

Opmerking 2.8 Het oplossen van een n op n puzzel en het handelsreizigersprobleem zijn NP-complete problemen die kunnen geformuleerd worden als zoekproblemen. We kunnen m.a.w. *niet* verwachten dat de oplossingsmethodes binnen de artificiële intelligentie een algoritme zullen opleveren dat alle instanties efficiënt kan oplossen. ∎

Voorbeeld 2.9 Voor de 8-puzzel in Figuur 2.1 bestaat een optimale uit 26 acties. Een mogelijke oplossing, met `L` als afkorting voor actie `Links` en analoog voor de andere acties, is de volgende:

`L, B, R, O, R, O, L, L,B, R, R, O, L, L, B, R, R, B, L, L, O, R, R,B, L, L` ∎

2.2 Algemene Zoekalgoritmen

2.2.1 Boomgebaseerd Zoeken

Het algemene algoritme gekend als BOOMGEBASEERD ZOEKEN houdt een lijst bij van mogelijke partiële oplossingen (plannen) die nog verder uitgewerkt (geëxpandeerd) moeten worden. Deze lijst wordt de OPEN LIJST genoemd. Bij de start van de uitvoering bestaat deze open lijst enkel uit het plan corresponderend met de initiële toestand van het zoekprobleem. Bij elke iteratie van het algoritme wordt een plan gekozen uit deze lijst (volgens één of andere strategie). Wanneer de (eind)toestand van het gekozen plan voldoet aan de doeltest dan stopt het algoritme. Wanneer dit niet het geval is dan worden de plannen voor alle opvolgers van de (eind)toestand van het gekozen plan toegevoegd aan de open lijst (waardoor deze ook beschikbaar worden voor expansie). Wanneer de open lijst op een bepaald moment leeg is dan geeft het algoritme aan dat er geen oplossing werd gevonden.

Conceptueel bouwen we dus een ZOEKBOOM op. Elke top van deze zoekboom stelt een sequentie van acties voor. Het is uiteraard de bedoeling om de zoekproblemen op te lossen aan de hand van een zo klein mogelijke zoekboom. In een zoekboom stelt elke top een plan voor dat o.a. de huidige toestand bijhoudt. Dezelfde toestand kan (en zal) in het algemeen *meerdere malen* voorkomen in een zoekboom.

Voorbeeld 2.10 In Figuur 2.3 ziet men de start van een zoekboom voor de 8-puzzel. Hier ziet men reeds dat dezelfde toestand meerdere malen kan voorkomen op de open lijst. ∎

Algoritme 2.1 Boomgebaseerd zoeken.

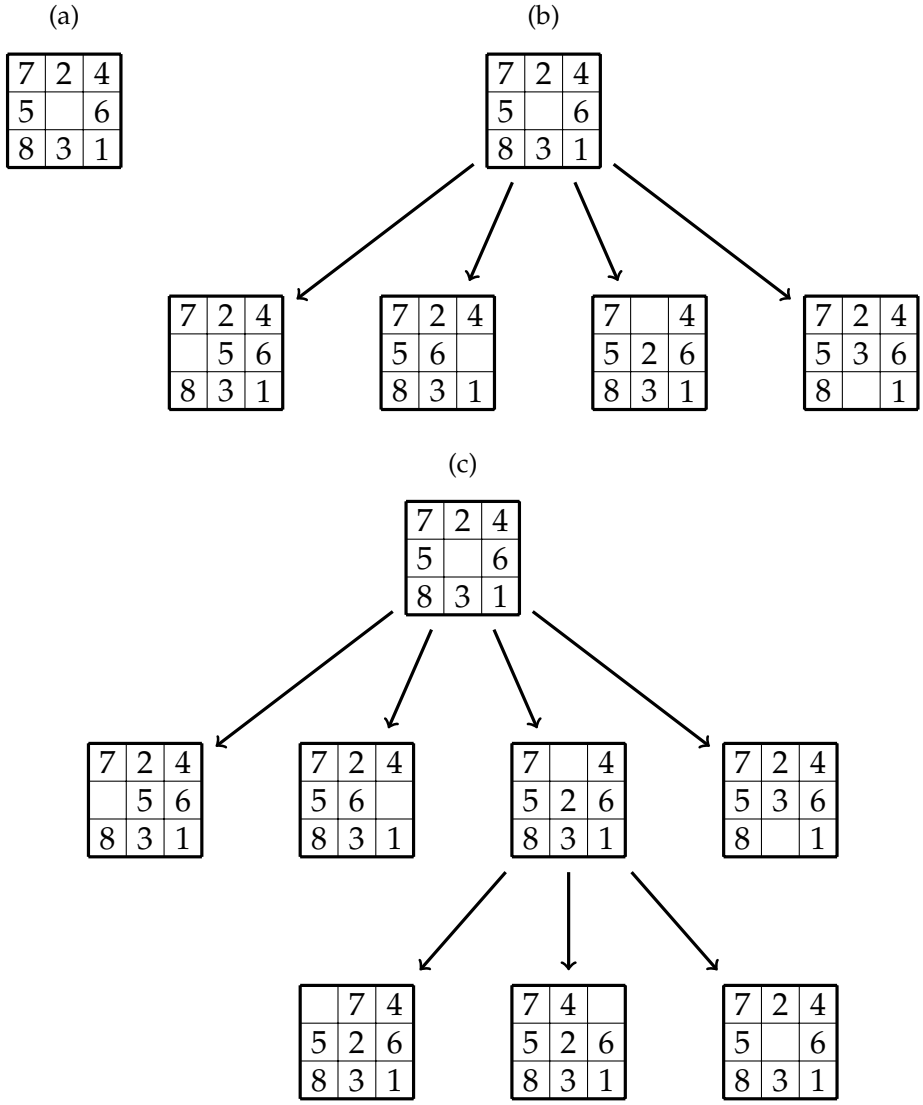
Invoer Een zoekprobleem P.
Uitvoer Een sequentie van acties die een oplossing is van het zoekprobleem of error wanneer er geen oplossing werd gevonden.

```
1:  function TREESEARCH(P)
2:      f ← nieuwe lege lijst                          ▷ De open lijst
3:      f.ADD(nieuw plan gebaseerd op initiële toestand P)
4:      while f ≠ ∅ do
5:          c ← f.CHOOSEANDREMOVEPLAN                  ▷ Kies het volgende plan
6:          if P.GOALTEST(c.GETSTATE) = true then
7:              return GETACTIONSEQ(c)
8:          else
9:              for (s, a) ∈ c.GETSTATE.GETSUCCESSORS do
10:                 f.ADD(nieuw plan gebaseerd op (s, a) en c)
11:             end for
12:         end if
13:     end while
14:     return error: geen oplossing gevonden          ▷ Open lijst is leeg.
15: end function
```

Implementatie van een plan

Conceptueel stelt elke top van de zoekboom een sequentie van acties voor om een bepaalde toestand te bereiken startend vanaf de initiële toestand van het zoekprobleem.

Het is echter *niet nodig* om in elke top het volledige pad op te slaan. Zolang we weten wat het vorige plan is kunnen we, in combinatie met de laatst gekozen actie, het volledige plan opstellen.



Figuur 2.3: Deel van de zoekboom opgebouwd door boomgebaseerd zoeken. We kiezen "willekeurig" welk plan op de open lijst (i.e. welk blad) wordt geëxpandeerd. Initieel (zie (a)) bevat de open lijst slechts één plan. Dit plan wordt verwijderd van de open lijst en wordt geëxpandeerd. De vier opvolgers van de initiële toestand worden toegevoegd aan de open lijst (zie (b)). Veronderstel nu dat (om één of andere reden) de toestand met de blanco bovenaan wordt gekozen voor expansie. De drie opvolgers worden toegevoegd aan de open lijst die nu 6 elementen bevat (zie (c)). Merk op dat één van de gegenereerde opvolgers opnieuw de initiële toestand is. Bij boomgebaseerd zoeken wordt hier geen rekening mee gehouden en wordt deze toestand opnieuw toegevoegd aan de open lijst.

We gebruiken een klasse `Plan` om een plan voor te stellen. Zo'n `Plan` bestaat uit vier velden:

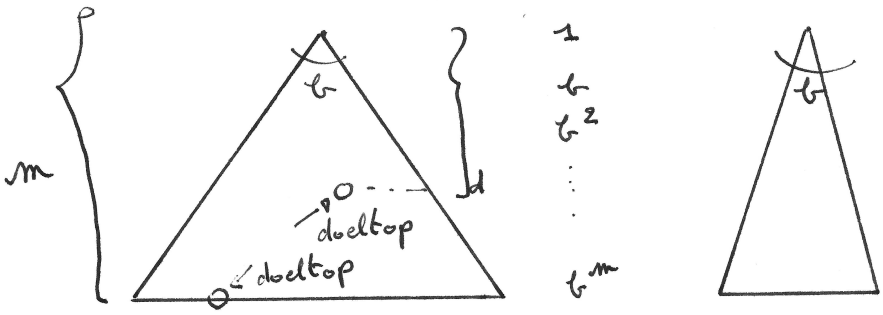
- De huidige toestand.
- De laatst gekozen actie a; deze is enkel leeg voor het plan geassocieerd met de initiële toestand.
- De *voorganger* of *ouder* van dit plan. Een referentie naar het plan waarvan dit plan is afgeleid door het toepassen van de huidige actie a.
- De totale kost van dit plan. Traditioneel wordt deze kost met g genoteerd. Strikt genomen kunnen we deze kost ook berekenen door het volgen van de voorganger-referenties. Deze berekening heeft echter een uitvoeringstijd die lineair is in het aantal acties van het plan.

2.2.2 Criteria voor Zoekalgoritmen

Zoekalgoritmen kunnen op verschillende manieren worden geëvalueerd. De volgende vier criteria worden vaak gebruikt.

1. Een zoekalgoritme is COMPLEET wanneer het algoritme, voor elk zoekprobleem met een oplossing, effectief een oplossing vindt.
2. Een zoekalgoritme is OPTIMAAL wanneer het niet enkel *een* oplossing vindt maar steeds een optimale oplossing teruggeeft voor elk zoekprobleem met een oplossing.
3. De TIJDSCOMPLEXITEIT van een zoekalgoritme bepaalt de uitvoeringstijd van het algoritme. We nemen aan dat de uitvoeringstijd evenredig is met het aantal gegenereerde toppen.
4. De RUIMTECOMPLEXITEIT van een zoekalgoritme bepaalt de hoeveelheid geheugen die het algoritme nodig heeft tijdens de uitvoering. Dit wordt meestal uitgedrukt als het maximaal aantal toestanden dat gelijktijdig moet worden bijgehouden.

De volgende maten worden vaak gebruikt om de tijds- en ruimtecomplexiteit van zoekalgoritmen uit te drukken. Vooreerst is er de VERTAKKINGSFACTOR b. Deze geeft het maximaal aantal opvolgers van een top in de



Figuur 2.4: Illustratie van de grootheden b, d en m voor een zoekboom. Links een voorbeeld van een zoekboom met een "grote" vertakkingsfactor, rechts een zoekboom met een "kleine" vertakkingsfactor.

zoekboom. De diepte van de meest ondiepe top waarvan de toestand een doeltoestand is (kortweg een *doeltop* genoemd) wordt genoteerd met d. Met m, tenslotte, duidt men de maximale lengte (gemeten als het aantal genomen acties) van een pad in de toestandsruimte aan.

