# Drie Varianten van het A*-Algoritme

door

Rudy van Vliet

(Wiskunde & Informatica, 1987)

Mei 1994

In het kader van een
Projectstudie Kunstmatige Intelligentie
Docent: dr. I.G. Sprinkhuizen-Kuyper
Rijksuniversiteit te Leiden

# Inhoudsopgave

# Voorwoord

In veel problemen moeten we, om tot een oplossing te komen, een toestandsruimte systematisch doorzoeken. Hiervoor bestaan verschillende algoritmes en één daarvan is het A*-algoritme (spreek uit: A-ster-algoritme). Dit algoritme is geschikt om het kortste pad van een begintoestand naar een verzameling eindtoestanden te bepalen. De eindtoestand die het algoritme vindt of het kortste pad daar naar toe, is dan de oplossing van het probleem.

Om snel een oplossing te vinden, hanteert het A*-algoritme een heuristische functie die meet hoe veelbelovend een toestand is. Omdat het mogelijk is dat een toestand via verschillende paden vanaf de starttoestand bereikbaar is, maakt het algoritme tevens gebruik van het principe van dynamisch programmeren: alleen het kortste pad naar een toestand wordt dan onthouden.

Binnen deze algemene beschrijving van het A*-algoritme bestaan er verschillende varianten. De vraag is dan of er qua efficiëntie grote verschillen zijn tussen deze varianten. Het is zinvol om te weten of er één is die wat dat betreft duidelijk uitsteekt boven de andere of, andersom, of er een variant is die beduidend minder goed presteert. In het kader van een Projectstudie Kunstmatige Intelligentie bij dr. I.G. Sprinkhuizen-Kuyper voor mijn studie Informatica aan de Rijksuniversiteit Leiden heb ik daarom onderzoek gedaan naar de efficiëntie van drie van deze varianten. Ik heb me daarbij gericht op de werking van deze varianten voor eindige toestandsruimtes die bovendien voor aanvang van het zoekproces volledig bekend zijn. Mijn bevindingen zijn in dit verslag vastgelegd.

In het eerste hoofdstuk zal in het kort het algemene zoekprobleem worden geschetst dat we met een algoritme willen oplossen. Ook worden daarin de graafrepresentatie van een toestandsruimte en enige notaties geïntroduceerd. Hoofdstuk 2 bevat een beschrijving van drie bekende algemene zoektechnieken: Breadth-First-Search, Depth-First-Search en Best-First-Search. Het A*-algoritme, waarin het idee van Best-First-Search is uitgewerkt, komt in hoofdstuk 3 aan bod. Daarbij wordt het algemene kader van het algoritme, waarbinnen de verschillende varianten vallen, beschreven. Een bewijs van de correcte werking van het A*-algoritme, alsmede enkele prettige eigenschappen van het algoritme wanneer de heuristische functie aan bepaalde voorwaarden voldoet, vinden we in hoofdstuk 4. In hoofdstuk 5 worden de drie varianten van het algoritme die ik heb onderzocht, gegeven. Van deze drie varianten wordt de laatste in hoofdstuk 6 grondig geanalyseerd. Vervolgens worden de varianten in hoofdstuk 7 met elkaar vergeleken. Dit wordt gedaan aan de hand van theoretische overwegingen, geconstrueerde voorbeelden en een serie willekeurige voorbeelden van verschillende vorm en grootte. Tenslotte trek ik in hoofdstuk 8 enkele conclusies uit het onderzoek.

# Hoofdstuk 1

# Inleiding

In deze inleiding vertellen we eerst in het kort iets over het basisidee achter veel algoritmes: het zoeken in een toestandsruimte. Vervolgens vertalen we het begrip toestandsruimte en alle daarmee samenhangende begrippen naar corresponderende termen in de theorie van grafen, waarna we enige notationele afspraken maken.

## 1.1 Zoeken in een Toestandsruimte

In veel problemen die niet volslagen triviaal zijn, zodat de oplossing niet direcht zichtbaar is, zullen we systematisch moeten zoeken om de (een) oplossing te bereiken. Zo'n zoekproces kan worden opgevat als een wandeling door een toestandsruimte: een verzameling toestanden met mogelijke overgangen tussen de toestanden. We starten met wandelen in een begintoestand of starttoestand, en door gebruik te maken van de toestandsovergangen hopen we te eindigen in een gewenste eindtoestand. Die eindtoestand of het pad naar die eindtoestand (de reeks gebruikte overgangen om van die begintoestand in die eindtoestand te komen) representeert dan de gezochte oplossing. De verzameling eindtoestanden zullen we ook wel het doel noemen. Merk op dat er in principe meerdere begin- en eindtoestanden kunnen zijn.

Vaak gaat het er niet om om alleen maar *een* pad naar een eindtoestand te vinden, maar liefst ook het kortste pad. Hierbij kan het kortste pad een pad zijn dat uit zo min mogelijk toestandsovergangen bestaat, of, als er aan elke toestandsovergang kosten zijn verbonden, een pad met minimale totale kosten. We eisen dat de kosten van elke toestandsovergang groter dan of gelijk aan 0 zijn. Ook moeten de kosten eindig zijn. Mochten er aan een toestandsovergang toch oneindig grote kosten verbonden zijn, dan kunnen we die toestandsovergang net zo goed verwijderen. Zo'n overgang kunnen we toch niet benutten. Als de kosten van elke overgang gelijk zijn aan 1, dan is het minimale-kosten pad gewoon het pad met het kleinste aantal toestandsovergangen.

## 1.2 Graafrepresentatie van een Toestandsruimte

Een toestandsruimte zoals we die in de vorige paragraaf hebben geïntroduceerd, kan op een natuurlijke wijze gerepresenteerd worden door een graaf. De toestanden corresponderen dan met knopen, de toestandsovergangen komen overeen met takken tussen knopen. Een overgang tussen twee toestanden die in beide richtingen te gebruiken is, gaat over in een niet-gerichte tak; als de toestandsovergang in maar één richting gebruikt kan worden, wordt die gerepresenteerd door een gerichte tak. De (niet-negatieve)

kosten van een overgang vormen het gewicht van de bijbehorende tak. We noteren het gewicht van de tak van knoop $k_1$ naar $k_2$ met $c(k_1, k_2)$. Een pad is in deze situatie een rij aaneengesloten takken: als één tak uit het pad eindigt in een bepaalde knoop, moet de volgende tak van het pad beginnen in die knoop.

Het is nu de bedoeling om vanuit een beginknoop (startknoop) door het volgen van takken uit de graaf in een eindknoop (doelknoop) terecht te komen. Als we dit willen bereiken via een minimale-kosten pad, moet de som van de gewichten van de takken op een pad zo klein mogelijk zijn. We spreken dan ook wel over een pad met minimale lengte. Een zoekboom is de boom van paden die we aflopen in een zoekproces. In sommige gevallen heeft de graafrepresentatie van een toestandsruimte zelf al een boomstructuur: er is dan voor elke (bereikbare) knoop in de graaf maar één pad van de (ene) startknoop naar die knoop.

In het vervolg van dit verslag zullen we meestal spreken in termen van grafen: knopen, takken, etc. Daarbij moeten we in het achterhoofd houden dat we die termen zonder problemen kunnen vervangen door de corresponderende uitdrukkingen uit toestandsruimtes. Als het zo uitkomt, zullen we echter ook expliciet de laatste naamgeving gebruiken. We zullen in dit verslag alleen grafen met een eindig aantal knopen bekijken die bovendien van tevoren volledig bekend zijn.

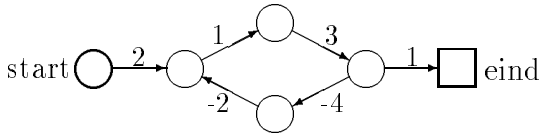
Vanaf nu zullen we aannemen dat we in de graaf (precies) één startknoop $s$ hebben. Dit is geen echte beperking, want we kunnen desgewenst zonder negatieve consequenties een extra knoop aan de graaf toevoegen die alleen uitgaande takken met gewicht 0 heeft naar de oorspronkelijke startknopen. Als we dan die extra knoop uitroepen tot de enige startknoop voor het zoekproces, hebben we een graaf geconstrueerd die voor het zoekproces equivalent is aan de originele graaf, maar die nog maar één startknoop bevat. De mogelijkheid van meerdere eindknopen houden we nog wel helemaal open. Dit verschil in beperking tussen begin- en eindknopen sluit aan bij de intuïtieve interpretatie van een (zoek-)boom: er is één beginknoop: de wortel van de boom en er zijn meerdere eindknopen mogelijk: de bladeren van de boom.

Verder nummeren we de knopen van 1 tot en met $N$: het totaal aantal knopen. Tenslotte nemen we aan dat de graaf geen takken van een knoop naar zichzelf bevat. Dit is ook geen ernstige beperking van de mogelijkheden: omdat elke tak een gewicht heeft dat minstens 0 is, heeft het toch geen nut om van een knoop naar zichzelf te wandelen.

In de figuren in dit verslag zijn knopen getekend als een cirkel met eventueel het nummer van de knoop erin. Een uitzondering hierop vormen de eindknopen. Eindknopen worden aangegeven met een vierkant. Een startknoop is steeds te herkennen aan de vettere lijn waarmee zo'n knoop getekend is. Als er bij een knoop met een kleiner lettertype een getal staat, dan is dat de waarde van een functie voor die knoop.

Een pijl tussen twee knopen stelt een gerichte tak voor. Een niet-gerichte tak is in feite gelijkwaardig aan twee gerichte takken met hetzelfde gewicht. Zo'n tak wordt aangeduid met een lijn tussen de betreffende knopen. Als een tak voorzien is van een getal, houdt dat zijn gewicht in. Als er niet expliciet een gewicht is aangegeven, komt dit overeen met een gewicht 1.

Omdat het gewicht van elke tak minstens 0 is, kunnen er geen cykels van takken met een negatief totaal gewicht ontstaan. Met zulke cykels zou er geen minimale-kosten pad van de startknoop naar een eindknoop hoeven bestaan. We kunnen dan namelijk in principe oneindig vaak zo'n cykel doorlopen, waarbij we elke keer de lengte van het afgelopen pad met dezelfde hoeveelheid verlagen (zie Fig. 1.1).




Figuur 1.1: Een cykel met negatieve kosten

# Hoofdstuk 2

# Algemene Zoektechnieken

Tijdens een zoekproces stellen we ons voortdurend de vraag in welke richting we verder zullen zoeken. Er zijn verschillende zoektechnieken, procedures om het zoekproces te sturen, die deze vraag op verschillende wijze beantwoorden. We zullen hier drie algemene zoektechnieken kort bespreken, te weten: Breadth-First-Search, Depth-First-Search en Best-First-Search. In de volgende hoofdstukken zullen we dan dieper ingaan op een algoritme dat gebruikt maakt van Best-First-Search: het A*-algoritme.

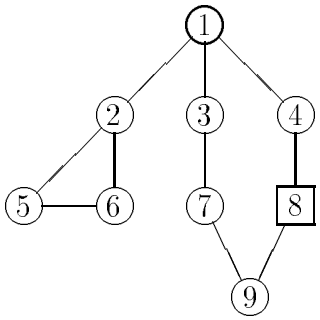
De drie genoemde algemene zoektechnieken kunnen op verschillende manieren beschreven worden. De beschrijfwijze en notatie die we in dit verslag hanteren, zijn afgeleid van die in [Sprinkhuizen-Kuyper1990]. Hierbij maken al de drie technieken gebruik van een lijst *OPEN* van geopende knopen of *Open* knopen: knopen die al wel zijn ontdekt, maar nog niet volledig zijn onderzocht (let overigens op het gebruik van de hoofdletters in *OPEN* en *Open*). De knopen in *OPEN* zijn volgens een bepaald criterium geordend. We kunnen derhalve spreken van de voorste en ook van de achterste knoop in *OPEN*. Naast de *Open* knopen kennen we verder nog *Onbekende* en *Gesloten* knopen. *Onbekende* knopen zijn knopen die we nog helemaal niet zijn tegengekomen tijdens het zoeken; *Gesloten* knopen, daarentegen, zijn knopen die we we al onderzocht hebben; zij zijn (voorlopig) niet meer van belang voor het zoekproces.

Aan het begin van het zoekproces bevat de lijst *OPEN* alleen de startknoop. Alle andere knopen zijn dan nog *Onbekend*.