In Figuur 2.4 ziet men een illustratie van deze concepten.

Eigenschap 2.11 Het aantal toppen in een zoekboom met vertakkingsfactor b en maximale diepte m wordt gegeven door

$$\frac{b^{m+1}-1}{b-1} = \mathcal{O}(b^m). \tag{2.1}$$

∎

Bewijs We nemen aan dat elke top exact b opvolgers heeft.

Het aantal toppen op diepte nul is 1, het aantal toppen op diepte 1 is b, het aantal toppen op diepte 2 is b^2, enzovoort. Het totaal aantal toppen is dus

$$1 + b + b^2 + \cdots + b^m.$$

Een gesloten formule voor deze som wordt inderdaad gegeven door (2.1).♢

Opmerking 2.12 De laag met diepte m in een zoekboom met vertakkingsfactor b bevat b^m toppen. Dit is méér dan alle voorgaande lagen samen; deze bevatten in totaal slechts

$$1 + b + b^2 + \cdots + b^{m-1} = \frac{b^m - 1}{b-1} \approx b^{m-1}$$

toppen. ∎

			3			
		3	2	3		
	3	2	1	2	3	
3	2	1	0	1	2	3
	3	2	1	2	3	
		3	2	3		
			3			

Figuur 2.5: Toestanden in een grid. Centraal staat de starttoestand. Voor toestanden die op afstand drie of minder liggen staat de afstand tot de initiële toestand aangeduid in de desbetreffende cel.

2.2.3 Graafgebaseerd zoeken

Het grootste probleem van boomgebaseerd zoeken is dat dit algoritme *niet onthoudt waar het reeds geweest is*. Dit zorgt ervoor dat we in sommige gevallen (bv. diepte eerst zoeken, cfr. infra) te maken krijgen met *oneindige lussen*, en dat we in veel andere gevallen een grote hoeveelheid werk herhaaldelijk uitvoeren.

Voorbeeld 2.13 (Herhaalde toestanden in een grid) Een vaak voorkomende situatie is die van een agent die leeft in een grid met vier acties: `Boven`, `Onder`, `Links` en `Rechts`. De branching factor b is in dit geval gelijk aan 4. Vergelijken we nu eventjes het aantal toestanden op afstand d van een willekeurig toestand met het aantal toppen van de zoekboom op diepte d bij boomgebaseerd zoeken. Figuur 2.5 toont een grid. We kunnen nu gemakkelijk vergelijken:

d	aantal toestanden op afstand d	aantal toppen in zoekboom op diepte d
0	1	1
1	4	4
2	8	16
3	12	64
$\vdots$	$\vdots$	$\vdots$
d	$4d$	4^d

Uit deze tabel zien we dat het aantal verschillende toestanden *lineair* toe-

neemt met de diepte, terwijl het aantal toppen in de zoekboom *exponentieel* toeneemt met de diepte. ∎

De oplossing voor het probleem van de herhaalde toestanden bestaat erin om eenvoudigweg te onthouden welke toestanden reeds geëxpandeerd zijn in, wat men noemt, een GESLOTEN LIJST[3]. Merk op dat de gesloten lijst *toestanden* bevat terwijl de open lijst *plannen* bevat.

Bij GRAAFGEBASEERD ZOEKEN wordt elke toestand hoogstens éénmaal geëxpandeerd. Wanneer een plan van de open lijst wordt gehaald dat een toestand bevat die reeds geëxpandeerd is, dan wordt deze niet opnieuw geëxpandeerd. De pseudocode voor graafgebaseerd zoeken vind je in Algoritme 2.2.

Algoritme 2.2 Graafgebaseerd zoeken.

Invoer Een zoekprobleem P.
Uitvoer Een sequentie van acties of error wanneer er geen oplossing werd gevonden.

```
1:  function GRAPHSEARCH(P)
2:      f ← nieuwe lege lijst                          ▷ De open lijst
3:      closed ← ∅                  ▷ Verzameling geëxpandeerde toestanden
4:      f.ADD(nieuw plan gebaseerd op initiële toestand P)
5:      while f ≠ ∅ do
6:          c ← f.CHOOSEANDREMOVEPLAN                  ▷ Kies het volgende plan
7:          if P.GOALTEST(c.GETSTATE) = true then
8:              return GETACTIONSEQ(c)
9:          else
10:             if c.GETSTATE ∉ closed then
11:                 closed ← closed ∪ c.GETSTATE
12:                 for (s, a) ∈ c.GETSTATE.GETSUCCESSORS do
13:                     f.ADD(nieuw plan gebaseerd op (s, a) en c)
14:                 end for
15:             end if
16:         end if
17:     end while
18:     return error: geen oplossing gevonden
19: end function
```

[3]De gesloten lijst wordt best geïmplementeerd als een verzameling.

2.3 Blinde Zoekmethoden

Blinde zoekmethoden kunnen enkel gebruikmaken van de informatie die verschaft wordt door de definitie van het zoekprobleem. Ze beschikken niet over extra informatie die hen kan helpen bij het zoekproces. We bespreken nu vier blinde zoekmethoden en hun eigenschappen.

2.3.1 Breedte Eerst Zoeken

Bij breedte eerst zoeken wordt voor de open lijst een *wachtrij* gebruikt. Dit is een FIFO data structuur. Bij breedte eerst wordt de zoekboom systematisch laag per laag opgebouwd. In Figuur 2.6 ziet men een illustratie van het breedte eerst algoritme op een binaire boom.

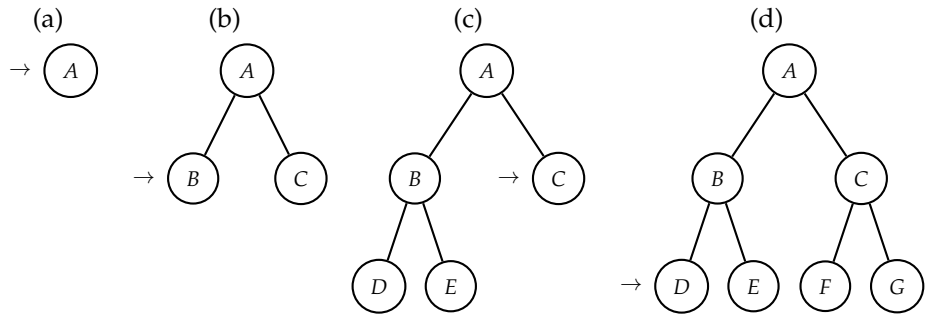
Omdat breedte eerst systematisch de lagen in de zoekboom onderzoekt zal het algoritme steeds een oplossing vinden voor elk zoekprobleem dat effectief een oplossing heeft. Dit betekent dat breedte eerst een compleet zoekalgoritme is. Het algoritme vindt steeds de meest ondiepe doeltop, i.e. het retourneert een oplossing met een minimaal *aantal* acties. Wanneer acties een verschillende kost hebben is dit niet noodzakelijk een oplossing met de kleinste kost. Het kan immers beter zijn om meerdere goedkope acties te doen i.p.v. een kleiner aantal duurdere acties. Breedte eerst is m.a.w. niet optimaal. Breedte eerst is wel optimaal in het bijzonder geval dat alle acties dezelfde kost hebben.

We onderzoeken nu de tijdscomplexiteit van breedte eerst zoeken. Veronderstel dat het zoekprobleem een oplossing heeft en dat de meest ondiepe doelknoop diepte d heeft. Als deze doeltop de "meest rechtse" top is dan worden alle toppen op de dieptes 0 t.e.m. d geëxpandeerd. Het aantal *gegenereerde* toppen is m.a.w. (op een term b na) gelijk aan

$$1 + b + b^2 + \cdots + b^d + b^{d+1} = \mathcal{O}(b^{d+1}).$$

De tijdscomplexiteit van breedte eerst is m.a.w. exponentieel in de diepte van de meest ondiepe doeltop.

Het maximaal aantal toppen dat moet worden bijgehouden in de open lijst wordt bereikt wanneer men de doeltop expandeert. Op dit moment wordt zo goed als de volledige laag op diepte $d + 1$ bijgehouden in de open lijst.



Figuur 2.6: Illustratie van breedte eerst zoeken op een binaire boom. De knoop die zal geëxpandeerd worden is steeds aangeduid met een pijltje. De doelknoop is D. Net vóór het algoritme eindigt bestaat de open lijst uit de volgende plannen: $\{D \to B \to A, E \to B \to A, F \to C \to A, G \to C \to A\}$ (in die volgorde).

Deze laag bevat b^{d+1} toppen[4]. De ruimtecomplexiteit is m.a.w. eveneens $\mathcal{O}(b^{d+1})$.

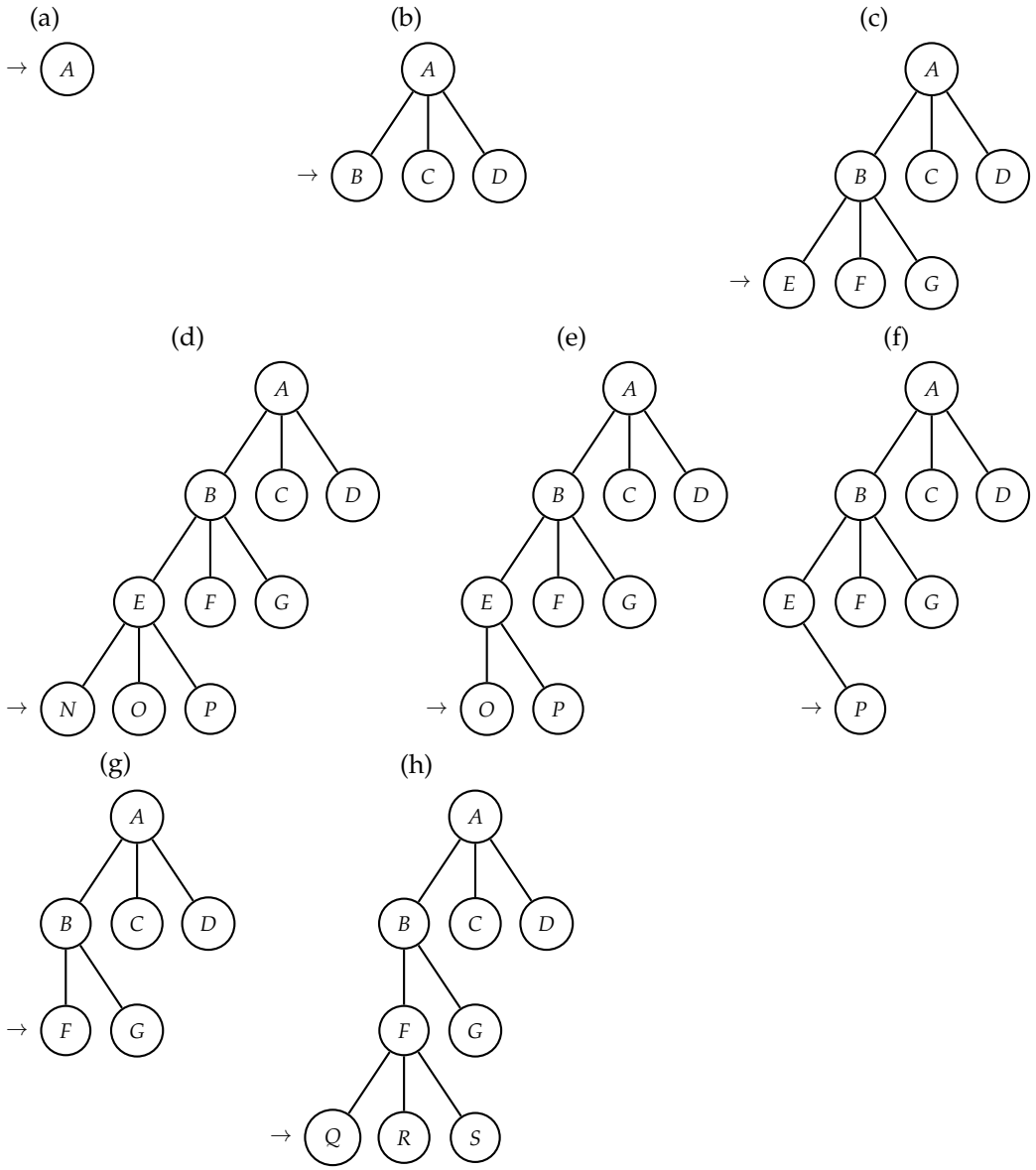
Bij graafgebaseerd breedte eerst zoeken kan men veel tijd winnen (t.o.v. boomgebaseerd zoeken) wanneer veel toestanden meerdere malen voorkomen in de zoekboom. Het extra geheugen dat men moet spenderen aan het bijhouden van de gesloten lijst weegt niet op tegen te tijdswinst die men kan maken. Om deze reden wordt breedte eerst zoeken meestal uitgevoerd in zijn graafgebaseerde versie.

2.3.2 Diepte Eerst Zoeken

Diepte eerst zoeken is in zekere zin duaal aan breedte eerst zoeken: hier gebruikt men een LIFO structuur voor het bijhouden van de open lijst. Deze stapel zorgt ervoor dat men zo snel mogelijk zo diep mogelijk in de boom afdaalt. In Figuur 2.7 ziet men een illustratie van diepte eerst zoeken op een ternaire boom.

Diepte eerst zoeken genereert steeds een linkerdeel van de boom. Wanneer m eindig is en de enige doeltop helemaal rechts onderaan in de boom zit dan worden alle toppen van de boom gegenereerd. In het slechtste geval is de tijdscomplexiteit m.a.w. $\mathcal{O}(b^m)$. Dit is dezelfde exponentiële (en dus slechte) tijdscomplexiteit als bij breedte eerst.

[4]De verwijzingen naar de voorgangers zorgen ervoor dat in principe de volledige zoekboom in het geheugen aanwezig blijft. Zie echter opmerking 2.12.



Figuur 2.7: Illustratie van diepte eerst zoeken op een ternaire boom. De toppen op diepte drie (i.e. N, O en P) hebben geen opvolgers. De doeltoestand is Q. De top die als volgende gekozen wordt voor expansie wordt aangeduid met een pijltje. De enige blaadjes van de boom (elementen van de open lijst) zijn de top gekozen voor expansie, zijn broers en voorouders en hun broers. Dit aantal is *lineair* in b en m.

Diepte eerst kan, zelfs wanneer er een oplossing is, in bepaalde gevallen toch in een oneindige lus geraken. Diepte eerst is m.a.w. niet compleet en bijgevolg ook niet optimaal. Zelfs wanneer diepte eerste een oplossing vindt is deze niet gegarandeerd de optimale oplossing. De oplossing ontdekt door diepte eerst is immers steeds de "meeste linkse" doeltop.

Waar diepte eerst wel goed op scoort is op het vlak van benodigd geheugen. Wanneer een bepaalde top wordt geëxpandeerd dan behoren enkel de broers van zijn voorouders tot de open lijst. Aangezien er maximaal m niveaus zijn en er hoogstens b broers zijn is de ruimtecomplexiteit van diepte eerste van de orde $\mathcal{O}(b \cdot m)$. Dit is een *lineaire* functie van b. Dit is dus een heel stuk beter dan de exponentiële tijdscomplexiteit in het geval van breedte eerst.

Normaalgesproken is het niet zeer zinvol om diepte eerst uit te voeren in de graafgebaseerde vorm. Men verliest immers de goede eigenschappen betreffende de ruimtecomplexiteit. Het kan wel zinvol zijn om diepte eerste aan te passen zodanig dat er geen actiesequenties worden geprobeerd die terugkeren naar een toestand die reeds op het huidige pad ligt. Op die manier kan eenvoudig vermeden worden dat diepte eerst terechtkomt in een oneindige lus. Men kan er echter niet mee vermijden dat diepte eerst oneindig lange actiesequenties probeert in toestandsruimten met een oneindig aantal toestanden.

2.3.3 Iteratief verdiepen

Iteratief verdiepen is geen directe toepassing van het boomgebaseerd zoekalgoritme dat gegeven werd in Algoritme 2.1. Het is in essentie een lus rond diepte-eerst zoeken waarbij het zoekproces wordt afgebroken wanneer een bepaalde diepte wordt bereikt; dit zoekproces wordt DIEPTE-GELIMITEERD ZOEKEN genoemd. Algoritme 2.3 geeft een implementatie voor diepte-gelimiteerd zoeken. Het algoritme wordt recursief geïmplementeerd en bij elke recursieve oproep wordt de maximale toegelaten diepte met één verminderd. Wanneer de toegelaten diepte de waarde nul bereikt dan wordt het meegegeven plan niet verder geëxpandeerd (en gebeuren er dus ook geen recursieve oproepen meer).

Het algoritme heeft een bijzondere returnwaarde nl. "hit boundary" om aan te geven dat er geen oplossing werd gevonden binnen de opgegeven dieptelimiet maar dat tijdens het zoekproces de dieptelimiet minstens éénmaal

werd bereikt. Deze returnwaarde geeft m.a.w. aan dat een oplossing *eventueel* kan gevonden worden wanneer de maximale toegelaten diepte wordt verhoogd.

Bij elke iteratie van ITERATIEF VERDIEPEN, zie Algoritme 2.4, wordt de maximaal toegelaten diepte met één verhoogd. Het algoritme stopt de eerste maal dat een oplossing wordt gevonden of wanneer het duidelijk is dat er geen oplossing is. Op die manier vermijden we het probleem van diepte eerst dat we terechtkomen in een oneindige lus. We vinden op deze manier immers steeds de meest ondiepe doeltop. Tegelijkertijd behouden we de goede eigenschappen m.b.t. de de ruimtecomplexiteit van diepte eerst. Figuur 2.8 illustreert iteratief verdiepen op een binaire boom.

Op het eerste zicht zou men denken dat dit proces een gigantische hoeveelheid werk teveel doet. De eerste lagen van de zoekboom worden immers meerdere malen opgebouwd. De eerste lagen van de zoekboom bevatten echter relatief weinig toppen tegenover de diepere lagen zodat de hoeveelheid werk die "teveel" wordt verricht relatief beperkt blijft. Een berekening maakt dit duidelijk. Veronderstel dat de meest ondiepe oplossing zich bevindt op diepte d. We bekijken nu hoeveel toppen er gegenereerd worden bij elke oproep van diepte-gelimiteerd zoeken, waarna we al deze gegenereerde toppen optellen.

diepte	aantal gegenereerde knopen
0	1
1	$1 + b$
2	$1 + b + b^2$
$\vdots$	$\vdots$
d	$1 + b + b^2 + \cdots + b^d$

Als we nu de som uitvoeren dan zien we dat de term 1 in totaal $d + 1$ keer voorkomt, de term b komt d keer voor, de term b^2 komt $d - 1$ keer voor enzovoort. Het totaal aantal gegenereerde toppen is dus

$$(d + 1) + db + (d - 1)b^2 + (d - 2)b^3 + \cdots + b^d = \mathcal{O}(b^d).$$

Om onszelf te overtuigen dat de hoeveelheid "extra" werk beperkt blijft is het interessant om het aantal toppen gegenereerd in de laatste iteratie te vergelijken met het totaal aantal toppen gegenereerd in alle iteraties ervoor.

Algoritme 2.3 Diepte-gelimiteerd zoeken

Invoer Een zoekprobleem P, een maximale diepte d.

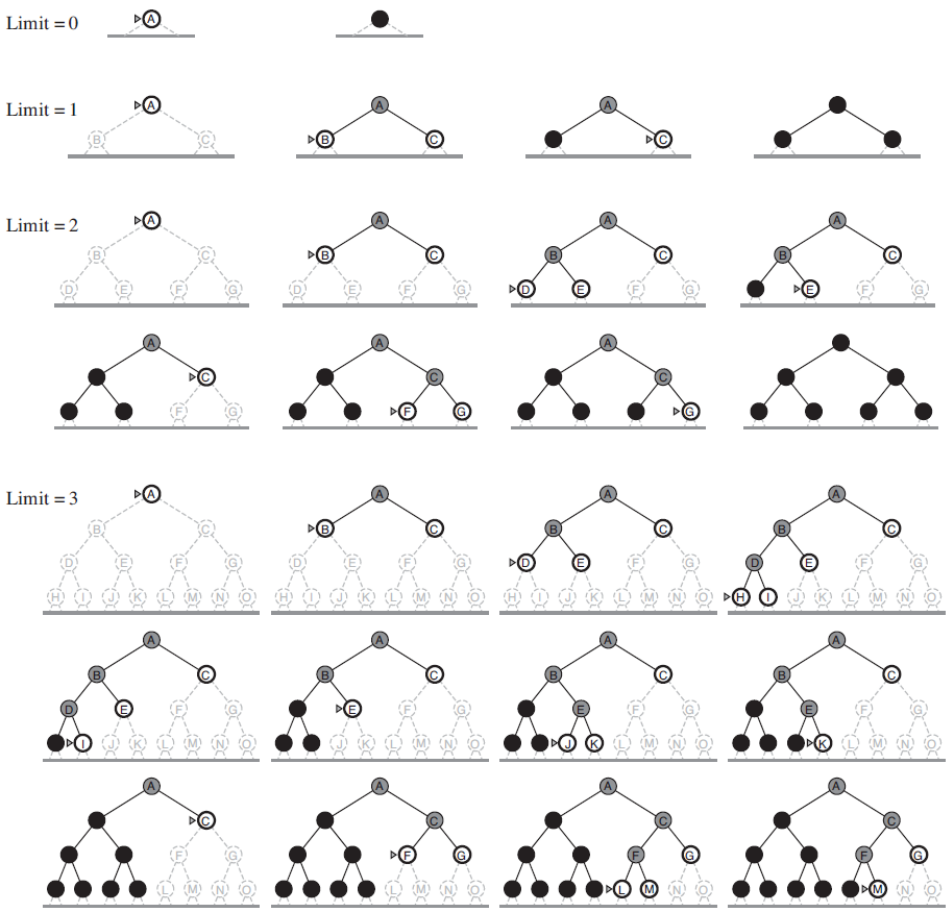
Uitvoer Een sequentie van acties wanneer een oplossing werd gevonden met diepte d of minder; een "hit boundary" conditie wanneer tijdens het zoekproces de maximale diepte werd bereikt of "error" wanneer er geen oplossing werd gevonden.

```
1:  function DepthLimitedSearch(P, d)
2:      c ← nieuw plan gebaseerd op initiële toestand P
3:      return DlsRecursive(c, P, d)
4:  end function
```

Invoer Een huidig plan c, een zoekprobleem P en een maximale diepte d.