Tijdens het zoekproces wordt steeds de voorste knoop $k_1$ uit *OPEN* gehaald en deze wordt onderzocht (ontwikkeld/geëxpandeerd). Dit houdt in dat allereerst wordt bekeken of $k_1$ soms een eindknoop is. Is dat het geval, dan zijn we klaar; we hoeven niet meer verder te zoeken. Is dat niet het geval, dan kijken we welke knopen allemaal in één stap: dat wil zeggen: via één tak, bereikbaar zijn vanuit $k_1$. Deze knopen noemen we de opvolgers van $k_1$. Het bepalen van deze opvolgers wordt ook wel het genereren van de opvolgers van $k_1$ genoemd. Vervolgens verwijderen we volgens een bepaalde regel eventueel enkele van deze opvolgers, omdat die toch niet meer tot een optimale oplossing kunnen leiden. De resterende opvolgers verklaren we geopend en plaatsen we op de hun toebedachte positie in de lijst *OPEN*. Op dat moment worden deze opvolgers gepromoveerd tot kinderen van $k_1$; $k_1$ heet dan hun ouder. De onderzochte knoop $k_1$ wordt nu *Gesloten*.

De zoekboom bestaat op een bepaald moment in het zoekproces uit de dan *Open* of *Gesloten* knopen, met takken tussen elke knoop (behalve de startknoop $s$) en zijn ouder. De startknoop is de wortel van de zoekboom. Het pad vanaf $s$ naar een knoop $k$ is dus het pad $(s, \ldots, Ouder(Ouder(k)), Ouder(k), k)$.

Het zoekproces stopt wanneer we, zoals we al zagen, een eindknoop uit *OPEN* halen




Figuur 2.1: Voorbeeldgraaf voor dit hoofdstuk

of wanneer *OPEN* geen knopen meer bevat.

Merk op dat we ook spreken van het expanderen van een knoop $k_1$ als $k_1$ een eindknoop is en we vrijwel niets met die knoop doen nadat we hem uit *OPEN* gehaald hebben.

De drie algemene zoektechnieken verschillen hoofdzakelijk in de positie binnen de lijst *OPEN* waar nieuw geopende knopen worden geplaatst. Dat kan achteraan, vooraan of, afhankelijk van de knoop op elke positie zijn. Verder kan binnen een bepaalde zoektechniek nog onderscheid gemaakt worden naar de regels die voor het verwijderen van gegenereerde opvolgers worden gehanteerd.

De algoritmische beschrijvingen van de drie zoektechnieken die in dit hoofdstuk gegeven zullen worden, zijn niet al te gedetailleerd. Een gevolg daarvan is dat er dubbelzinnigheden kunnen ontstaan, met name wat betreft de status van de knopen. In principe bevat de lijst *OPEN* namelijk *Open* knopen, maar het kan voorkomen dat een knoop meerdere keren in *OPEN* staat. Als zo'n knoop er nu met zijn voorste voorkomen uitgehaald en vervolgens ontwikkeld wordt, zou hij daarna *Gesloten* zijn, terwijl hij met zijn andere voorkomen nog in *OPEN* staat. Omwille van de eenvoud besteden we hier verder echter geen aandacht aan. Het gaat ons in dit hoofdstuk tenslotte vooral om de globale werking van de drie zoektechnieken. We zullen bij het A*-algoritme dat we in hoofdstuk 3 geven, zien dat de genoemde dubbelzinnigheid zich daarbij niet voordoet.

Voor alle algoritmes die we in dit verslag beschrijven, maken we gebruik van 'inspringen' om de reikwijdte van, bijvoorbeeld, een lus of een voorwaardelijke instructie aan te geven. We vertrouwen erop dat deze impliciete notatie de structuur van de algoritmes duidelijk weergeeft.

Ter verduidelijking en verluchtiging zullen we voor elk van de drie zoektechnieken na de algoritmische beschrijving de werking ervan illustreren aan de hand van een voorbeeldgraaf. De graaf die we daarvoor gebruiken, is die van Fig. 2.1.

## 2.1 Breadth-First-Search

Bij Breadth-First-Search (in de breedte zoeken / zijwaarts zoeken) onderzoeken we achtereenvolgens de startknoop $s$, de knopen die in één stap vanaf $s$ bereikbaar zijn

(de knopen op niveau 1), de knopen die in twee stappen vanaf $s$ bereikbaar zijn (de knopen op niveau 2), enz. We houden op elk moment een groot aantal verschillende paden van bijna dezelfde lengte (aantal stappen) in ons achterhoofd. Zij vullen de volledige breedte van de graaf tot een bepaald niveau. Deze situatie bereiken we door de nieuw geopende knopen steeds achter aan *OPEN* toe te voegen. De lijst heeft dus de First-In-First-Out eigenschap: de knopen die er het eerst in gezet worden, worden er ook weer het eerst uitgehaald.

Het algemene algoritme voor Breadth-First-Search wordt dus:

*OPEN* := [$s$];
*Eind* := 0; (* nog geen eindknoop gevonden *)

WHILE (*OPEN* <> []) AND (*Eind* = 0) DO
  haal voorste knoop $k_1$ uit *OPEN*;
  IF $k_1$ is een eindknoop THEN
    *Eind* := $k_1$
  ELSE
    genereer alle opvolgers van $k_1$;
(*) verwijder volgens een geschikt criterium eventueel een aantal opvolgers;
    voeg de resterende opvolgers in volgorde van ontdekking
      aan het eind van *OPEN* toe (achteraan); (* zij worden dus *Open* *)
  $k_1$ :=*Gesloten*;

Wanneer het algoritme eindigt met *Eind* = 0, hebben we kennelijk geen eindknoop bereikt. Dit gebeurt dan en slechts dan als er geen pad van de startknoop $s$ naar het doel bestaat.

Bij (*) kunnen we er voor kiezen om een aantal opvolgers van $k_1$ te verwijderen, zodat ze niet achter in *OPEN* gezet zullen worden. Er zijn verschillende criteria om te beslissen of een opvolger verwijderd moet worden of niet.

Wanneer de takken in de graaf ongericht zijn, zoals in ons voorbeeld, dan is startknoop $s$ weer in één stap bereikbaar vanaf al zijn kinderen. Het heeft echter weinig zin om in een zoekproces eerst van $s$ naar een kind te wandelen en dan weer terug. Daarom ligt het voor de hand om $s$ als opvolger van zijn kinderen te verwijderen. We kunnen dit veralgemeniseren tot een eerste regel voor het verwijderen van zojuist gegenereerde opvolgers:

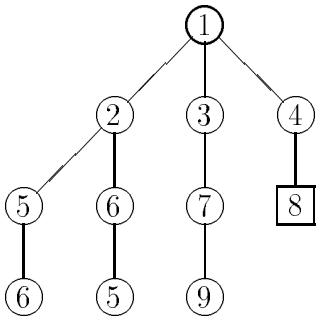
- verwijder elke opvolger die tevens een voorouder van de ontwikkelde knoop is

De regel houdt in dat we elke opvolger van een knoop $k_1$ die al op het pad van $s$ naar $k_1$ ligt, verwijderen. Op zo'n manier voorkomen we dat we (zonder zin) in cykels gaan rondlopen. Wanneer we deze regel hanteren, wordt de boom in Fig. 2.2 de zoekboom voor Breadth-First-Search in onze voorbeeldgraaf.

De lijst *OPEN* ziet er in dit geval achtereenvolgens als volgt uit: $[1]$, $[2,3,4]$, $[3,4,5,6]$, $[4,5,6,7]$, $[5,6,7,8]$, $[6,7,8,6]$, $[7,8,6,5]$, en tenslotte $[8,6,5,9]$.

Andere regels dan de regel die hier gebruikt is voor het verwijderen van opvolgers, kunnen zijn:

- verwijder geen enkele opvolger
- verwijder elke opvolger die al *Gesloten* is




Figuur 2.2: Zoekboom voor Breadth-First-Search in onze voorbeeldgraaf

- verwijder elke opvolger die al *Open* of *Gesloten* is

Welke regel we ook gebruiken, met Breadth-First-Search vinden we altijd het pad met het kleinste aantal stappen naar een eindknoop, als zo'n pad bestaat. Met eventueel verschillende gewichten (= kosten) op de takken houdt deze zoektechniek geen rekening.

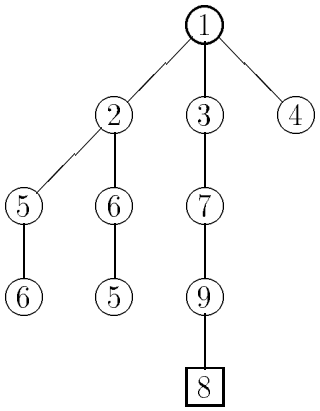
## 2.2 Depth-First-Search

De zoektechniek Depth-First-Search (in de diepte zoeken / voorwaarts zoeken) kenmerkt zich doordat eerst een pad in zijn volle diepte wordt onderzocht op de aanwezigheid van een eindknoop, voordat een ander pad zo nodig nog wordt geprobeerd. We lopen van $s$ naar zijn eerst gegenereerde kind, vervolgens van dat kind naar diens eerst gegenereerde kind, enz. Pas als we op een knoop stuiten die geen kinderen (meer) heeft, kijken we bij zijn ouder (één niveau hoger in de zoekboom) of die nog andere kinderen heeft. Deze volgorde van zoeken bereiken we door de kinderen van een onderzochte (en vóór uit *OPEN* gehaalde) knoop weer vóór in *OPEN* te plaatsten, het eerste kind meest vooraan. De knoop die het laatst in *OPEN* wordt gezet, komt als eerste in aanmerking om er weer uitgehaald te worden; de lijst *OPEN* heeft dus de Last-In-First-Out eigenschap.

Aldus wordt het algemene algoritme voor Depth-First-Search:

*OPEN* := [$s$];
*Eind* := 0; (* nog geen eindknoop gevonden *)

WHILE (*OPEN* <> []) AND (*Eind* = 0) DO
  haal voorste knoop $k_1$ uit *OPEN*;
  IF $k_1$ is een eindknoop THEN
    *Eind* := $k_1$
  ELSE
    genereer alle opvolgers van $k_1$;
(*) verwijder volgens een geschikt criterium eventueel een aantal opvolgers;




Figuur 2.3: Zoekboom voor Depth-First-Search in onze voorbeeldgraaf

voeg de resterende opvolgers in volgorde van ontdekking
aan het begin van *OPEN* toe (vooraan); (* zij worden dus *Open* *)
$k_1$ := *Gesloten*;

Bij Depth-First-Search kunnen we in principe dezelfde vier regels gebruiken om opvolgers van de onderzochte knoop $k_1$ eventueel te verwijderen (bij (*)) als bij Breadth-First-Search. De optie om geen enkele opvolger te verwijderen is echter bijzonder gevaarlijk als het mogelijk is om in een cykel in de graaf rond te lopen. Dit wordt op heel eenvoudige wijze duidelijk als we weer naar ons voorbeeld kijken. Wanneer knoop $s = 1$ het eerste kind is van zijn eerste kind knoop 2, wat we gezien de nummering mogen verwachten, dan zal het algoritme bij het ontwikkelen van knoop 1 knoop 2 vóór in *OPEN* zetten, waarna bij het ontwikkelen van knoop 2 knoop 1 weer vooraan komt te staan. Op zo'n manier blijft het zoekproces beperkt tot de knopen 1 en 2 en zullen we nooit de gezochte eindknoop vinden. Net zo kan het mis gaan als de graaf grotere cykels bevat.

Daarom zullen we op z'n minst een opvolger van knoop $k_1$ moeten verwijderen als die ook een voorouder van $k_1$ blijkt te zijn. Wanneer we dat doen zal het algoritme voor elke instantie een eindknoop bereiken, als er tenminste één bereikbaar is vanuit $s$. Het zal daarvoor echter in het algemeen niet het kortste pad gebruiken, ongeacht of we daar het pad met de minste stappen of dat met de kleinste lengte onder verstaan.

Ook bij ons voorbeeld levert het toepassen van Depth-First-Search niet het pad met de minste stappen op. Wanneer we alleen de voorouders van een knoop verwijderen, is de zoekboom namelijk de boom in Fig. 2.3.

In de loop van het zoekproces is de lijst *OPEN* achtereenvolgens gelijk aan: [1], [2, 3, 4], [5, 6, 3, 4], [6, 6, 3, 4], [6, 3, 4], [5, 3, 4], [3, 4], [7, 4], [9, 4] en als laatste [8, 4].