Uitvoer Een sequentie van acties wanneer een oplossing werd gevonden met diepte d of minder startend vanaf het huidig plan; een "hit boundary" conditie wanneer tijdens het zoekproces de maximale diepte werd bereikt of "error" wanneer er geen oplossing werd gevonden.

```
5:  function DlsRecursive(c, P, d)
6:      if P.goalTest(c.getState) = true then
7:          return getActionSeq(c)                    ▷ Oplossing gevonden
8:      end if
9:      if d = 0 then
10:         return "hit boundary"                     ▷ Grens bereikt
11:     end if
12:     boundaryHit ← false   ▷ Grens bereikt in één van de rec. oproepen?
13:     for (s, a) ∈ c.getState.getSuccessors do
14:         child ← nieuw plan gebaseerd op (s, a) en c
15:         sol ← DlsRecursive(child, P, d − 1)       ▷ Recursieve oproep
16:         if sol = "hit boundary" then
17:             boundaryHit ← true
18:         else
19:             if sol ≠ "error: geen oplossing gevonden" then
20:                 return sol                        ▷ Effectieve oplossing gevonden
21:             end if
22:         end if
23:     end for
24:     if boundaryHit = true then
25:         return "hit boundary"
26:     else
27:         return "error: geen oplossing gevonden"
28:     end if
29: end function
```



Figuur 2.8: Illustratie van 4 iteraties van iteratief verdiepen op een binaire boom. De doeltoestand is M. Het aantal gegenereerde toppen in de eerste drie iteraties samen is $1 + 3 + 7 = 11$. Het aantal gegenereerde toppen in de vierde iteratie is 13. Bron: (Russell and Norvig, 2014).

Algoritme 2.4 Iteratief verdiepen

Invoer Een zoekprobleem P

Uitvoer Een sequentie van acties wanneer een oplossing werd gevonden of "error" wanneer er geen oplossing werd gevonden.

```
1: function ITERATIVEDEEPENING(P)
2:     d ← 0
3:     sol ← DEPTHLIMITEDSEARCH(P, d)
4:     while sol = "hit boundary" do
5:         d ← d + 1
6:         sol ← DEPTHLIMITEDSEARCH(P, d)
7:     end while
8:     return sol                          ▷ Oplossing of error
9: end function
```

Het aantal toppen gegenereerd in de laatste iteratie (met maximale toegelaten diepte d) is

$$N(d) = 1 + b + b^2 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}.$$

In het algemeen is het aantal toppen gegenereerd in de iteratie met maximale toegelaten diepte i gelijk aan

$$N(i) = \frac{b^{i+1} - 1}{b - 1}.$$

Het totaal aantal genereerde toppen in alle iteraties *behalve de laatste* is dus:

$$\begin{aligned}
\sum_{i=0}^{d-1} \frac{b^{i+1} - 1}{b - 1} &= \frac{1}{b-1}\left((\sum_{i=0}^{d-1} b^{i+1}) - d \right) \\
&= \frac{1}{b-1}\left((1 + b + b^2 + \cdots + b^d - 1) - d \right) \\
&= \frac{1}{b-1}\left(\frac{b^{d+1} - 1}{b-1} - 1 - d \right) \\
&= \frac{1}{b-1}(N(d) - 1 - d) \\
&\approx \frac{1}{b-1} N(d).
\end{aligned}$$

Wanneer de vertakkingsfactor b bv. gelijk is aan 4, dan hebben we dus ongeveer 33% van het werk "teveel" gedaan.

Voorbeeld 2.14 Veronderstel dat de vertakkingsfactor b gelijk is aan 4 en dat de meest ondiepe oplossing zich bevindt op diepte 10. Het aantal toppen gegenereerd bij de laatste iteratie (met dieptelimiet 10) is gelijk aan:

$$\frac{4^{11}-1}{4-1} = 1\,398\,101.$$

Het aantal toppen gegenereerd in alle voorgaande iteraties is

$$\frac{1}{3}(1\,398\,101 - 1 - 10) = 466\,030.$$

∎

Iteratief verdiepen is een compleet zoekalgoritme en zal steeds de oplossing met het minste aantal acties vinden. Het algoritme is in het algemeen niet optimaal maar wel in het bijzondere geval dat alle acties dezelfde kost hebben. Het algoritme heeft een exponentiële tijdscomplexiteit maar slechts een lineaire ruimtecomplexiteit.

2.3.4 Uniforme Kost Zoeken

Uniforme kost zoeken (Eng. *uniform cost search*) tracht het probleem dat breedte eerst niet noodzakelijk optimaal is wanneer acties een verschillende kost hebben op te lossen door steeds het plan te expanderen waarvoor de totale kost van dit plan minimaal is. De open lijst wordt hier m.a.w. geïmplementeerd aan de hand van een prioriteitswachtrij. Een kleinere kost betekent een grotere prioriteit.

Het idee achter uniforme kost zoeken is dus in essentie gelijk aan het algoritme van Dijkstra.

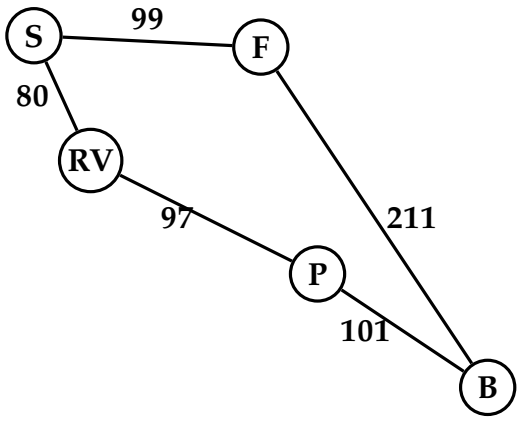
Uniforme kost zoeken is, wanneer alle acties een kost hebben die groter of gelijk is aan één of andere positieve ϵ, een compleet en optimaal algoritme. De tijd- en ruimtecomplexiteit is $\mathcal{O}(b^{1+\lfloor C^*/\epsilon \rfloor})$, waarbij C^* de kost van de optimale oplossing voorstelt.

Voorbeeld 2.15 (Belang van plaats doeltest) Veronderstel dat men op het deel van de kaart van Roemenië zoals getoond in Figuur 2.9 van S(ibiu) naar B(ucharest) wenst te reizen. We voeren het graafgebaseerde uniforme kost zoeken algoritme uit. Om plaats te winnen korten we de toestanden af tot hun eerste letters. Het verloop van het algoritme zie je in Tabel 2.1.

Bij dit voorbeeld zie je dat de *plaats* van de doeltest van cruciaal belang is. Na het expanderen van F(agaras) verschijnt het doel, nl. B(ucharest),

open lijst $\{(\text{pad}, g)\}$	gekozen plan	geëxpandeerde toestanden
$\{(S, 0)\}$	$(S, 0)$	$\emptyset$
$\{(F \to S, 99), (RV \to S, 80)\}$	$(RV \to S, 80)$	$\{S\}$
$\{(F \to S, 99),$ $(P \to RV \to S, 177),$ $(S \to RV \to S, 160)\}$	$(F \to S, 99)$	$\{S, RV\}$
$\{(P \to RV \to S, 177),$ $(S \to RV \to S, 160),$ $(B \to F \to S, 310),$ $(S \to F \to S, 198)\}$	$(S \to RV \to S, 160)$ niet geëxpandeerd	$\{S, RV, F\}$
$\{(P \to RV \to S, 177),$ $(B \to F \to S, 310),$ $(S \to F \to S, 198)\}$	$(P \to RV \to S, 177)$	$\{S, RV, F\}$
$\{(RV \to P \to RV \to S, 274),$ $(B \to P \to RV \to S, 278)$ $(B \to F \to S, 310),$ $(S \to F \to S, 198)\}$	$(S \to F \to S, 198)$ niet geëxpandeerd	$\{S, RV, F, P\}$
$\{(RV \to P \to RV \to S, 274),$ $(B \to P \to RV \to S, 278)$ $(B \to F \to S, 310)\}$	$(RV \to P \to RV \to S, 274)$ niet geëxpandeerd	$\{S, RV, F, P\}$
$\{(B \to F \to S, 310)\}$	$(B \to P \to RV \to S, 278)$ doel bereikt	$\{S, RV, F, P\}$

Tabel 2.1: Verloop van uniforme kost zoeken om van Sibiu naar Bucharest te gaan volgens de kaart in Figuur 2.9.



Figuur 2.9: Deel van de kaart van Roemenië. De kost om van de ene stad naar de andere te gaan staat bij de bogen. Figuur gebaseerd op: (Russell and Norvig, 2014).

reeds op de open lijst (met een actiesequentie die *niet* de optimale is). Het algoritme stopt echter niet op dit moment en verklaart slechts succes eens de doeltest slaagt voor het plan dat gekozen werd voor expansie. Enkel dan is het gegarandeerd dat het algoritme steeds een optimale oplossing vindt.∎

2.4 Geïnformeerde Zoekmethoden

De blinde of niet geïnformeerde zoekmethoden hebben geen toegang tot domeinkennis om het zoeken meer efficiënt te laten verlopen. Daardoor gaan ze vaak toestanden expanderen die wij (als mens) "stom" vinden. Probeer bv. maar eens het uniforme kost algoritme toe te passen wanneer je van Sibiu naar Bucharest wenst te gaan, gebruikmakend van de volledige kaart van Roemenië in Figuur 2.11!

Wanneer wij bv. een weg moeten gaan plannen tussen twee steden in een gebied waar de weg niet kennen dan gaan wij toch altijd te neiging hebben om eerst te gaan kijken naar steden die al "in de juiste richting" liggen, i.e. steden waarvoor de afstand in vogelvlucht tot de doelstad klein is. We beschouwen de afstand in vogelvlucht als een goede indicator of schatting voor de werkelijke afstand[5]. We gebruiken een *heuristiek* om het zoekproces efficiënter te laten verlopen. We bekijken nu hoe dit idee geïmplementeerd

[5]Dit hoeft niet noodzakelijk zo te zijn, bv. wanneer er meren, rivieren of bergketens aanwezig zijn.

kan worden a.d.h.v. twee zoekalgoritmes. Eerst bespreken we echter nog enkele eigenschappen van heuristieken.

2.4.1 Heuristieken

Definitie 2.16 Een HEURISTIEK h is een afbeelding van de verzameling toestanden S naar de verzameling van niet-negatieve reële getallen $\mathbb{R}^+$, i.e.

$$h\colon S \to \mathbb{R}^+ \colon s \mapsto h(s).$$ ∎

Opmerking 2.17 Het is uiteraard de bedoeling dat de heuristiek een goede schatting is voor de werkelijke kost naar het doel, i.e. wanneer $h(s)$ klein is dan is dit, hopelijk, een goede indicator dat s ook effectief niet ver van een doeltoestand verwijderd is, en omgekeerd wanneer $h(s)$ groot is dan moet dit erop wijzen dat s ver van het doel verwijderd is.

Aangezien de heuristiek h zal gebruikt worden in de zoekalgoritmes is het ook noodzakelijk dat *h snel te berekenen* is. ∎

Voorbeeld 2.18 (Heuristieken voor 8-puzzel) Voor de 8-puzzel kan bv. gekeken worden naar het aantal vakjes (genummerd van 1 t.e.m. 8) dat niet op zijn juiste plaats staat in vergelijking met doel

$$h_1 = \text{aantal niet-lege vakjes (tegeltjes) dat niet op zijn juiste plaats staat} \tag{2.2}$$

Deze heuristiek volgt de redenering dat hoe meer vakjes verkeerd zijn, hoe meer zetten nog nodig zijn om de puzzel volledig correct te maken.

We kunnen echter niet enkel rekening houden met het aantal verkeerde vakjes maar ook met de afstand (uitgedrukt m.b.v. de Manhattan afstand) tot hun doel:

$$h_2 = \sum_{i=1}^{8} (\text{Manhattan afstand van vakje } i \text{ tot zijn correcte plaats}). \tag{2.3}$$

Voor de linkerpuzzel in Figuur 2.1 neemt h_1 de waarde 8 aan, want alle niet-lege vakjes staan op de verkeerde plaats. De heuristiek h_2 neemt hier de waarde

$$3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

aan. Inderdaad, vakje 1 moet minstens 3 maal verschoven worden, vakje 2 minstens 1 maal, enzovoort. ∎

De heuristieken h_1 en h_2 zijn bijzonder in die zin dat ze de werkelijke afstand *nooit overschatten*. Zo'n heuristieken noemen we toelaatbaar.

Definitie 2.19 Een heuristiek $h\colon S \to \mathbb{R}^+$ is TOELAATBAAR als voor elke toestand s geldt dat $h(s) \leq C^*(s)$ waarbij C^* de kost van een optimale oplossing voorstelt van s naar een doeltoestand. ∎

Stelling 2.20 Wanneer de heuristiek h toelaatbaar is, dan is $h(g) = 0$ voor elke doeltoestand g. ∎

Bewijs Voor een doeltoestand g geldt dat de kost van de optimale oplossing gelijk is aan nul, zodat

$$0 \leq h(g) \leq C^*(g) = 0,$$

waaruit onmiddellijk volgt dat $h(g) = 0$. ♢

Definitie 2.21 Een heuristiek $h\colon S \to \mathbb{R}^+$ is CONSISTENT als voor elke doeltoestand g geldt dat $h(g) = 0$ en als bovendien voor elke toestand s en elke actie a op s met $s' = T(s, a)$ geldt dat

$$h(s) \leq c(s, a, s') + h(s').$$ ∎

Opmerking 2.22 Wanneer we de heuristiek beschouwen als "in vogelvlucht op doel afgaan" dan is een heuristiek consistent wanneer het steeds korter is om in vogelvlucht op doel af te gaan ($h(s)$) dan om eerst een actie te nemen a.d.h.v. het transitiemodel ($c(s, a, s')$) en dan in vogelvlucht op doel af te gaan ($h(s')$). ∎

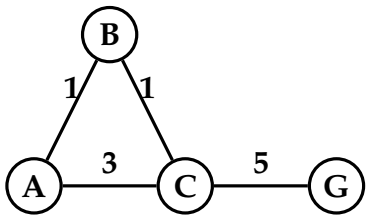
Stelling 2.23 Als een heuristiek consistent is, dan is ze ook onmiddellijk toelaatbaar. ∎

Bewijs We kunnen voor elke actie a die een toestand s omzet in s' de *heuristische kost* definiëren als $h(s) - h(s')$. Voor een consistente heuristiek geldt dat

$$h(s) - h(s') \leq c(s, a, s'). \tag{2.4}$$

Voor een consistente heuristiek is de heuristische kost m.a.w. steeds kleiner dan de werkelijke kost.

Beschouw nu een willekeurige toestand s waarvoor de kost van een optimale oplossing gegeven wordt door $C^*(s)$. We moeten aantonen dat



s	G	A	B	C
$h(s)$	0	4	6	1

Figuur 2.10: Voorbeeld van een eenvoudige toestandsruimte met een toelaatbare maar niet-consistente heuristiek.

$h(s) \leq C^*(s)$. Neem aan dat een optimale oplossing bestaat uit de volgende opeenvolging van acties (en bijhorende toestanden):

$$s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \longrightarrow \cdots \longrightarrow s_{n-2} \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n = g.$$

Nu geldt uiteraard dat

$$\begin{aligned} C^*(s) &= c(s_0, a_1, s_1) + c(s_1, a_2, s_2) + \cdots + c(s_{n-2}, a_{n-1}, s_{n-1}) \\ &\quad + c(s_{n-1}, a_n, s_n) \\ &\geq \big(h(s_0) - h(s_1)\big) + \big(h(s_1) - h(s_2)\big) + \cdots + \big(h(s_{n-2}) - h(s_{n-1})\big) \\ &\quad + \big(h(s_{n-1}) - h(s_n)\big) \\ &= h(s_0) - h(s_n) \\ &= h(s) - h(g) \\ &= h(s). \end{aligned}$$

Hierbij hebben we in de eerste stap gebruikgemaakt van het feit dat, voor een consistente heuristiek, de heuristische kost van een actie hoogstens gelijk is aan de werkelijke kost, zie vgl. (2.4). In de laatste stap maakten we gebruik van het feit dat, bij definitie, $h(g) = 0$ voor elke consistente heuristiek.

Dit toont aan dat elke consistente heuristiek ook toelaatbaar is. ♢

Opmerking 2.24 (Toelaatbare maar inconsistente heuristiek) Het is *niet* zo dat elke toelaatbare heuristiek ook consistent is. Beschouw de toestandsruimte in Figuur 2.10, waarbij G de doeltoestand voorstelt. We zien op het zicht dat de kost van de optimale oplossing voor de verschillende toestanden gegeven wordt door

s	G	A	B	C
$C^*(s)$	0	7	6	5

Wanneer we dit vergelijken met de heuristiek h uit Figuur 2.10, dan zien we dat voor elke toestand s geldt dat

$$h(s) \leq C^*(s).$$

De heuristiek h is m.a.w. toelaatbaar.

Opdat de heuristiek h consistent zou zijn moet voor elke toestand s en voor elke opvolger s' gelden dat

$$h(s) \leq c(s, a, s') + h(s'). \tag{2.5}$$

We controleren dit nu systematisch voor de acht verschillende mogelijkheden

s	s'	$h(s)$	$c(s,a,s') + h(s')$	vgl. (2.5) voldaan?
A	B	4	$1+6=7$	ja
A	C	4	$3+1=4$	ja
B	A	6	$1+4=5$	nee
B	C	6	$1+1=2$	nee
C	A	1	$3+4=7$	ja
C	B	1	$1+6=7$	ja
C	G	1	$5+0=5$	ja
G	C	0	$5+1=6$	ja

We zien dat in twee gevallen de ongelijkheid (2.5) niet voldaan is, zo is bv. voor $s = B$ en $s' = A$ de heuristische kost $6 - 4 = 2$, en dit is groter dan de werkelijke kost. Een analoge redenering geldt voor $s = B$ en $s' = C$.∎

2.4.2 Gulzig Beste Eerst

De gulzig beste eerst zoekmethode maakt gebruik van een heuristiek h. De methode kiest steeds de top met de kleinste waarde van h als de volgende top die wordt geëxpandeerd. De open lijst wordt hier dus, net als bij uniforme kost zoeken, geïmplementeerd als een prioriteitswachtrij en een kleinere waarde voor h betekent een grotere prioriteit.

Voorbeeld 2.25 (Niet-optimaliteit gulzig beste eerst) In dit voorbeeld passen we het gulzig beste eerst algoritme toe op de toestandsruimtegraaf in Figuur 2.10 waarbij we de volgende consistente heuristiek gebruiken:

s	G	A	B	C
$h(s)$	0	7	6	5

De gebruikte heuristiek is in zekere zin "perfect" omdat ze voor elke toestand s gelijk is aan $C^*(s)$. De initiële toestand is A en de doeltoestand is G.

open lijst $\{(\text{pad}, f = h)\}$	gekozen top
$\{(A, 7)\}$	$(A, 7)$
$\{(B \to A, 6), (C \to A, 5)\}$	$(C \to A, 5)$
$\{(B \to A, 6), (A \to C \to A, 7)$ $(G \to C \to A, 0)\}$	$(G \to C \to A, 0)$

Het algoritme gaat recht op doel af. In dit geval behoort elke geëxpandeerde toestand ook effectief tot de geretourneerde oplossing. Helaas is de gevonden oplossing niet de optimale oplossing. In dit geval komt dit omdat het algoritme geen rekening houdt met de (hoge) kost om vanuit A de toestand C te bereiken. Het algoritme houdt enkel rekening met het feit dat C er "beter" uitziet dan B op basis van hun heuristische waarden. ∎

Voorbeeld 2.26 (Incompleetheid gulzig beste eerst) In dit voorbeeld passen we het gulzig beste eerst algoritme toe op de toestandsruimtegraaf in Figuur 2.10 waarbij we de volgende consistente heuristiek gebruiken:

s	G	A	B	C
$h(s)$	0	3	4	5

De initiële toestand is A en de doeltoestand is G. Bij de start bestaat de open lijst enkel uit de toestand A met bijhorende heuristische waarde 3. We expanderen A en voegen B en C toe aan de open lijst. Aangezien B een lagere heuristische waarde heeft dan C wordt B als eerste geëxpandeerd. De opvolgers A en C worden aan de open lijst toegevoegd. De open lijst bevat op dit moment dus drie plannen: $C \to A$, $C \to B \to A$ en $A \to B \to A$. Het

laatste plan (voor toestand A) heeft de laagste heuristische waarde ($h = 3$) en bijgevolg wordt dit als volgende plan geëxpandeerd: het algoritme zit vast in een oneindige lus.

Zelfs met een consistente heuristiek is het boomgebaseerde gulzig beste eerst algoritme niet compleet. ∎

Opmerking 2.27 Omdat we in Voorbeeld 2.26 werken in een eindige toestandsruimte zou de graafgebaseerde versie van gulzig beste eerst *wel* een oplossing vinden. Of het gulzig beste eerst algoritme hier eindigt met de optimale oplossing hangt af van de manier waarop gekozen wordt tussen de plannen

$$(C \to A, h = 5) \quad \text{en} \quad (C \to B \to A, h = 5).$$ ∎

We hebben reeds gezien dat het gulzig beste eerst algoritme niet compleet en niet optimaal is. Ook de tijd- en ruimte complexiteit zijn in het slechtste geval van de orde $\mathcal{O}(b^m)$. Deze exponentiële tijdscomplexiteit kan door het gebruik van een goede heuristiek echter sterk teruggedrongen worden.

2.4.3 A* Zoekalgoritme

Het probleem van de gulzig beste eerst zoekmethode is dat er *enkel* rekening wordt gehouden met de waarde van de heuristiek en niet met de kost van de reeds afgelegde weg. Er wordt m.a.w. waardevolle informatie genegeerd.