## 2.3 Best-First-Search

De zoektechniek Best-First-Search (eerst via de beste zoeken) is gebaseerd op een heuristische functie $h$ die een maat is voor de veelbelovendheid van de knopen in de graaf. Een graaf is dan meer dan alleen een verzameling knopen en een verzameling (al dan niet gerichte) gewogen takken tussen die knopen. Aan elke knoop $k$ is een getal $h(k)$ verbonden dat iets zegt over de kwaliteit van de knoop. Zo kan $h(k)$ een schatting zijn van de afstand van knoop $k$ tot het doel. Hierbij is die afstand de lengte van het kortste pad van de knoop tot een eindknoop. Wanneer we bijvoorbeeld in een netwerk van wegen, straten en paden een korte route naar een zeker doel willen bepalen, zouden we op elk kruispunt van wegen als heuristische-functiewaarde de hemelsbreed-afstand tot het doel kunnen nemen.

Bij Best-First-Search ontwikkelen we dan steeds die *Open* knoop $k_1$ die de laagste waarde $h(k_1)$ heeft. Een eenvoudige manier om die knoop te bepalen is door de lijst *OPEN* gesorteerd te houden op de $h$-waarde van de knopen: lage waardes vooraan, hoge waardes achteraan. Dit kunnen we bereiken door elke keer als we een nieuw-gegenereerd kind aan *OPEN* willen toevoegen, dit op zo'n positie te doen dat *OPEN* gesorteerd blijft (invoeg-sorteren). Dan kunnen we net als bij de andere zoektechnieken steeds de voorste knoop uit de lijst halen.

Het algoritme voor Best-First-Search wordt dus:

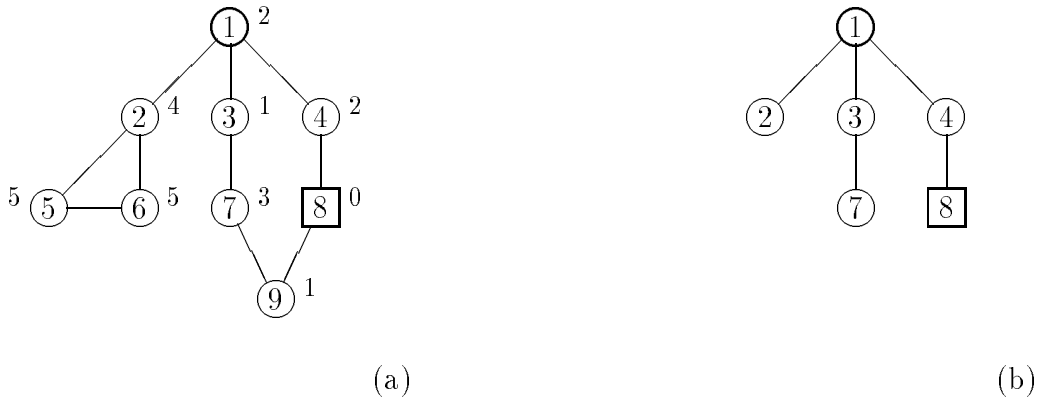
*OPEN* := [$s$];
*Eind* := 0; (* nog geen eindknoop gevonden *)

WHILE (*OPEN* <> []) AND (*Eind* = 0) DO
  haal voorste knoop $k_1$ uit *OPEN*; (* met de laagste $h$-waarde *)
  IF $k_1$ is een eindknoop THEN
    *Eind* := $k_1$
  ELSE
    genereer alle opvolgers van $k_1$;
(*) verwijder volgens een geschikt criterium eventueel een aantal opvolgers;
    voeg de resterende opvolgers aan *OPEN* toe
      zo dat *OPEN* gesorteerd blijft op $h$-waarde; (* zij worden dus *Open* *)
  $k_1$ := *Gesloten*;

Met Best-First-Search combineren we de voordelen van Breadth-First-Search en Depth-First-Search. Als de functie $h$ betrouwbaar is, kunnen we een goed pad naar een eindknoop achter elkaar een heel eind opbouwen, zonder ons om andere paden te hoeven bekommeren (net als bij Depth-First-Search). Aan de andere kant, als een pad toch minder veelbelovend blijkt dan het aanvankelijk leek, zijn we niet gedwongen om het tot het eind te blijven volgen. We kunnen dan zonder moeite overstappen op een ander pad (zoals we bij Breadth-First-Search voortdurend doen).

Om te voorkomen dat we tussen een paar veelbelovende knopen heen en weer blijven lopen is het opnieuw sterk aan te raden om bij (*) in ieder geval de opvolgers van een ontwikkelde knoop die tevens een voorouder van hem zijn te verwijderen.

We kunnen deze variant van Best-First-Search ook op ons voorbeeld, uitgebreid met een heuristische functie, toepassen. Laat $h$ zijn als aangegeven in Fig. 2.4(a), dan ziet de zoekboom voor Best-First-Search eruit als in Fig. 2.4(b).




Figuur 2.4: (a) Onze voorbeeldgraaf uitgebreid met een heuristische functie $h$; (b) de bijbehorende zoekboom voor Best-First-Search

De lijst *OPEN* bevat voor dit voorbeeld respectievelijk [1], $[3, 4, 2]$, $[4, 7, 2]$ en $[8, 7, 2]$. In dit geval vinden we met Best-First-Search het kortste pad van de startknoop naar het doel. Er zijn echter eenvoudig instanties te bedenken waarvoor dat niet gebeurt.

# Hoofdstuk 3

# Het A*-Algoritme

Het A*-algoritme is een algoritme om in een graaf het kortste pad van de startknoop naar het doel te vinden. Het maakt hiervoor gebruik van een vorm van Best-First-Search. Voordat we over het algoritme zelf kunnen uitweiden, hebben we eerst enkele definities nodig.

## 3.1 Definities

De kwaliteit van een knoop $k$ wordt in het A*-algoritme niet alleen bepaald door de waarde $h(k)$, maar tevens door de waarde $g(k)$ met

$g(k)$ = de lengte van het kortste pad van de startknoop $s$ naar knoop $k$ dat we tot nu toe zijn tegengekomen

In paragraaf 2.3 zagen we al dat $h(k)$ een schatting voor de afstand van $k$ tot het doel kan voorstellen. In het A*-algoritme is dat inderdaad het geval, ofwel

$h(k)$ = een schatting van de lengte van het kortste pad van knoop $k$ naar een eindknoop

Omdat alle takken een niet-negatief gewicht hebben, is het redelijk om te eisen dat $h(k) \geq 0$ voor elke knoop $k$. Met deze functie $h$ is

$f(k)$ = $g(k) + h(k)$ = een schatting van de lengte van het kortste pad van de startknoop $s$ via knoop $k$ naar een eindknoop

Als we verder definiëren

$g^*(k)$ = de lengte van het kortste pad van de startknoop $s$ naar knoop $k$
$h^*(k)$ = de lengte van het kortste pad van knoop $k$ naar een eindknoop
$f^*(k)$ = $g^*(k) + h^*(k)$ = de lengte van het kortste pad van de startknoop $s$ via knoop $k$ naar een eindknoop

dan zijn $g$, $h$ en $f$ schatters van $g^*$, $h^*$, respectievelijk $f^*$. Het is duidelijk dat $g(s) = g^*(s) = 0$ en dat $g(k) \geq g^*(k)$ voor elke knoop $k$. Verder geldt voor elke eindknoop $k$ dat $h^*(k) = 0$. Het is ook eenvoudig in te zien dat $f^*(s) =$ de lengte van het kortste pad van de startknoop $s$ naar een eindknoop. Naar zo'n kortste pad zijn we uiteindelijk op zoek. We noteren de lengte van het (een) kortste pad met $L_0$.

Voor het A*-algoritme eisen we dat $h(k) \leq h^*(k)$ voor elke knoop $k$. We noemen $h$ dan een onderschatter van $h^*$. Kennelijk is $h$ optimistisch ten aanzien van de nog af te

leggen afstand tot het doel. Een gevolg hiervan is dat voor elke eindknoop $k$ ook geldt dat $h(k) = 0$, omdat $0 \leq h(k) \leq h^*(k) = 0$. Later, in paragraaf 4.4, zullen we zien waarom het zo belangrijk voor het A*-algoritme is dat $h$ een onderschatter van $h^*$ is. Als er minstens één knoop $k$ is met $h(k) > h^*(k)$, heet $h$ een overschatter van $h^*$. In dat geval gaat het A*-algoritme over in het A-algoritme.

De eis dat $h$ een onderschatter is, wordt niet in elke definitie van het A*-Algoritme opgenomen. In [Nilsson1982] en [Nau,Kumar,Kanal1984] komt hij echter gewoon voor. Vanwege zijn grote belang hebben wij deze eis dan ook overgenomen.

Merk op dat in het voorbeeld uit paragraaf 2.3 $h$ een overschatter is, want bijvoorbeeld $h(4) = 2 = h^*(4) + 1$ en $h(7) = 3 = h^*(7) + 1$.

Indien voor elke tak in de graaf van knoop $k_1$ naar knoop $k_2$ geldt dat $h(k_1) \leq c(k_1, k_2) + h(k_2)$, wordt $h$ een monotone schatter genoemd. Wanneer we ons realiseren dat $h(k)$ een schatting is voor de afstand van knoop $k$ naar het doel, blijkt de voorwaarde dat $h(k_1) \leq c(k_1, k_2) + h(k_2)$ een heel natuurlijke te zijn; het is dan gewoon een soort driehoeksongelijkheid. Als voor elke knoop $k$ zelfs geldt dat $h(k) = h^*(k)$, heet $h$ een perfecte schatter. In paragraaf 4.5 komen we hier nog op terug.

## 3.2 Het Algoritme

Omdat we met het A*-algoritme het kortste pad van de startknoop naar het doel willen vinden, hanteren we de $f$-waarde van de knopen om te beslissen welke knoop we uit de lijst *OPEN* zullen halen. Die waarde is tenslotte een schatting van de lengte van het kortste pad van de startknoop naar een eindknoop dat via knoop $k$ loopt. De knoop met de laagste waarde wordt het eerst geëxpandeerd. De opvolgers die daarbij gegenereerd worden en die aan *OPEN* toegevoegd moeten worden, worden daar in oplopende volgorde van hun $f$-waardes ingezet.

Naast de functie $f$ maakt het algoritme gebruik van het principe van dynamisch programmeren. Dit houdt in dat als we meerdere paden naar een bepaalde knoop tegenkomen, we alleen het beste pad onthouden en het andere pad, danwel de andere paden 'vergeten'. In ons geval heeft dat betrekking op de functie $g$. Stel dat we op een gegeven moment voor een knoop $k$ een beste pad vanaf de startknoop, met lengte $g(k)$ hebben genoteerd en dat we vervolgens door ontwikkeling van één of andere knoop een beter pad naar $k$ ontdekken. Dan hoeven we niet meer verder te zoeken via het oude pad, omdat dat toch nooit meer het kortste pad van start naar doel kan opleveren. Alleen het betere pad is dan nog van belang en we kunnen de oude waarde van $g(k)$ inruilen voor de nieuwe. Omdat $f(k) = g(k) + h(k)$, kan ook $f(k)$ aangepast worden.

Een gevolg van het op deze manier toepassen van dynamisch programmeren is dat een opvolger $k_2$ van een zojuist ontwikkelde knoop $k_1$ verwijderd kan worden, als we al eerder een pad naar $k_2$ hebben ontdekt met een lengte kleiner dan of gelijk aan de lengte van het pad via $k_1$. Omdat we eisen dat alle takken een niet-negatief gewicht hebben, impliceert dit in het bijzonder dat een opvolger van $k_1$ verwijderd wordt als die tevens een voorouder van $k_1$ blijkt te zijn.

Aan het eind van het algoritme willen we nu ook het (kortste) pad van de startknoop naar een eindknoop kunnen reconstrueren, als we tenminste zo'n pad hebben gevonden. Daarom onthouden we voor elke knoop die we tot een bepaald moment hebben ontdekt in het algoritme, het beste pad tot dan toe vanaf de startknoop. Misschien zijn er soms meerdere beste paden met dezelfde lengte, maar dan heeft het toch geen zin om er meer dan één te onthouden. Die beste paden slaan we eenvoudig op door voor elke knoop

zijn voorganger in het pad (zijn ouder dus) te onthouden.