Bij de A* zoekmethode wordt de open lijst nog steeds geïmplementeerd als een prioriteitswachtrij maar de volgende top die wordt geëxpandeerd is de top (plan) n waarvoor

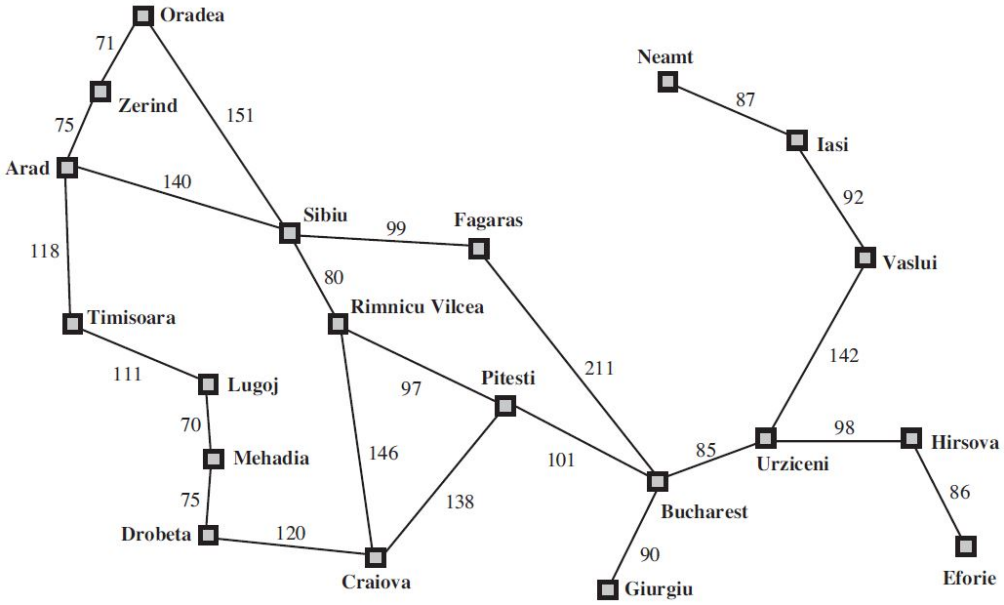
$$f(n) = g(n) + h(n)$$

minimaal is. Hierbij is $g(n)$ de totale kost van het plan n en is $h(n)$ (met misbruik van notatie) de waarde van de heuristiek voor de toestand die hoort bij deze top.

Voorbeeld 2.28 (A* met een toelaatbare heuristiek) We proberen op de kaart van Roemenië, zie Figuur 2.11, een weg te vinden van Arad naar Bucharest. We gebruiken het A* algoritme met als heuristiek de afstand in vogelvlucht (tot Bucharest), zoals gegeven in Tabel 2.2. De opbouw van de zoekboom kan je volgen in Figuur 2.12.

Stad	Afstand	Stad	Afstand
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

Tabel 2.2: Afstanden in vogelvlucht tot Bucharest. Gebaseerd op (Russell and Norvig, 2014).



Figuur 2.11: Een vereenvoudige kaart van Roemenië. Bron: (Russell and Norvig, 2014).

We starten in Arad en initieel bestaat de open lijst enkel uit het plan A. De f-waarde die bij dit plan hoort is $f = 0 + 366$, want de afgelegde afstand om van Arad in Arad te geraken is 0 en de geschatte afstand naar Bucharest (in vogelvlucht) is 366.

Dit enige plan wordt van de open lijst gehaald. Aangezien Arad niet voldoet aan de doeltest wordt dit plan geëxpandeerd. De plannen $S \to A$, $T \to A$ en $Z \to A$ worden aan de open lijst toegevoegd:

$$(S \to A, 140 + 253 = 393), (T \to A, 118 + 329 = 447),$$
$$(Z \to A, 75 + 374 = 449).$$

Aangezien het plan $(S \to A, 140 + 253 = 393)$ de kleinste f-waarde heeft wordt dit als volgende van de open lijst gehaald. Het doel is nog niet bereikt en dus wordt dit plan geëxpandeerd. De volgende plannen worden toegevoegd aan de open lijst:

$$(A \to S \to A, 280 + 366 = 646), (F \to S \to A, 239 + 176 = 415),$$
$$(O \to S \to A, 291 + 380 = 671), (RV \to S \to A, 220 + 193 = 413).$$

De open lijst bestaat nu uit 6 plannen waarvan het plan $RV \to S \to A$ de kleinste f-waarde heeft. Dit plan wordt van de open lijst gehaald, en aangezien het doel nog niet werd bereikt wordt het geëxpandeerd en worden de volgende plannen toegevoegd aan de open lijst:

$$(C \to RV \to S \to A, 366 + 160 = 526), (P \to RV \to S \to A, 317 + 100 = 417)$$
$$(S \to RV \to S \to A, 300 + 253 = 553).$$

Van de 8 plannen op de open lijst heeft het plan $F \to S \to A$ de laagste f-waarde, nl. 415. Dit plan wordt dus als volgende geëxpandeerd aangezien het doel nog niet werd bereikt. De plannen

$$(S \to F \to S \to A, 338 + 253 = 591), (B \to F \to S \to A, 450 + 0)$$

worden toegevoegd aan de open lijst. Merk op dat de open lijst op dit moment reeds een plan bevat dat aan de doeltest voldoet. Het algoritme stopt echter nog niet. Aangezien het plan $P \to RV \to S \to A$ een f-waarde heeft van 417 is het eventueel nog mogelijk om hierlangs een betere oplossing te vinden. We expanderen dit plan en voegen de volgende plannen toe aan de open lijst:

$$(B \to P \to RV \to S \to A, 418 + 0 = 418),$$
$$(C \to P \to RV \to S \to A, 455 + 160 = 615),$$
$$(RV \to P \to RV \to S \to A, 414 + 193 = 607).$$

Het plan $B \to P \to RV \to S \to A$ is het plan met de laagste f-waarde. Dit plan wordt van de open lijst gehaald. De doeltest slaagt voor dit plan (we hebben Bucharest bereikt) en het algoritme eindigt en geeft de volgende oplossing terug:

$$A \to S \to RV \to P \to B.$$

Dit is de optimale oplossing. ∎

Voorbeeld 2.29 (A* met een niet toelaatbare heuristiek) Beschouw de toestandsruimtegraaf in Figuur 2.10 maar gebruik de volgende heuristiek

s	G	A	B	C
$h(s)$	0	4	8	1

Deze heuristiek is niet toelaatbaar omdat de waarde voor $h(B)$ strikt groter is dan de kost van de optimale oplossing van B naar de doeltoestand G.

We voeren nu A* uit startend in de toestand A.

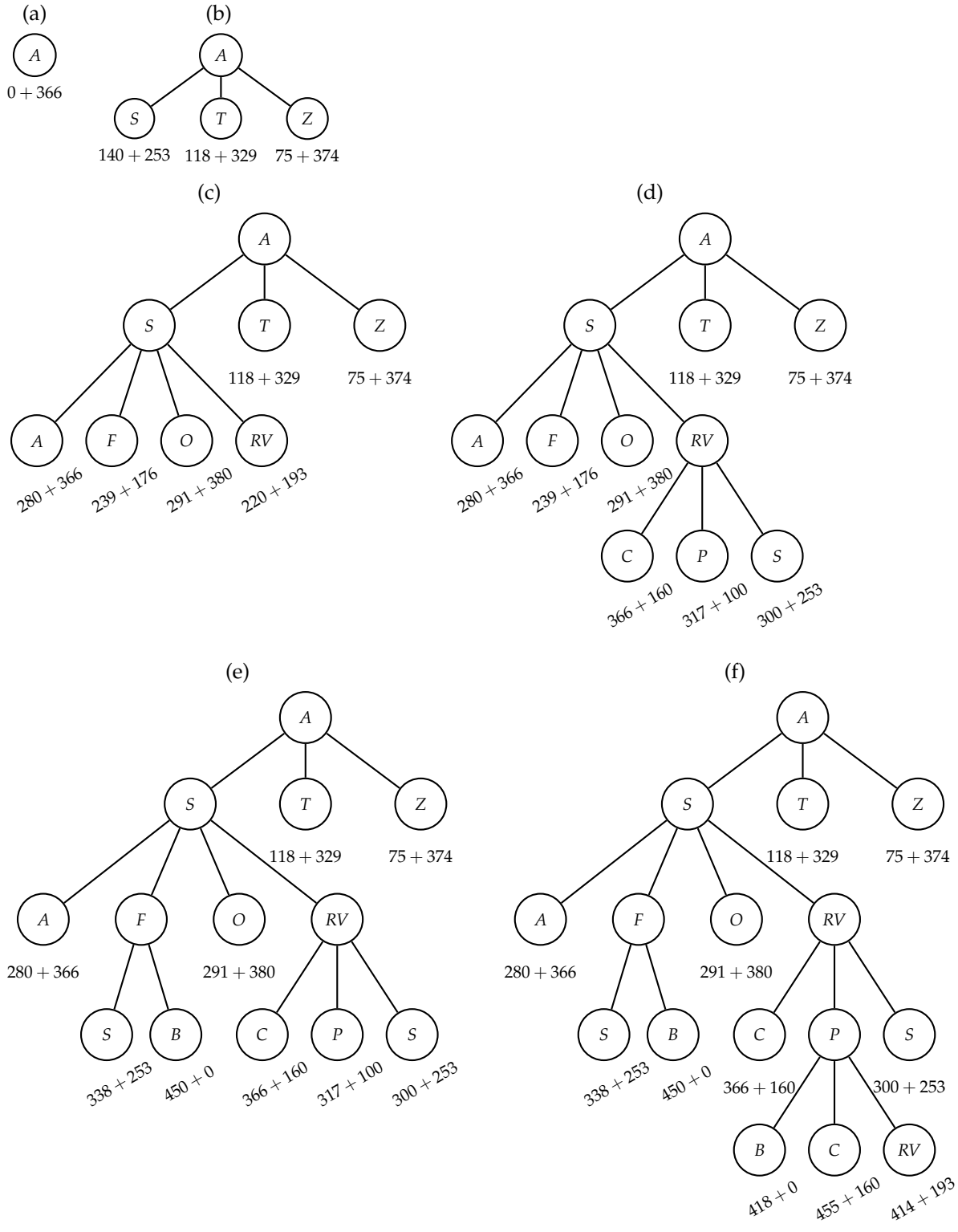
open lijst $\{(\text{pad}, f)\}$	gekozen knoop
$\{(A, 0+4)\}$	$(A, 0+4)$
$\{(B \to A, 1+8), (C \to A, 3+1)\}$	$(C \to A, 3+1)$
$\{(B \to A, 1+8), (A \to C \to A, 6+4)$ $(B \to C \to A, 4+8), (G \to C \to A, 8+0)\}$	$(G \to C \to A, 8+0)$

Het algoritme eindigt dus met de oplossing $A \to C \to G$. Deze oplossing is niet optimaal want de oplossing $A \to B \to C \to G$ heeft een kleinere kost.

Dit voorbeeld toont aan dat A* niet noodzakelijk optimaal is wanneer een niet-toelaatbare heuristiek wordt gebruikt. ∎

De volgende stelling toont aan dat het gebruik van een toelaatbare heuristiek *voldoende* is om optimaliteit te garanderen wanneer men het A* algoritme gebruikt in zijn boomgebaseerde versie (op voorwaarde dat alle acties een strikt positieve kost hebben).

Stelling 2.30 Wanneer boomgebaseerde A* gebruikmaakt van een toelaatbare heuristiek h en wanneer alle acties een kost hebben groter of gelijk aan



Figuur 2.12: Progressie van A* bij het zoeken van Arad naar Bucharest in Roemenië.

een zekere strikt positieve ϵ, dan is A* compleet en optimaal, i.e. dan vindt het algoritme steeds een optimale oplossing wanneer die bestaat. ∎

Bewijs We tonen eerst aan dat de open lijst steeds een plan bevat dat uitgebreid kan worden tot een optimale oplossing; zo'n plan noemen we een "uitbreidbaar" plan. Dit is uiteraard zo bij de start van het algoritme aangezien de open lijst een plan bevat met een leeg pad. Bij elke iteratie van het algoritme zijn er twee mogelijkheden: ofwel werd een niet-uitbreidbaar plan gekozen voor expansie en staan de uitbreidbare plannen nog steeds op de open lijst. Ofwel werd een uitbreidbaar plan gekozen: in dit geval is minstens één van zijn opvolgers ook een "uitbreidbaar" plan.

Noem $C^*(s)$ de kost van de optimale oplossing vanaf de toestand s tot een doeltoestand. Voor zo'n uitbreidbaar plan p geldt wegens de toelaatbaarheid van de heuristiek h dat

$$\begin{aligned} f(p) &= g(p) + h(p) \\ &\leq g(p) + C^*(p) && \text{toelaatbaardheid } h \\ &= C^*(s_0). && s_0 \text{ is de starttoestand} \end{aligned}$$

Veronderstel dat op een bepaald moment in het algoritme een "gevaarlijke" situatie optreedt in die zin dat er een plan n op de open lijst verschijnt waarvoor de doeltest voldaan is maar waarvoor de kost van het pad suboptimaal is. We tonen nu aan dat dit plan *n nooit zal gekozen worden voor expansie*. Noem p een uitbreidbaar plan op de open lijst. Er geldt

$$\begin{aligned} f(n) &= g(n) + h(n) \\ &= g(n) && n \text{ is doeltoestand en } h \text{ is toelaatbaar} \\ &> C^*(s_0) && \text{want } n \text{ is suboptimaal} \\ &\geq f(p) && \text{zie voorgaande berekening} \end{aligned}$$

Dit toont inderdaad aan dat het plan n nooit zal gekozen worden voor expansie, want p moet zeker eerder worden gekozen. Het algoritme kan bijgevolg nooit een suboptimale oplossing teruggeven.

We tonen tenslotte aan dat A* steeds eindigt wanneer er een oplossing is. Inderdaad, omdat elke actie een kost heeft van minstens ϵ, kunnen er slechts een eindig aantal plannen zijn met kost kleiner of gelijk aan $f(p)$ met p het uitbreidbare plan met minimale $f(p)$ dat nu op de open lijst staat. Dit

open lijst $\{(\text{pad}, f)\}$	gekozen top	geëxpandeerde toestanden
$\{(A, 0+4)\}$	$(A, 0+4)$	$\emptyset$
$\{(B \to A, 1+6), (C \to A, 3+1)\}$	$(C \to A, 3+1)$	$\{A\}$
$\{(B \to A, 1+6), (A \to C \to A, 6+4)$ $(B \to C \to A, 4+6), (G \to C \to A, 8+0)\}$	$(B \to A, 1+6)$	$\{A, C\}$
$\{(A \to C \to A, 6+4), (B \to C \to A, 4+6)$ $(G \to C \to A, 8+0), (A \to B \to A, 2+4)$ $(C \to B \to A, 2+1)$	$(C \to B \to A, 2+1)$ niet geëxpandeerd	$\{A, C, B\}$
$\{(A \to C \to A, 6+4), (B \to C \to A, 4+6)$ $(G \to C \to A, 8+0), (A \to B \to A, 2+4)\}$	$(A \to B \to A, 2+4)$ niet geëxpandeerd	$\{A, C, B\}$
$\{(A \to C \to A, 6+4), (B \to C \to A, 4+6)$ $(G \to C \to A, 8+0)\}$	$(G \to C \to A, 8+0)$ doeltest geslaagd	$\{A, C, B\}$

Tabel 2.3: Uitvoering van het graafgebaseerde A*-algoritme met een toelaatbare maar inconsistente heuristiek.

plan wordt dus binnen een eindig aantal iteraties geëxpandeerd, waardoor er nu een uitbreidbaar plan op de open lijst staat dat één actie minder ver verwijderd is van de doeltoestand dan p. Uiteindelijk zal er dus een plan worden gekozen dat aan de doeltest voldoet. Zoals reeds aangetoond kan dit plan geen suboptimale oplossing naar het doel bevatten. ♢

Uit volgend voorbeeld blijkt dat het voor het A* algoritme niet voldoende is om te werken met een toelaatbare heuristiek wanneer men graafgebaseerd zoeken uitvoert.

Voorbeeld 2.31 (Toelaatbaarheid en graafgebaseerd zoeken) Bekijk de toestandsruimte en de heuristiek in Figuur 2.10. Zoals reeds gezien in Opmerking 2.24 is deze heuristiek toelaatbaar maar inconsistent.

We voeren A* uit in de graafgebaseerde versie. Het verloop van het algoritme kan je volgen in Tabel 2.3.

Zoals je ziet eindigt A* hier met een foutief antwoord: de sequentie $A \to C \to G$ is *geen* optimale oplossing. Het probleem is dat de sequentie van acties naar de toestand C die het eerst werd geëxpandeerd *niet* de optimale sequentie is. De betere sequentie die later werd ontdekt ($A \to B \to C$)

wordt niet meer in beschouwing genomen want C is reeds geëxpandeerd en is dus reeds een element van de gesloten lijst.

Dit kan niet gebeuren wanneer er gewerkt wordt met een consistente heuristiek. ∎

Stelling 2.32 Wanneer graafgebaseerde A* gebruikmaakt van een consistente heuristiek h en wanneer alle acties een kost hebben groter of gelijk aan een zekere strikt positieve ϵ, dan is A* compleet en optimaal, i.e. dan vindt het algoritme steeds een optimale oplossing wanneer die bestaat. ∎

Bewijs Zonder bewijs. ♢

A* is duidelijk een interessant algoritme: onder "milde" beperkingen van de gebruikte heuristiek is A* compleet en optimaal. De toppen die door A* worden geëxpandeerd zijn alle toppen n waarvoor

$$f(n) = g(n) + h(n) < C^*(s_0).$$

Er wordt geen enkele top geëxpandeerd waarvoor $f(n)$ strikt groter is dan $C^*(s_0)$. Het hangt van de implementatie van het algoritme of toppen met $f(n) = C^*(s_0)$ worden geëxpandeerd of niet. M.a.w. hoe beter (groter) de heuristiek hoe minder toppen worden geëxpandeerd[6].

A* is echter geen wonderalgoritme: de tijds- en ruimtecomplexiteit zijn in het slechtste geval nog steeds exponentieel, waarbij in de meeste gevallen A* sneller een tekort heeft aan geheugen dan aan tijd.

2.5 Ontwerpen van Heuristieken

Het is duidelijk dat een goede heuristiek een grote positieve invloed kan hebben op de tijds- en ruimtecomplexiteit van een algoritme zoals A*. Hoe kunnen we nu zo'n heuristieken construeren?

We bekijken twee manieren om dit aan te pakken.

2.5.1 Gebruik van Vereenvoudige Problemen

Voorbeeld 2.33 (Agent in Doolhof) Veronderstel dat de agent zich in een grid beweegt en dat de agent zich door middel van de acties `Boven`, `Onder`,

[6] Opletten: indien h "te groot" wordt is deze misschien niet meer toelaatbaar!

`Links` en `Rechts` naar een bepaalde locatie in het grid moet bewegen. Echter, sommige lokaties zijn muren zodat het grid een doolhof voorstelt. Als we nu de *muren wegdenken* dan is de oplossing van dit vereenvoudigde probleem (Eng. *relaxed problem*) onmiddellijk te berekenen. Het aantal benodigde acties is niets anders dan de Manhattan-afstand tussen de huidige locatie en de doellocatie; dit is meteen ook een aanvaardbare (en consistente) heuristiek voor het oorspronkelijke probleem. ∎

Voorbeeld 2.34 (De 8-puzzel) Wanneer we de regels voor de 8-puzzel wat formeler opschrijven dan kunnen we zeggen dat vakje A naar vakje B kan verplaatst worden als

1. de vakjes A en B aangrenzend zijn; **en**
2. het vakje B is het lege vakje.

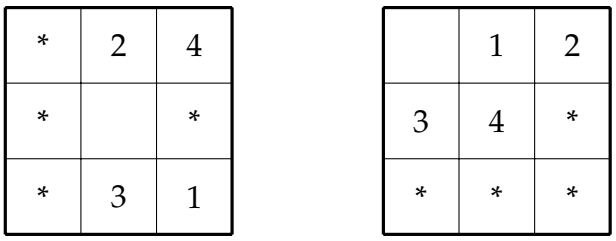
Door nu één of meerdere van deze condities (restricties) weg te laten bekomen we de volgende drie vereenvoudigde problemen.

1. Vakje A kan naar vakje B worden verplaatst als A en B aangrenzend zijn.
2. Vakje A kan naar vakje B worden verplaatst als B het lege vakje is.
3. Vakje A kan naar vakje B worden verplaatst (zonder voorwaarden).

Wanneer we het derde vereenvoudigde probleem bekijken dan zien we dat de optimale oplossing van dit probleem erin bestaat om elk vakje dat verkeerd staat te verplaatsen naar zijn juiste positie. Het aantal acties dat hiervoor nodig is precies het aantal vakjes dat niet op zijn juiste plaats staat; dit is niets anders de heuristiek h_1 uit vergelijking (2.2).

De optimale oplossing van het eerste vereenvoudigde probleem bestaat erin om elk vakje naar zijn correcte positie te schuiven (zonder er rekening mee te houden dat er andere vakjes “in de weg” kunnen staan). Het aantal acties nodig per vakje is de Manhattan-afstand van zijn huidige positie tot zijn doelpositie. Uit dit vereenvoudigde probleem leiden we m.a.w. de heuristiek h_2 uit vergelijking (2.3) af.

Het tweede vereenvoudigde probleem geeft aanleiding tot nog een andere heuristiek. ∎



Figuur 2.13: Een voorbeeld van een deelprobleem voor de 8-puzzel.

Het is uiterst belangrijk dat de vereenvoudigde problemen efficiënt kunnen opgelost worden, i.e. *zonder het uitvoeren van een zoekalgoritme*! Immers, zoals reeds vermeld is het belangrijk dat een heuristiek efficiënt berekend kan worden.

Opmerking 2.35 Het is interessant om op te merken dat het proces van het vinden van heuristieken kan geautomatiseerd worden. Het programma ABSolver heeft zo een nieuwe heuristiek voor de 8-puzzel ontwikkeld die beter was dan de reeds bestaande (Prieditis, 1993). Het programma vond ook de eerste nuttige heuristiek voor het oplossen van Rubik's kubus. ∎

2.5.2 Patroon Databanken

Een aanvaardbare heuristiek kan ook gevonden worden als de kost van een optimale oplossing voor een *deelprobleem*. Veronderstel dat we in de 8-puzzel de vakjes 5 t.e.m. 8 vervangen door een identiek symbool, bv. een sterretje. In Figuur 2.13 zie je de instanties van dit soort puzzel wanneer dit proces wordt toegepast op de 8-puzzels uit Figuur 2.1. Het is duidelijk dat het minimum aantal stappen nodig om de puzzel aan de linkerkant van Figuur 2.13 om te zetten naar de puzzel aan de rechterkant van Figuur 2.13 een ondergrens is voor het aantal stappen nodig om de originele puzzel op te lossen.