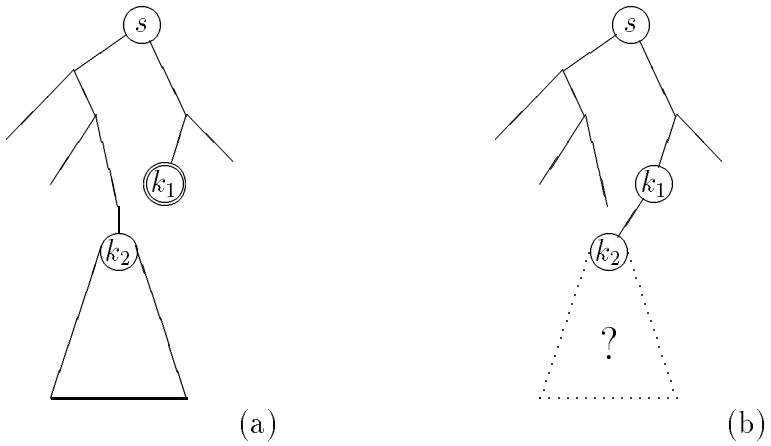
Wanneer we dit alles in acht nemen, krijgen we de volgende pseudo-code voor ons A*-algoritme:

```
FOR alle knopen k DO
   f[k] := ∞;
   Ouder[k] := 0;
   Status[k] := Onbekend;

g[s] := 0;
f[s] := g[s] + h[s];
Status[s] := Open;
OPEN := [s];
Eind := 0;        (* nog geen eindknoop gevonden *)

WHILE (OPEN <> []) AND (Eind = 0) DO
   haal voorste knoop k1 uit OPEN;        (* met de laagste f-waarde *)
   IF k1 is een eindknoop THEN
      Eind := k1
   ELSE
      FOR alle knopen k2 die in 1 stap vanuit k1 bereikbaar zijn DO
         gg := g[k1] + c[k1, k2];
         ff := gg + h[k2];
         IF ff < f[k2] THEN        (* we hebben een (beter) pad naar k2 gevonden *)
            g[k2] := gg;
            f[k2] := ff;
            IF Status[k2] = Gesloten THEN
(*)            onderneem verdere actie
            ELSE
               IF Status[k2] = Open THEN
                  haal k2 uit de lijst OPEN        (* met zijn oude f-waarde *)
               ELSE
                  Status[k2] := Open;
               voeg k2 toe aan de lijst OPEN
                  zo dat OPEN gesorteerd blijft op f-waarde;
            Ouder[k2] := k1;
   Status[k1] := Gesloten;
```

Als dit algoritme eindigt met $Eind <> 0$, hebben we een pad van de startknoop $s$ naar de eindknoop $Eind$ gevonden. Dat pad is $(s, \ldots, Ouder(Ouder(Eind)), Ouder(Eind), Eind)$.

We definiëren een stap van het A*-algoritme als het uit *OPEN* halen en expanderen van een knoop $k_1$. Bij het expanderen hoort ook het volledig administratief verwerken van de eventueel gevonden verbeteringen in de $f$-waardes. Een stap is kortom wat er in één executie van het inwendige van de WHILE-lus wordt gedaan.

Merk op dat elke knoop volgens dit algoritme op een bepaald moment hooguit één keer in *OPEN* kan staan. Als een knoop $k_2$ in *OPEN* wordt gezet, geldt immers òf dat hij daarvoor niet *Open* was, òf dat hij eerst uit *OPEN* gehaald is, omdat hij via




Figuur 3.1: (a) De zoekboom van het A*-algoritme juist vóórdat er een beter pad van de startknoop $s$ naar knoop $k_2$ (via knoop $k_1$) wordt ontdekt; (b) de zoekboom nadat het betere pad naar $k_2$ is ontdekt; de $g$- en $f$-waardes van de nakomelingen van $k_2$ kloppen niet meer en het is de vraag wat we daarmee zullen doen.

$k_1$ een betere $f$-waarde krijgt. Als gevolg hiervan kan de lijst *OPEN* nooit meer dan $N$ elementen bevatten (zelfs niet meer dan $N-1$, want de startknoop $s$ staat alleen aan het begin van het algoritme, in z'n eentje in *OPEN*). Omdat bovendien bij het ontwikkelen van een knoop die knoop zelf niet als kind gegenereerd zal worden, volgt nu tevens dat er geen dubbelzinnigheid kan ontstaan over de status van een knoop. Dit kon bij de beschrijving van de algoritmes in hoofdstuk 2 nog wel gebeuren.

Eén onderdeel van het algoritme is nog niet precies ingevuld. Dat onderdeel (*) behelst wat gedaan moet worden als we ontdekken dat een eerder pad naar een *Gesloten* knoop $k_2$ minder goed is dan het pad via de knoop $k_1$. Het oude pad en de lengte daarvan mogen we 'vergeten', maar het is de vraag hoe we dat het best kunnen aanpakken. Omdat $k_2$ immers in het verleden al geëxpandeerd is, hebben de opvolgers die daarbij gegenereerd zijn een $g$- en een $f$-waarde die gebaseerd is op de oude en inmiddels verbeterde $g$- en $f$-waarde van $k_2$. De $g$- en $f$-waardes van die opvolgers kloppen dus waarschijnlijk ook niet meer, zodat ze moeten worden aangepast. Hetzelfde geldt voor de eventuele opvolgers van de opvolgers, enz. Een voorbeeld van zo'n situatie is geschetst in Fig. 3.1. De dubbele cirkel in Fig. 3.1(a) houdt in dat de betreffende knoop op dat moment in *OPEN* staat. Deze notatie zullen we ook in de rest van dit verslag gebruiken. Wanneer een eindknoop ook *Open* is, zullen we die met een dubbel vierkant aangeven.

Er zijn verschillende methodes om de invloed van de niet meer correcte $g$- en $f$-waardes van al *Open* of zelfs *Gesloten* knopen te elimineren. Elke methode bepaalt een andere variant van het A*-algoritme. In de hoofdstukken 5, 6 en 7 zullen we drie van deze varianten uitgebreid behandelen.

# Hoofdstuk 4

# Correctheid en Andere Eigenschappen van het A*-Algoritme

Voordat we de verschillende varianten van het A*-algoritme gaan bekijken, zullen we in dit hoofdstuk bewijzen dat het algemene algoritme correct is; als er een pad naar het doel bestaat, levert het A*-algoritme inderdaad het kortste pad en als het doel niet bereikbaar is vanaf de startknoop $s$, komt het algoritme daar ook achter. Daarna zullen we enkele prettige eigenschappen van het A*-algoritme bespreken die gelden als de heuristische functie $h$ aan bepaalde voorwaarden voldoet.

Voor het bewijs nemen we aan dat de $g$- en de $f$-waarde van elke *Open* en elke *Gesloten* knoop overeenstemt met het beste pad dat we tot nu toe naar die knoop ontdekt hebben. Dat wil zeggen: als $(s, \ldots, Ouder(Ouder(k)), Ouder(k), k)$ het kortste pad is dat we tot een zeker moment naar een knoop $k$ gevonden hebben, dan is op datzelfde moment $f(k) = g(k) + h(k)$ met $g(k)$ de lengte van dit pad. Weliswaar hebben we gezien dat de $g$-waarde en dus ook de $f$-waarde van een knoop tijdelijk niet meer juist hoeft te zijn als er een beter pad naar één van zijn (voor)ouders is gevonden, maar we vertrouwen erop dat dit door de drie nog uit te werken methodes wordt gecorrigeerd.

Om te beginnen merken we op dat het algoritme nooit paden met cykels zal opbouwen. We beginnen immers met één extreem kort pad: $(s)$, zonder cykels. Verder worden alle paden tak voor tak geconstrueerd. Als we nu een pad met een cykel zouden hebben, moet die cykel ooit eens gesloten zijn met een tak $(k_1, k_2)$. Op dat moment creëren we een pad met een cykel naar $k_2$, terwijl we eerder al datzelfde pad zonder die cykel zijn tegengekomen. Vanwege de niet-negatieve gewichten van de takken kan het nieuwe pad met de cykel niet korter zijn dan het oude pad zonder de cykel naar $k_2$. Het nieuwe pad zal dus niet door het algoritme geaccepteerd worden.

## 4.1 Eindigheid van het Algoritme

We tonen nu eerst aan dat het algoritme voor een graaf met een eindig aantal knopen $N$ altijd eindigt. We kijken daartoe naar de lijst *OPEN*.

Er zijn maar twee redenen om een knoop aan *OPEN* toe te voegen. De eerste is dat de knoop *Onbekend* was, maar dat er nu toch een pad naar de knoop gevonden is. Dit

kan hoogstens $N-1$ keer: voor elke knoop behalve $s$ één keer.[1] De tweede reden is dat we voor een eerder ontdekte knoop een beter pad gevonden hebben dan zijn huidige beste pad. In een graaf met een eindig aantal knopen $N$, is er echter ook maar een eindig aantal verschillende paden zonder cykels vanaf een startknoop $s$ naar een knoop $k$ (welgeteld maximaal $\sum_{i=0}^{N-2} \left( \begin{array}{c} N-2 \\ i \end{array} \right) i! \approx (N-2)! \times e$). We kunnen daarom ook maar eindig vaak een beter pad naar een knoop $k$ vinden. Elke knoop $k$ kan dus maar eindig veel keren in *OPEN* gezet worden.

Omdat het algoritme begint met een eindig kleine lijst ($OPEN = [s]$), kan er ook maar eindig vaak een element uit gehaald worden. Elke keer dat het algoritme de WHILE-lus doorloopt, wordt er echter een knoop (de voorste) uit *OPEN* gehaald. De WHILE-lus kan dus maar een eindig aantal keren doorlopen worden. Omdat het inwendige van de lus ook een eindige tijd vergt en er buiten de lus alleen een korte initialisatiestap wordt uitgevoerd, moet het algoritme wel eindigen.

## 4.2 Geen Pad

Als er geen pad bestaat van de startknoop $s$ naar het doel, dient het algoritme te eindigen met een lege lijst *OPEN* en $Eind = 0$. Bij het expanderen van een knoop $k_1$ genereren we alleen maar kinderen die via een tak in de graaf vanuit $k_1$ bereikbaar zijn. Wanneer er geen pad, dat wil zeggen: geen serie aaneensluitende takken van $s$ naar een eindknoop bestaat, kunnen we die takken logischerwijs ook niet gebruiken om uiteindelijk een eindknoop te genereren. Er zal dus nooit een eindknoop in *OPEN* gezet worden, laat staan dat er ooit één uitgehaald wordt. Omdat we weten dat het algoritme wel altijd eindigt, is de enige mogelijkheid dat de lijst *OPEN* uitgeput raakt.

## 4.3 Een Pad

Als er een pad van de startknoop $s$ naar het doel bestaat, mag het algoritme niet eindigen zonder een eindknoop gevonden te hebben, dus met een lege lijst *OPEN* en $Eind = 0$. In deze paragraaf zullen we laten zien dat dat inderdaad niet gebeurt.

We beschouwen een pad van $s$ naar een eindknoop $k^*$. Dat mag een willekeurig pad zijn; het hoeft niet per se het kortste pad te zijn. Wel kunnen we het pad altijd zo kiezen dat er geen cykels in zitten.

Dan bevindt zich, zolang de eindknoop $k^*$ nog niet uit *OPEN* gehaald is (en we het pad officieel ontdekt hebben), op elk tijdstip minstens één knoop van het pad in *OPEN*. Deze uitspraak bewijzen we met inductie naar het aantal stappen dat in het A*-algoritme wordt uitgevoerd, dat wil zeggen: het aantal keer dat de WHILE-lus doorlopen wordt.

Aan het begin van het algoritme is *OPEN* gelijk aan $[s]$ en omdat $s$ uiteraard op het bewuste pad ligt, is het gestelde waar in de uitgangssituatie.