Wat men nu kan doen is een databank aanleggen met voor elk patroon de optimale oplossingskost van het deelprobleem. In dit geval is het aantal deelproblemen $9 \times 8 \times 7 \times 6 \times 5 = 15120$. Het aanleggen van de databank kan (omdat de acties omkeerbaar zijn) vereenvoudigd worden door (graafgebaseerde) breedte-eerst uit te voeren startend vanaf het doelpatroon, en bij te houden wat de afstand is vanaf het doelpatroon (dat gebruikt werd

als initiële toestand). Figuur 2.14 geeft een illustratie van dit proces. Uit de boom in Figuur 2.14 lezen we af dat de optimale oplossingkost van

3	1	2
4	*	
*	*	*

naar

	1	2
3	4	*
*	*	*

gelijk is aan drie.

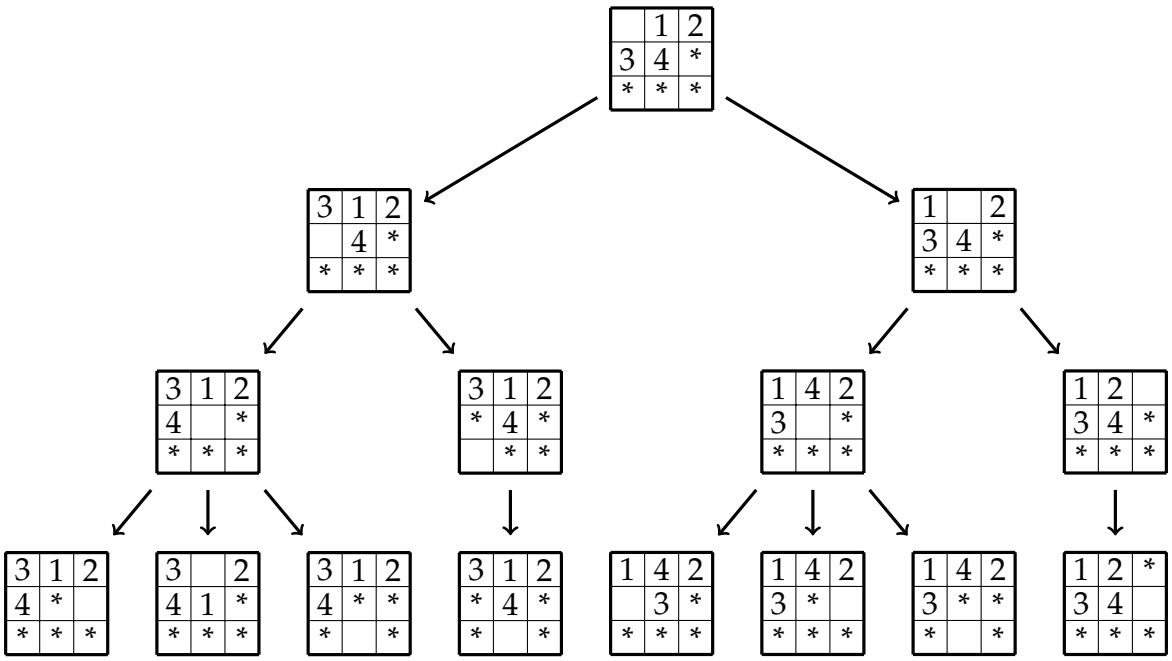
Uiteraard is het tamelijk arbitrair om de vakjes 5 t.e.m. 8 te vervangen door sterretjes. Men kan evengoed de vakjes 1 t.e.m. 4 vervangen. Op die manier bekomt men een tweede heuristiek. Voor sommige probleeminstanties zal de eerste heuristiek een betere (grotere) waarde geven dan de tweede en voor andere probleeminstanties zal dit net omgekeerd zijn. Men kan echter de twee heuristieken *combineren* door het nemen van het maximum:

$$h(s) = \max\big(h_1(s), h_2(s)\big).$$

Het resultaat is een betere heuristiek die nog steeds aanvaardbaar is.

2.6 Oefeningen

1. (Russell and Norvig, 2014, Oefening 3.4) Beschouw twee vrienden die in verschillende steden wonen, bv. in Roemenië. Bij elke actie kunnen we elke vriend simultaan naar een naburige stad op de kaart verplaatsen. De hoeveelheid tijd nodig om zich van stad i naar de aanpalende stad j te verplaatsen is gelijk aan de afstand $d(i, j)$. Bij elke actie moet



Figuur 2.14: Illustratie van het achterwaarts zoeken voor het opbouwen van een patroon databank. De drie eerste lagen van het breedte-eerst zoekproces worden getoond. Er wordt graafgebaseerd zoeken gebruikt dus elk patroon komt hoogstens éénmaal voor in de zoekboom. (Om de tekening te vereenvoudigen tonen we toestanden die niet opnieuw zullen geëxpandeerd worden niet.) Bovendien stopt hier het zoeken pas nadat de volledige boom is opgebouwd; er is m.a.w. geen expliciete doeltoestand. In de figuur worden slechts de eerste drie lagen van de zoekboom getoond.

de vriend die eerst aankomt wachten tot de andere ook aankomt. De twee vrienden willen zo vlug als mogelijk samenkomen.

a) Geef een gedetailleerde beschrijving van het zoekprobleem.

b) Beschouw $D(i,j)$ als de afstand in vogelvlucht tussen de twee steden i en j. Welke van volgende heuristieken zijn toelaatbaar wanneer de eerste vriend zich in stad i en de tweede zich in stad j bevindt.

   i. $D(i,j)$

   ii. $2 \cdot D(i,j)$

   iii. $D(i,j)/2$

c) Zijn er toestanden (in de wetenschap dat er een pad is tussen alle steden op de kaart) waarvoor geen oplossing bestaat? Leg uit.

2. Een aantal robots (bv. k) leven in een rooster waarin sommige locaties muren zijn. Twee robots kunnen zich nooit op dezelfde locatie bevinden. Elke robot heeft zijn eigen bestemming. Bij elke tijdseenheid verplaatsen de robots zich simultaan naar een aanpalend (vrij) vierkant of blijven ze staan. Twee robots die zich naast elkaar bevinden kunnen niet van plaats wisselen in één tijdseenheid. Elke tijdseenheid kost één punt. Beantwoord de volgende vragen:

   a) Geef een minimale correcte voorstelling van een toestand in de toestandsruimte.

   b) Schat de grootte van de toestandsruimte in voor een rooster met als afmetingen $M \times N$.

   c) Welke van volgende heuristieken zijn toelaatbaar? Beargumenteer je antwoord, meer bepaald: geef een situatie waarvan je aangeeft dat de gegeven heuristiek niet toelaatbaar is wanneer je dit beweert.

      i. Som van de Manhattan afstanden voor elke robot tot zijn doellocatie.

      ii. Som van de kosten van de optimale paden indien de robots alleen in de omgeving voortbewegen, m.a.w. zonder obstructie door andere robots.

      iii. Maximum van de Manhattan afstanden vanuit elke robotpositie tot zijn doelpositie.

      iv. Maximum van de kosten van de optimale paden paden indien de robots alleen in de omgeving voortbewegen, m.a.w. zonder obstructie door andere robots.

      v. Aantal robots die zich niet op hun doellocatie bevinden.

3. Beschouw de toestandsruimtegraaf in Figuur 2.15. De starttoestand is steeds S en de doeltoestand is G. Voer een aantal zoekalgoritmes uit op deze toestandsruimtegraaf. Indien er ergens "random" een top uit de open lijst moet gekozen worden, neem je de lexicografisch kleinste top. Hierdoor wordt de oplossing steeds uniek.

   De heuristiek gebruikt door de geïnformeerde zoekmethoden staat gegeven in onderstaande tabel:

toestand	S	A	B	C	D	E	G
h	6	0	6	4	1	10	0

Geef voor onderstaande blinde en geïnformeerde zoekmethodes het pad naar de doeltoestand. Geef ook aan welke toppen geëxpandeerd werden en dit in de juiste volgorde van hun expansie. De geïnformeerde zoekmethoden gebruiken de heuristiek h.

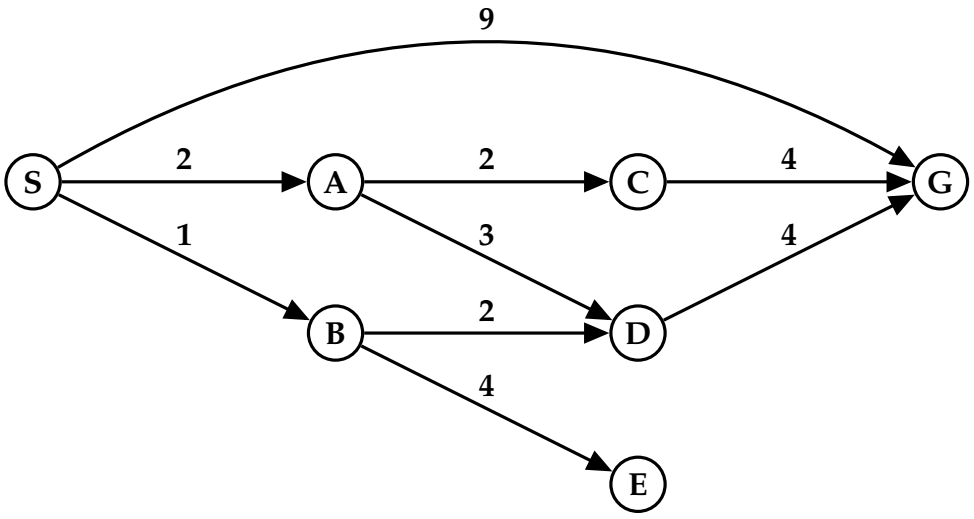
a) Diepte-eerst (boomgebaseerd)

b) Gulzig beste-eerst (boomgebaseerd)

c) Uniforme kost zoeken (boomgebaseerd)

d) A^* (boomgebaseerd)

e) A^* (graafgebaseerd)



Figuur 2.15: Toestandsruimtegraaf voor vraag 3.

4. Pas het gulzig beste eerst algoritme toe om van Arad naar Bucharest te gaan op de kaart van Roemenië in Figuur 2.11. Gebruik de afstand in vogelvlucht uit Tabel 2.2 als heuristiek.

5. Gebruik A* om van Lugoj naar Bucharest te gaan op de kaart van Roemenië in Figuur 2.11 met de afstand in vogelvlucht als heuristiek Teken de opgebouwde zoekboom en geef aan in welke volgorde de plannen verwijderd worden van de open lijst. Los deze oefening eerst op

voor boomgebaseerd zoeken en vervolgens voor graafgebaseerd zoeken.

6. Beschouw het spel Rush Hour zoals aangebracht in Oefening **??** van Hoofdstuk **??**.

   Geef minstens twee aanvaardbare heuristieken voor dit probleem verschillend van de triviale aanvaardbare heuristiek $h = 0$.

Bibliografie

Bellman, R. (1978). *An introduction to artificial intelligence: Can computers think?* Thomson Course Technology.

Nilsson, N. J. (1998). *Artificial Intelligence : a new synthesis*. Morgan Kaufmann Publishers.

Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine learning*, 12(1-3):117–141.

Rich, E. and Knight, K. (1991). *Artificial Intelligence*. McGraw Hill Higher Education, 2 edition.

Russell, S. J. and Norvig, P. (2014). *Artificial Intelligence: A Modern Approach*. Pearson Education, Limited, Harlow, third edition.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.

Winston, P. H. (1992). *Artificial Intelligence*. Addison-Wesley, 3 edition.