Stel nu dat het na een willekeurig aantal stappen van het A*-algoritme klopt (de inductiehypothese) en laat $k'$ de laagst op het pad gelegen (dat wil zeggen: de op het pad dichtst bij $k^*$ gelegen) knoop in *OPEN* zijn. Dan zijn alle knopen beneden $k'$ op

[1] Dit lijkt misschien in tegenspraak met wat we in paragraaf 5.2 zullen zien: dat knopen die al *Open* of *Gesloten* waren, weer *Onbekend* gemaakt worden. In dat geval is er echter geen sprake van echte onbekendheid. Een knoop is dan al wel eens ontdekt, maar er is inmiddels een beter pad naar die knoop gevonden.

het pad nog *Onbekend*. Vanwege de keuze van $k'$ zijn ze immers niet *Open*, terwijl ze ook niet *Gesloten* kunnen zijn. Een *Gesloten* knoop is namelijk al ontwikkeld, zodat ook de knoop eronder op het pad bekend en dus *Gesloten* moet zijn, zodat ook de knoop daaronder al *Gesloten* moet zijn, enz.

In de volgende stap van het algoritme wordt nu een knoop $k_1$ uit *OPEN* gehaald en geëxpandeerd. Als deze knoop niet gelijk is aan $k'$, blijft $k'$ in *OPEN* staan en geldt het gestelde ook na afloop van de stap. Indien $k_1 = k'$, zijn er twee mogelijkheden: $k_1 = k' = k^*$ of $k_1 = k' \neq k^*$. In het eerste geval eindigt het algoritme omdat het de eindknoop $k^*$ uit *OPEN* gehaald heeft. In het andere geval verdwijnt $k'$ uit *OPEN*, maar zal zijn opvolger $k_2$ op het pad naar $k^*$ als kind van $k_1$ aan *OPEN* worden toegevoegd. Die opvolger was immers *Onbekend*, had daarom een oneindig grote waarde van $f$ en krijgt nu een eindige waarde: $g(k_1) + c(k_1, k_2) + h(k_2)$. Het gevolg hiervan is dat ook in dit geval na de stap nog een knoop van het pad in *OPEN* staat. De inductiestap is dus gelukt.

We kunnen derhalve concluderen dat àls de lijst *OPEN* leeg is, de eindknoop $k^*$ uit *OPEN* gehaald en *Gesloten* moet zijn. Het algoritme eindigt dus niet zonder een pad naar het doel gevonden te hebben. Omdat het wel eindigt, vindt het algoritme wel een pad. Het kan evenwel met een ander pad op de proppen komen dan hetgeen wij beschouwd hebben.

## 4.4 Het Kortste Pad

We hebben aangetoond dat als er een pad bestaat van de startknoop $s$ naar het doel, het A*-algoritme dan zo'n pad weet te vinden. Om het correctheidsbewijs van het A*-algoritme te voltooien, moeten we nu nog laten zien dat dan inderdaad het kortste pad wordt gevonden.

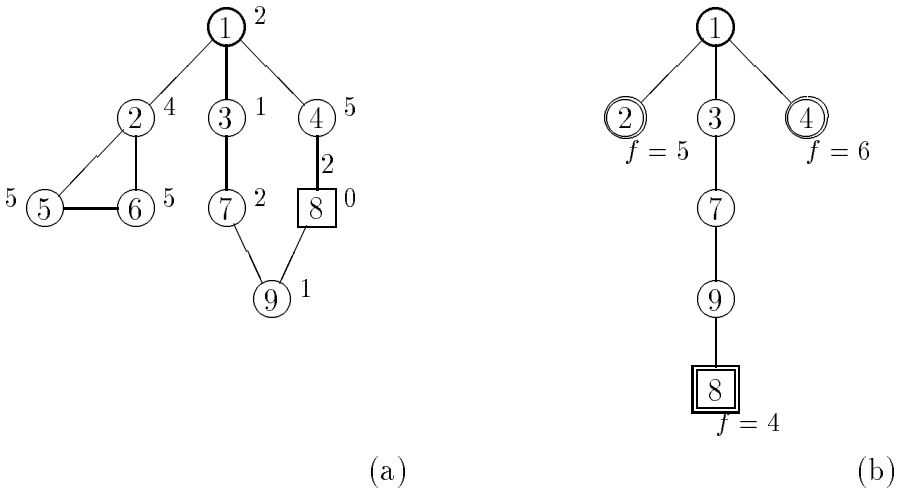
Hier wordt het van belang dat de heuristische functie $h$ een onderschatter is van de functie $h^*$ die de lengte van het kortste pad van een knoop $k$ naar het doel meet. Eerst zullen we demonstreren dat we deze eis niet zonder meer kunnen laten vallen. We doen dit aan de hand van een voorbeeld waarin $h$ een overschatter van $h^*$ is. Het A*-algoritme verwordt dan tot het A-algoritme.

### 4.4.1 Een Graaf met een Overschatter $h$

Het voorbeeld dat we gebruiken (Fig. 4.1(a)) wijkt op een paar punten af van ons voorbeeld uit paragraaf 2.3. De tak van knoop 4 naar knoop 8 heeft een gewicht 2 in plaats van 1 gekregen, de overige takken hebben nog steeds gewicht 1. Verder is van verscheidene knopen de heuristische waarde van $h$ veranderd. Hierdoor is knoop 4 de enige knoop waarvoor $h$ $h^*$ overschat, maar dat is voldoende om het fout te laten lopen. Nadat er 4 knopen zijn ontwikkeld, hebben we de zoekboom van Fig. 4.1(b) verkregen.

Omdat $h$ de kosten vanaf knoop 4 zwaar overschat, vond en vindt het algoritme het niet de moeite waard om die knoop te expanderen. Eindknoop 8 staat nu vóór in *OPEN* en wordt daar als eerste uitgehaald. Het algoritme eindigt dan en geeft ten onrechte $(1, 3, 7, 9, 8)$ als kortste pad van knoop 1 naar knoop 8. Het werkelijk kortste pad $(1, 4, 8)$ ontdekken we niet.

Om met een overschatter als heuristische functie toch een gegarandeerd kortste pad te vinden, zullen we langer door moeten gaan met het ontwikkelen van *Open* knopen.




Figuur 4.1: (a) Voorbeeld van een graaf met een heuristische functie $h$ die $h^*$ in één knoop overschat; (b) de bijbehorende zoekboom voor het A-algoritme; het kortste pad wordt niet gevonden

We kunnen het algoritme dan pas beëindigen, wanneer alle knopen die nog in *OPEN* staan, een $g$-waarde hebben die minstens zo hoog is als de lengte van een al gevonden pad naar een eindknoop.

### 4.4.2 De Functie $h$ is wel een Onderschatter

We zullen nu bewijzen dat als de heuristische functie $h$ wel een onderschatter van $h^*$ is, het algoritme wel het (een) kortste pad van de startknoop naar het doel vindt. We nemen dus aan dat, zoals het hoort bij het A*-algoritme, $0 \leq h(k) \leq h^*(k)$ voor elke knoop $k$.

Laat nu $P_0 = (k_{i_0} = s, k_{i_1}, k_{i_2}, \ldots, k_{i_m})$ een pad met minimale lengte $L_0$ zijn en veronderstel dat het algoritme een pad $P_1 = (k_{j_0} = s, k_{j_1}, k_{j_2}, \ldots, k_{j_n} = k^*)$ oplevert, dat geen kortste pad is. Dan is $f(k^*) = g(k^*)+h(k^*) = g(k^*) =$ de lengte van dit pad $> L_0$.

We beschouwen nu de toestand van het algoritme één stap voordat het eindigt. Op dat moment staat $k^*$ vóór in de lijst *OPEN*. Als hij uit *OPEN* gehaald wordt, zal het algoritme deze knoop als eindknoop herkennen en stoppen.

Op ditzelfde moment is een deel van het kortste pad $P_0$ bekend en met behulp van de relatie *Ouder* te reconstrueren. Laat $k_{i_l}$ de laatste knoop (de knoop met de grootste waarde voor $l$) van het kortste pad $P_0$ zijn die niet meer *Onbekend* is en waarvoor $g(k_{i_l}) = g^*(k_{i_l})$. Omdat $g(k_{i_0}) = g(s) = 0 = g^*(s) = g^*(k_{i_0})$, bestaat zo'n knoop. Dan geldt $f(k_{i_l}) = g(k_{i_l}) + h(k_{i_l}) = g^*(k_{i_l}) + h(k_{i_l}) \leq g^*(k_{i_l}) + h^*(k_{i_l}) = L_0 <$ de lengte van het pad $P_1 = f(k^*)$.

Omdat de knoop $k_{i_l}$ niet meer *Onbekend* is, is hij *Open* of *Gesloten*. Als hij *Open* zou zijn, zou hij, omdat $f(k_{i_l}) < f(k^*)$, vóór $k^*$ in de lijst *OPEN* staan. Dat kan echter niet, omdat $k^*$ vooraan staat. Dus moet $k_{i_l}$ *Gesloten*, ofwel al ontwikkeld zijn.

Indien $l$ gelijk zou zijn aan $m$, zou $k_{i_l}$ een eindknoop zijn. Omdat het algoritme eindigt wanneer het een eindknoop uit *OPEN* gehaald heeft, zou het dan nooit de huidige toestand bereikt kunnen hebben. Daarom moet $l < m$ zijn.

Bij het ontwikkelen van $k_{i_l}$ is dan ook de knoop $k_{i_{l+1}}$ gegenereerd. Maar dan moet gelden dat $g(k_{i_{l+1}}) \leq g(k_{i_l}) + c(k_{i_l}, k_{i_{l+1}}) = g^*(k_{i_l}) + c(k_{i_l}, k_{i_{l+1}}) = g^*(k_{i_{l+1}})$, want de tak $(k_{i_l}, k_{i_{l+1}})$ maakt deel uit van het kortste pad $P_0$. Omdat per definitie ook $g^*(k_{i_{l+1}}) \leq g(k_{i_{l+1}})$, is dan $g(k_{i_{l+1}}) = g^*(k_{i_{l+1}})$.

Dit is in tegenspraak met de keuze van $l$, zodat onze aanname verworpen moet worden. Het algoritme produceert derhalve wel een kortste pad.

## 4.5 Speciale Heuristische Functies $h$

Als de heuristische functie $h$ aan zekere voorwaarden voldoet, blijkt het A*-algoritme bepaalde prettige eigenschappen te hebben. In [Nilsson1982] wordt aangetoond dat als $h$ monotoon is, het A*-algoritme nooit een knoop zal expanderen als het nog niet een optimaal pad van de startknoop $s$ naar die knoop heeft gevonden. Het gevolg hiervan is dat we nooit met het probleem zitten dat we een betere $g$- en $f$-waarde vinden voor een al *Gesloten* knoop. Dan hoeven we ons dus ook geen zorgen te maken over de eventuele invloed van de incorrecte $g$- en $f$-waardes van de opvolgers van zo'n knoop op het eindresultaat van het algoritme.

Wij zullen in deze paragraaf ingaan op de gevolgen van een nog strengere voorwaarde op de functie $h$. In het geval dat $h$ een perfecte schatter van $h^*$ is, dat wil zeggen als $h = h^*$, blijkt het A*-algoritme alleen optimale paden af te lopen. In het bijzonder, als er maar één kortste pad is, zal het algoritme recht op het doel aflopen. Weliswaar worden in het algemeen ook knopen gegenereerd die niet op het kortste pad liggen, maar die zullen dan niet ontwikkkeld worden. Dit alles zullen we nu bewijzen.

Allereerst merken we op dat er nooit een knoop in *OPEN* kan staan met een $f$-waarde kleiner dan de lengte van het kortste pad. Stel immers dat er wel zo'n knoop $k_1$ in *OPEN* staat. Dan is, omdat $h(k) = h^*(k)$ en $g(k) \geq g^*(k)$ voor elke knoop $k$:

$$f^*(k_1) = g^*(k_1) + h^*(k_1) \leq g(k_1) + h(k_1) = f(k_1) < L_0 \leq f^*(k_1) \tag{4.1}$$

Dit is natuurlijk onmogelijk. Voor elke knoop $k_1$ in *OPEN* moet $f(k_1)$ dus minstens zo groot zijn als $L_0$.

Laten we nu een knoop $k_1$ in *OPEN* beschouwen waarvoor $f(k_1)$ gelijk is aan de lengte van het kortste pad. Dan gaat Vgl. 4.1 over in

$$f^*(k_1) = g^*(k_1) + h^*(k_1) \leq g(k_1) + h(k_1) = f(k_1) = L_0 \leq f^*(k_1) \tag{4.2}$$

Vanwege de buitenkanten moet overal in deze expressie gelijkheid gelden. Knoop $k_1$ ligt dus op een kortste pad van $s$ naar het doel en, omdat $g(k_1) = g^*(k_1)$, hebben we al een kortste pad naar $k_1$ gevonden.

Na dit voorbereidende werk hoeven we alleen nog maar aan te tonen dat er, zolang er nog geen oplossing gevonden is, op elk moment in het algoritme een knoop $k_1$ in *OPEN* staat waarvoor $f(k_1) = L_0$. Dit doen we met inductie naar het aantal stappen dat het algoritme tot een bepaald moment heeft uitgevoerd.

Na 0 stappen hebben we alleen nog maar de initialisatie achter de rug. *OPEN* bevat dan alleen de startknoop $s$ en gelukkig is $f(s) = g(s)+h(s) = 0+h(s) = g^*(s)+h^*(s) = f^*(s) = L_0$. De beginstap van het inductiebewijs is dus geslaagd.

Veronderstel nu dat voor zekere $i \leq 1$ tot en met de $i-1$-ste stap steeds een element in *OPEN* heeft gestaan met een $f$-waarde gelijk aan $L_0$ (inductiehypothese) en dat er nog geen eindknoop is gevonden. Dan geldt als gevolg van onze voorbereidende opmerkingen dat elke tot nu toe geëxpandeerde knoop een $f$-waarde had (en nog heeft)

van $L_0$ en dat ook het element dat nu vooraan in *OPEN* staat die $f$-waarde heeft. We gaan nu de $i$-de stap in het algoritme uitvoeren. We halen de knoop $k_1$ met de laagste $f$-waarde ($L_0$) uit *OPEN*. We weten dan dat $k_1$ op een kortste pad $P_0$ van $s$ naar het doel ligt en dat we het deel van dat kortste pad tot en met $k_1$ al hebben ontdekt.

Nu willen we $k_1$ expanderen. Indien $k_1$ een eindknoop is, zal het algoritme eindigen. Het gestelde is dan triviaal waar, omdat er wel een oplossing is gevonden. Stel dus dat $k_1$ geen eindknoop is. Dan gaan we de opvolgers van $k_1$ genereren. Laat $k_2$ het kind van $k_1$ op het pad $P_0$ zijn. Dan wordt ook $k_2$ gegenereerd met een nieuwe $g$-waarde die gelijk is aan $g^*(k_2)$, want we zitten op een kortste pad van $s$ via $k_2$ naar het doel. De $h$-waarde van $k_2$ is uiteraard gelijk aan $h^*(k_2)$, zodat de $f$-waarde van de nieuw gegenereerde knoop $k_2$ gelijk is aan $g^*(k_2) + h^*(k_2) = f^*(k_2) = L_0$. Er zijn nu drie mogelijkheden:

1. Knoop $k_2$ was nog *Onbekend*. Dan wordt $k_2$ nu in *OPEN* gezet met $f(k_2) = L_0$.

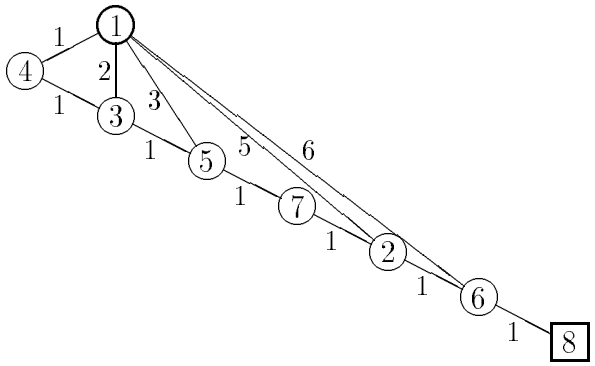
2. Knoop $k_2$ was al *Open*. Dan had $k_2$ een $f$-waarde die minstens zo groot was als $L_0$. Als die waarde al gelijk was aan $L_0$, blijft $k_2$ met de oude waarde in *OPEN* staan. Als die waarde groter was dan $L_0$, wordt $k_2$ nu met de nieuwe waarde $L_0$ in *OPEN* geplaatst.

3. Knoop $k_2$ was al *Gesloten*. Kennelijk hadden we $k_2$ al via een ander kortste pad ontdekt en vervolgens geëxpandeerd voordat we aan $k_1$ toekwamen. We beschouwen nu het (kortste) pad $P_0 = (k_{i_0} = s, k_{i_1}, \ldots, k_{i_p} = k_1, k_{i_{p+1}} = k_2, \ldots, k_{i_l}, k_{i_{l+1}}, \ldots, k_{i_m} = k^*)$, waarbij $k_{i_l}$ de laagst op dit pad gelegen knoop is die al *Gesloten* is. Dan is $l \geq p+1$ en, omdat het algoritme nog niet geëindigd was, $l < m$. Net als alle andere geëxpandeerde knopen had $k_{i_l}$ op het moment dat hij geëxpandeerd werd, een $f$-waarde die gelijk was aan $L_0$. We hadden dus al een kortste pad van $s$ naar $k_{i_l}$ gevonden. Wanneer we dat pad doortrekken naar $k_{i_{l+1}}$, hebben we een kortste pad naar die knoop. Omdat $k_{i_{l+1}}$ destijds bij het expanderen van $k_{i_l}$ is gegenereerd, staat het op dit moment met $g(k_{i_{l+1}}) = g^*(k_{i_{l+1}})$ in *OPEN*. Maar dan is ook $f(k_{i_{l+1}}) = g(k_{i_{l+1}}) + h(k_{i_{l+1}}) = g^*(k_{i_{l+1}}) + h^*(k_{i_{l+1}}) = f^*(k_{i_{l+1}}) = L_0$, want $k_{i_{l+1}}$ ligt op het kortste pad $P_0$. Ook na het ontwikkelen van $k_1$ staat $k_{i_{l+1}}$ nog met deze waarde voor $f$ in *OPEN*.

In alle drie gevallen kunnen we concluderen dat er na het ontwikkelen van $k_1$ weer of nog steeds een knoop in *OPEN* staat met als $f$-waarde $L_0$. Zolang we het doel nog niet bereikt hebben, kunnen we dus steeds een knoop expanderen die op een kortste pad van $s$ naar het doel ligt en die we bovendien via takken uit dat pad bereikt hebben. Het A*-algoritme bewandelt dus alleen kortste paden naar het doel.

**Terzijde** Afhankelijk van de nummering van de knopen kunnen voor bepaalde instanties al de drie in het inductiebewijs genoemde mogelijkheden zich inderdaad voordoen. Dit illustreren we met het voorbeeld in Fig. 4.2. We nemen daarbij aan dat gelijktijdig gegenereerde kinderen met dezelfde $f$-waarde in de volgorde van hun nummering in *OPEN* komen te staan.

Nadat de startknoop is geëxpandeerd, is *OPEN* gelijk aan $[2, 3, 4, 5, 6]$. Het ontwikkelen van knoop 2 levert geen betere waarde voor de *Open* knoop 6 op. Als vervolgens knoop 3 wordt ontwikkeld, wordt geen betere waarde voor de *Open* knoop 5 en al helemaal niet voor de eveneens *Open* knoop 4 gevonden. Bij het ontwikkelen van knoop 4 blijkt de gegenereerde opvolger 3 zelfs al *Gesloten* te zijn.




Figuur 4.2: Voor deze graaf met een perfecte schatter $h$ kunnen bij het ontwikkelen van de knopen in het A*-algoritme zowel *Onbekende* als *Open* en *Gesloten* opvolgers gegenereerd worden.

# Hoofdstuk 5

# Varianten van het A*-Algoritme

Zoals we in paragraaf 3.2 hebben opgemerkt, zijn er verschillende varianten van het A*-algoritme. In dit hoofdstuk zullen we er drie beschrijven.

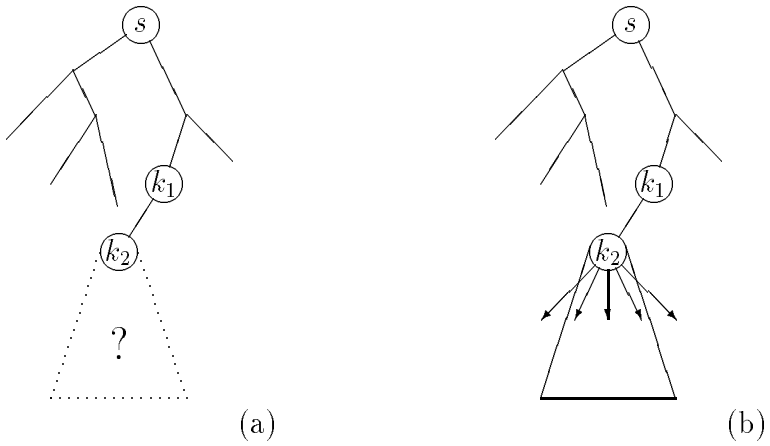
In de eerste variant, A*-Propageer, wordt een kortere padlengte naar een al *Gesloten* knoop gepropageerd naar al zijn nakomelingen (kinderen, kleinkinderen, enz.). De tweede variant, A*-Verwijder, kenmerkt zich door het expliciet tijdelijk verwijderen uit de zoekboom van toestanden met een onjuiste padlengte. In de derde variant, A*-Vertrouw, vinden we het niet nodig om expliciet onjuistheden te corrigeren en vertrouwen we erop dat dit automatisch gebeurt.

## 5.1 Propageren van de Juiste Padlengte

De eerste van de drie varianten van het A*-algoritme die we zullen bespreken, sluit het best aan bij de beschrijving van het algoritme zoals we die in hoofdstuk 3 en hoofdstuk 4 hebben gegeven. We gingen er daar vanuit dat elke knoop uit de graaf hooguit één keer in *OPEN* wordt gezet en dat voor elke knoop in *OPEN* de $g$-waarde gelijk is aan de lengte van het pad van $s$ naar die knoop zoals we dat via de *Ouder*-relatie hebben opgeslagen. Met variant A*-Propageer is dit inderdaad het geval. Wanneer we op een gegeven moment een beter pad naar een al eerder geëxpandeerde knoop $k_2$ ontdekken, wordt die verbetering ook aan zijn kinderen doorgegeven, met andere woorden: naar zijn kinderen gepropageerd (zie Fig. 5.1). Deze variant is ook uitgewerkt in [Rich1983].

Het is logisch dat de verbetering ook voor alle nakomelingen van $k_2$ in de zoekboom van het algoritme van toepassing is. Het beste pad dat we tot nu toe naar zo'n nakomeling hadden gevonden, liep immers via $k_2$. Wanneer we nu voor het eerste gedeelte van het pad (tot en met $k_2$) een korter alternatief ontdekt hebben, wordt het hele kortste pad naar die nakomeling korter.

We kunnen ons echter niet beperken tot de nakomelingen van $k_2$ in de zoekboom. Het voorbeeld in Fig. 5.2 is daarvan een bewijs. Na twee stappen in het algoritme hebben we de zoekboom in Fig. 5.2(b) gecreëerd en is *OPEN* gelijk aan $[3, 4, 5]$. Bij het ontwikkelen van knoop 3 vinden we voor geen van de buurknopen 1, 4 en 5 een korter pad. De zoekboom verandert dus nauwelijks (zie Fig. 5.2(c)). Vervolgens wordt knoop 4 geëxpandeerd en er wordt een verbetering voor de al *Gesloten* knoop 3 ontdekt. Omdat knoop 3 geen nakomelingen in de zoekboom heeft, zou deze verbetering aan geen enkele andere knoop worden doorgegeven. We krijgen dan dus de zoekboom uit Fig. 5.2(d). Nu bevindt alleen knoop 5 zich nog in *OPEN*. Die wordt er uit gehaald, als eindknoop herkend, waarna het pad $(1, 2, 5)$ met lengte 7 als het kortste pad van




Figuur 5.1: (a) De zoekboom juist nadat het betere pad van de startknoop $s$ naar knoop $k_2$ is ontdekt; (b) bij A*-Propageer wordt de verbetering naar alle opvolgers van $k_2$ gepropageerd, vervolgens naar de opvolgers van de opvolgers van $k_2$, enz.

knoop 1 naar knoop 5 wordt opgeleverd. Dit is niet terecht, want het pad $(1, 4, 3, 5)$ is met een lengte 6 echt korter.

Om zulke fouten te voorkomen, zullen we dus ook eventuele nakomelingen van knoop $k_2$ (in het voorbeeld knoop 3) die niet als zodanig in de zoekboom zijn opgenomen, moeten bekijken. Tot nu toe waren ze bereikbaar vanaf de startknoop over een pad dat korter was dan een pad via $k_2$. Zoals we in het voorbeeld gezien hebben, is het echter heel goed mogelijk dat we, dankzij de verbetering van het pad naar $k_2$ nu ook een beter pad naar zo'n nakomeling hebben ontdekt.

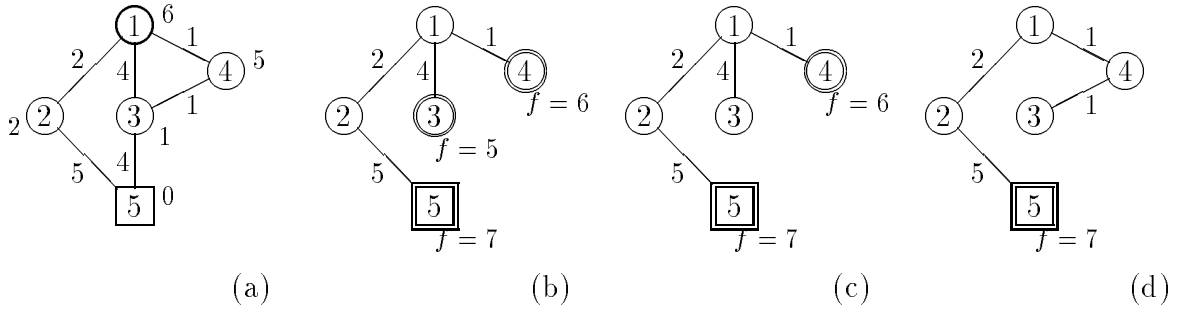
Daarom lopen we in principe alle knopen af die geopend zijn (geweest) en die over een pad van (ooit) geopende knopen vanaf knoop $k_2$ bereikbaar zijn. Het heeft uiteraard geen zin om ook *Onbekende* knopen te controleren, omdat we voor hen nog helemaal geen beste-padlengte hebben opgeslagen. Er valt dan dus ook geen verbeterde waarde te propageren. Als we bij het propageren bij een *Open* knoop aankomen, hoeven we de opvolgers van die knoop niet te onderzoeken. Omdat die knoop nog in *OPEN* staat, kan hij in de toekomst nog geëxpandeerd worden. De verbetering wordt dan vanzelf doorgegeven aan de kinderen die daarbij gegenereerd worden.

Een andere reden om niet dieper te gaan met het propageren van de verbetering van het pad naar $k_2$ is wanneer die verbetering geen verbetering van het pad naar een wel via $k_2$ bereikbare knoop $k_3$ oplevert. Dan hadden we kennelijk al een pad $P$ naar $k_3$ dat korter was dan het pad via $k_2$ en dat pad is ook korter dan het verbeterde pad via $k_2$. Alle knopen die via $k_3$ bereikbaar zijn, zijn dan nog steeds beter bereikbaar met behulp van het pad $P$ dan met behulp van het pad via $k_2$ en $k_3$. De gevonden verbetering is dus ook voor die knopen niet van belang.

Aldus wordt de propageerstap gegeven door het volgende (sub)algoritme

PROCEDURE*Propageer*($k_2$);

FOR alle knopen $k_3$ die in 1 stap vanuit $k_2$ bereikbaar zijn DO
$gg := g[k_2] + c[k_2, k_3]$;
$ff := gg + h[k_3]$;




Figuur 5.2: (a) Voorbeeldgraaf voor het A*-algoritme; (b) de zoekboom na twee stappen in het A*-algoritme; (c) de zoekboom nadat ook knoop 3 is geëxpandeerd; (d) de zoekboom als de verbetering voor knoop 3 die bij het ontwikkelen van knoop 4 wordt gevonden, alleen naar de (0) nakomelingen van knoop 3 in de zoekboom wordt gepropageerd.

```
IF ff < f[k3] THEN        (* we hebben een beter pad naar k3 gevonden *)
   g[k3] := gg;
   f[k3] := ff;
   IF Ouder[k3] <> k2 THEN
      Ouder[k3] := k2;
   IF Status[k3] = Open THEN
      haal k3 uit de lijst OPEN;        (* met zijn oude f-waarde *)
      voeg k3 toe aan de lijst OPEN
         zo dat OPEN gesorteerd blijft op f-waarde
   ELSE       (* kennelijk was k3 Gesloten *)
      Propageer(k3);
```

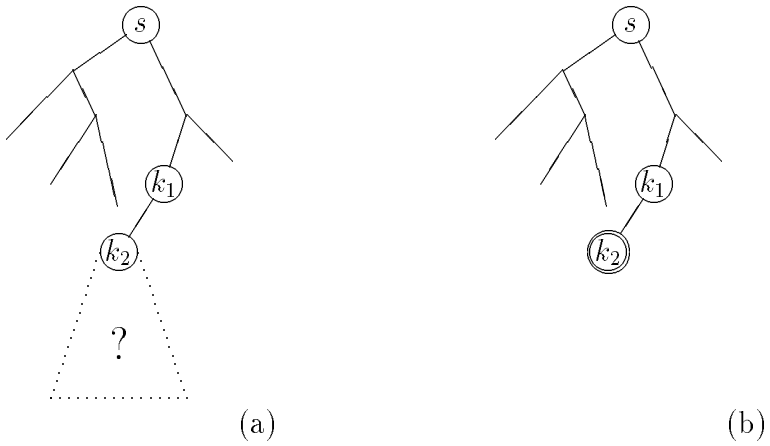
Bij (*) in het algoritme in paragraaf 3.2 kunnen we dan volstaan met de aanroep

*Propageer*($k_2$);

## 5.2 Weer Verwijderen van Knopen uit de Zoekboom

Wanneer we een korter pad naar een al geëxpandeerde knoop $k_2$ ontdekken dan het pad dat we tot nu toe geregistreerd hadden, kunnen we dat ook als volgt opvatten: elke knoop $k_3$ die we tot nu toe het best konden bereiken over een pad via $k_2$, heeft nu $g$- en $f$-waardes die niet meer geldig zijn. Om te voorkomen dat we met die ongeldige waardes verder gaan werken, kunnen we zo'n knoop $k_3$ maar net zo goed helemaal uit de zoekboom verwijderen.

Dit is precies wat er in de tweede variant van het A*-algoritme, A*-Verwijder, gebeurt. Deze variant wordt onder andere in [Nau,Kumar,Kanal1984] beschreven. Alle knopen die onder $k_2$ in de zoekboom hangen worden daaruit verwijderd. Terwijl ze al geopend of zelfs *Gesloten* waren, worden ze nu weer *Onbekend* gemaakt. Ze staan dan in principe weer open voor elke knoop die hen als opvolger genereert. De knoop $k_2$ zelf wordt met zijn verbeterde waardes weer in *OPEN* gezet. In Fig. 5.3 is dit alles




Figuur 5.3: (a) De zoekboom juist nadat het betere pad van de startknoop $s$ naar knoop $k_2$ is ontdekt; (b) bij A*-Verwijder wordt $k_2$ opnieuw in *OPEN* gezet en al zijn nakomelingen worden met hun verkeerde $g$- en $f$-waardes uit de zoekboom verwijderd.

uitgebeeld. Als $k_2$ nu als eerste geëxpandeerd wordt, krijgen zijn opvolgers meteen met zijn nieuwe waardes te maken. Zo kan de verbetering indirect toch naar de nakomelingen van $k_2$ gepropageerd worden, ook naar diegene die zojuist uit de zoekboom zijn verwijderd.

Het is echter niet noodzakelijk dat de gevonden verbetering in het pad van $s$ naar $k_2$ zo alsnog aan elke nakomeling $k_3$ van $k_2$ wordt doorgegeven. Wellicht staat er op dit moment een knoop in *OPEN* die er nog niet in stond toen $k_2$ voor het eerst ontwikkeld werd en via welke een nòg korter pad naar $k_3$ loopt dan het verbeterde pad via $k_2$. Wanneer dat nòg kortere pad naar $k_3$ eerder wordt ontdekt dan het verbeterde pad via $k_2$, zal het laatste door $k_3$ niet meer als een verbetering worden opgevat.

Bij het weer *Onbekend* maken van de nakomelingen van $k_2$ kunnen we ons gelukkig wel beperken tot die nakomelingen die in de zoekboom van het algoritme stonden. Een situatie als in Fig. 5.2 zal nu geen probleem opleveren. Elke knoop $k_3$ die al ontwikkeld en dus *Gesloten* was, maar nu weer onbekend is gemaakt, zal immers zonodig nogmaals ontwikkeld worden. Dan worden automatisch ook de opvolgers van $k_3$ bekeken die niet als zodanig in de oude zoekboom voorkwamen. Hetzelfde geldt voor knoop $k_2$ die *Gesloten* was, maar nu weer *Open* is geworden.

Voor het voorbeeld in Fig. 5.2 houdt dit alles in dat we, als we het pad van $s = 1$ naar $k_2 = 3$ via $k_1 = 4$ hebben ontdekt, de *Gesloten* knoop 3 weer in *OPEN* plaatsen. Omdat knoop 3 een blad van de zoekboom was, hoeven we verder geen knopen uit die zoekboom te verwijderen. De lijst *OPEN* bevat nu knoop 3 en knoop 5, waarvan knoop 3 er weer als eerste uitgehaald en geëxpandeerd wordt. Hierbij wordt ook het kortere pad naar de eindknoop 5 ontdekt.

De procedure die het verwijderen en *Onbekend* maken van knopen uit de zoekboom verzorgt, wordt dus:

PROCEDURE *Verwijder*($k_2$);

FOR alle kinderen $k_3$ van $k_2$ in de zoekboom DO
IF $Status[k_3] = Open$ THEN

haal $k_3$ uit de lijst *OPEN*
ELSE (* kennelijk was $k_3$ *Gesloten* *)
*Verwijder*($k_3$);
$f[k_3] := \infty$;
$Ouder[k_3] := 0$;
$Status[k_3] :=$ *Onbekend*;

In deze procedure duiken we alleen dieper de recursie in wanneer $k_3$ *Gesloten* is. Elke *Open* knoop is namelijk tevens een blad van de zoekboom, zodat hij geen nakomelingen heeft. Het heeft dan ook geen zin om hem op nakomelingen te onderzoeken.

In het algemene A*-algoritme zoals dat in paragraaf 3.2 is gegeven, moeten we nu bij (*) invullen:

*Verwijder*($k_2$);
$Status[k_2] :=$ *Open*;
voeg $k_2$ toe aan de lijst *OPEN*
zo dat *OPEN* gesorteerd blijft op $f$-waarde;

Tenslotte willen we bij dit algoritme nog één opmerking maken. Het feit dat een knoop nu weer *Onbekend* gemaakt en dus opnieuw ontdekt kan worden, houdt niet in dat het bewijs van de eindigheid van het A*-algoritme (zie paragraaf 4.1) totaal ontkracht wordt.

Een knoop $k_3$ wordt immers pas weer *Onbekend* gemaakt als voor zijn voorouder $k_2$ een pad is ontdekt dat korter is dan hetgeen we tot nu toe kenden. We hebben dan feitelijk ook een beter pad naar $k_3$ ontdekt. Omdat de verbetering bij knoop $k_2$ al daadwerkelijk is aangebracht, zullen we het oude pad niet nogmaals ontdekken. Het aantal keer dat $k_3$ *Onbekend* gemaakt wordt, is dus naar boven begrensd door het aantal verschillende paden zonder cykels van de startknoop $s$ naar $k_3$. In paragraaf 4.1 hebben we al gezien dat dit aantal eindig is, omdat de graaf eindig is.

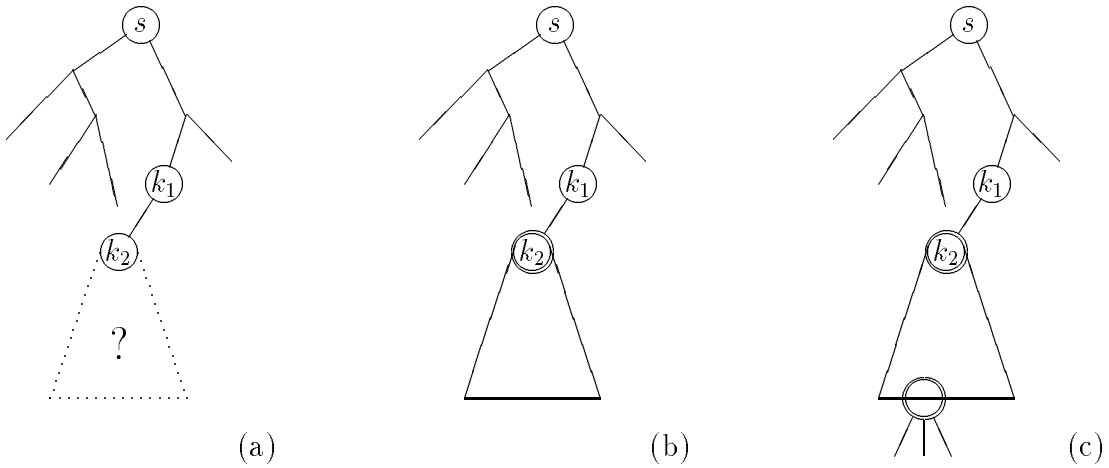
## 5.3 Vertrouwen op de Goede Afloop

De laatste variant van het A*-algoritme die we zullen bespreken, is het gemakkelijkst te implementeren. Het is daarentegen een stuk lastiger dan het was voor de andere twee varianten, om de correctheid van deze variant aan te tonen. De variant bestaat eruit dat we feitelijk niets bijzonders doen wanneer we een korter pad ontdekken naar een knoop $k_2$ die al geëxpandeerd is. We zetten $k_2$ weliswaar met zijn nieuwe, lagere $g$- en $f$-waardes opnieuw in *OPEN*, maar zijn nakomelingen laten we volkomen ongemoeid.

Deze variant is een verkorte versie van A*-Verwijder. In vergelijking met die laatste variant laten we gewoon het expliciet verwijderen van de nakomelingen van knoop $k_2$ in de zoekboom achterwege. Om deze variant van het A*-algoritme vast te leggen is het dus voldoende om bij (*) in het algoritme in paragraaf 3.2 de volgende twee regels neer te zetten:

$Status[k_2] :=$ *Open*;
voeg $k_2$ toe aan de lijst *OPEN*
zo dat *OPEN* gesorteerd blijft op $f$-waarde;

Op deze manier blijven de nakomelingen van $k_2$ in de zoekboom staan, terwijl de $g$- (en dus ook de $f$-)waardes van deze knopen niet meer overeenkomen met de lengte




Figuur 5.4: (a) De zoekboom juist nadat het betere pad van de startknoop $s$ naar knoop $k_2$ is ontdekt; (b) bij A*-Vertrouw zetten we alleen knoop $k_2$ opnieuw in *OPEN*; (c) we wensen geen situaties als deze, waarin een nakomeling van $k_2$ wordt ontwikkeld voordat zijn $g$- en $f$-waardes overeenkomen met zijn pad vanaf de startknoop $s$.

van het pad dat met de relatie *Ouder* is vastgelegd. In het bijzonder blijven er in het algemeen nakomelingen van $k_2$ met hun incorrecte $g$- en $f$-waardes in *OPEN* staan. Als deze knopen nu ontwikkeld worden voordat de verbetering van het pad naar $k_2$ tot hen is doorgedrongen, geven zij de onjuiste $g$- en $f$-waardes door aan hun kinderen. Omdat we niets aan die onjuiste waardes hebben, zouden we dan gewoon werk voor niets zitten doen.

We vertrouwen er echter op dat dit niet zal gebeuren en daarom noemen we deze variant A*-Vertrouw. In het volgende hoofdstuk zullen we bewijzen dat dit vertrouwen gerechtvaardigd is.

# Hoofdstuk 6

# Analyse van de Variant A*-Vertrouw

In dit hoofdstuk zullen we A*-Vertrouw aan een grondige analyse onderwerpen. We zullen aantonen dat er in deze variant van het A*-algoritme geen knopen ontwikkeld worden die niet ook bij de variant A*-Verwijder worden ontwikkeld. Het blijkt zelfs mogelijk dat er bij A*-Vertrouw minder knopen worden ontwikkeld dan bij A*-Verwijder voor dezelfde instantie. Tenslotte leiden we uit de vergelijking met A*-Verwijder de correctheid van A*-Vertrouw af.

Eerst zullen we echter enkele fundamentele stellingen moeten bewijzen.

## 6.1 Fundamentele Stellingen voor A*-Vertrouw

In deze paragraaf zullen we geregeld spreken van een knoop die *Open* gemaakt, *Open* gesteld, of geopend wordt. Daarmee bedoelen we dat het algoritme voor die knoop lagere $g$- en $f$-waardes heeft gevonden dan tot nu toe bekend waren, en dat de knoop met die nieuwe waardes in *OPEN* gezet wordt. Het is mogelijk dat de knoop voor het eerst in *OPEN* terecht komt. Echter, ook als hij er al eerder in gestaan heeft (of er zelfs nog in stond op het moment dat de lagere waardes werden gevonden) met een hogere $f$-waarde dan we nu ontdekt hebben, zullen we het hebben over het openen van die knoop.

We beginnen met enkele algemene observaties.

**Lemma 1** *Stel dat er in de zoekboom van A*-Vertrouw na $i \geq 1$ stappen een tak van knoop $k_1$ naar knoop $k_2$ voorkomt. Laat $g(k_1)$ en $g(k_2)$ de g-waarde voor knoop $k_1$, respectievelijk $k_2$ na $i$ stappen aanduiden. Dan is $g(k_2) \geq g(k_1) + c(k_1, k_2)$.*

**Bewijs** Laat knoop $k_2$ voor het laatst in de $j$-de stap van het algoritme geopend zijn ($1 \leq j \leq i$). Omdat $k_1$ zijn huidige ouder is, moet die in de $j$-de stap ontwikkeld zijn. Daarbij kwamen we tot de conclusie dat het pad van de startknoop $s$ via $k_1$ onmiddellijk naar $k_2$ het beste tot dan bekende pad van $s$ naar $k_2$ was. Laat $g'(k_1)$ en $g'(k_2)$ de $g$-waardes voor $k_1$, respectievelijk $k_2$ na $j$ stappen zijn. Dan stelden we in de $j$-de stap $g'(k_2)$ gelijk aan $g'(k_1) + c(k_1, k_2)$.

Sindsdien is $k_2$ niet opnieuw *Open* gemaakt, zodat de $g$-waarde van $k_2$ niet meer is gewijzigd tot en met de $i$-de stap. Er geldt dus $g(k_2) = g'(k_2)$. Het is daarentegen wel mogelijk dat er na de $j$-de stap een nog beter pad van $s$ naar $k_1$ is ontdekt. Dan is knoop $k_1$ opnieuw in *OPEN* geplaatst met een lagere waarde voor $g$. Omdat in de loop van het algoritme de $g$-waarde van een knoop nooit verhoogd wordt, is $g(k_1) \leq g'(k_1)$.

Dientengevolge is $g(k_2) = g'(k_2) = g'(k_1) + c(k_1, k_2) \geq g(k_1) + c(k_1, k_2)$. □

**Gevolg 2** *Stel dat er in de zoekboom van A*-Vertrouw na $i \geq 1$ stappen een pad van knoop $k_1$ naar knoop $k_2$ voorkomt. Dan geldt voor de g-waardes na $i$ stappen: $g(k_2) \geq g(k_1)+$ de lengte van het betreffende pad.*
**Bewijs** Volgt onmiddellijk door het herhaald toepassen van Lemma 1 voor de takken in het pad van $k_1$ naar $k_2$. □

**Gevolg 3** *Stel dat in de $i$-de stap van A*-Vertrouw ($i \geq 2$) knoop $k_2$ wordt ontwikkeld, waarbij één van zijn echte voorouders $k_1$ als opvolger wordt gegenereerd. Dan zal $k_1$ niet als kind van $k_2$ worden geaccepteerd; hij wordt uit de verzameling opvolgers van $k_2$ verwijderd. Het algoritme accepteert dus nooit een 'cykel in de zoekboom'.*
**Bewijs** Laat $g(k_1)$ en $g(k_2)$ de $g$-waardes van knoop $k_1$, respectievelijk $k_2$ na $i-1$ stappen voorstellen. Wanneer knoop $k_2$ in de $i$-de stap zijn voorouder $k_1$ als opvolger genereert, wordt (in feite) $g(k_2) + c(k_2, k_1)$ vergeleken met $g(k_1)$. Dankzij Gevolg 2 weten we dat $g(k_2) + c(k_2, k_1) \geq g(k_2) \geq g(k_1)+$ de lengte van het pad van $k_1$ naar $k_2 \geq g(k_1)$. Knoop $k_1$ zal daarom het aanbod van zijn nakomeling $k_2$ om ook als zijn ouder op te treden afwijzen; dat aanbod zou $k_1$ immers geen lagere $g$- en $f$-waardes opleveren. □

We zullen nu twee stellingen bewijzen over de aanwezigheid in het verleden van *Open* knopen op een huidig pad naar een momenteel *Open* knoop. Wanneer we het in dat verband hebben over de knopen op een pad in de zoekboom van knoop $k_0$ naar knoop $k_2$, sluiten we $k_2$ daarbij in, maar $k_0$ *niet*.

**Stelling 4** *Stel dat er na $i \geq 1$ stappen in A*-Vertrouw een Open knoop $k_2$ is. Dan heeft er op het huidige (!) pad van de startknoop $s$ naar $k_2$ na elke stap $j = 1, 2, \ldots, i$ minstens één Open knoop gestaan met dezelfde $g$- en $f$-waardes als na $i$ stappen.*
**Bewijs** Gaat met inductie naar het aantal stappen $i$. Merk allereerst op dat $k_2$ niet gelijk kan zijn aan de startknoop $s$ zelf. Die is immers alleen aan het begin, na 0 stappen *Open* en daarna niet meer.

Als we na $i = 1$ stappen een *Open* knoop $k_2$ hebben, is de enige waarde voor $j$ die we moeten controleren $j = 1$. Na $j = 1$ stap hebben we $k_2$ in *OPEN* staan met dezelfde $g$- en $f$-waardes als na $i = 1$ stap. Uiteraard ligt knoop $k_2$ op het pad van $s$ naar $k_2$.

Stel dat het gestelde waar is na $i-1$ stappen voor zekere $i \geq 2$ (inductiehypothese), en dat we na $i$ stappen een *Open* knoop $k_2$ hebben. Na $j = i$ stappen hebben we dan een knoop op het huidige pad van $s$ naar $k_2$ die *Open* is en dezelfde $g$- en $f$-waardes heeft als na $i$ stappen, namelijk $k_2$ zelf. Het is nu nog te bewijzen dat we ook na $j = 1, 2, \ldots i-1$ stappen op het huidige pad van $s$ naar $k_2$ steeds een *Open* knoop hebben gehad met dezelfde $g$- en $f$-waardes als na $i$ stappen. We beschouwen daartoe de knoop $k_1$ die in stap $i$ is geëxpandeerd. Omdat de startknoop $s$ alleen in de eerste stap wordt ontwikkeld, is $k_1 \neq s$. We onderscheiden twee gevallen:

- De knoop $k_1$ ligt niet op het huidige pad van $s$ naar $k_2$. Dan was $k_2$ na $i-1$ stappen al *Open* en was het vorige pad van $s$ naar $k_2$ (het pad na $i-1$ stappen) gelijk aan het huidige pad van $s$ naar $k_2$.

  Volgens de inductiehypothese hadden we na $j = 1, 2, \ldots, i-1$ stappen op het vorige (= huidige) pad van $s$ naar $k_2$ steeds een *Open* knoop met dezelfde $g$- en $f$-waardes als na $i-1$ stappen. Omdat $k_1$ niet op het huidige pad van $s$ naar $k_2$ ligt en ook niet gelijk is aan $s$ zelf, zijn die $g$- en $f$-waardes ook gelijk aan de $g$- en $f$-waardes na $i$ stappen.

- De knoop $k_1$ ligt wel op het huidige pad van $s$ naar $k_2$. Dan bekijken we het deel van dat pad tot en met $k_1$: het huidige pad van $s$ naar $k_1$ ($\neq s$). Omdat cykels altijd niet-negatieve lengtes hebben (zie Gevolg 3), is dit deel gelijk aan het pad van $s$ naar $k_1$ na $i-1$ stappen. Na $i-1$ stappen was knoop $k_1$ kennelijk *Open*, want anders had hij in de $i$-de stap niet geëxpandeerd kunnen worden.

  Volgens de inductiehypothese is er na $j = 1, 2, \ldots, i-1$ stappen op het vorige (= huidige) pad van $s$ naar $k_1$ steeds een *Open* knoop geweest met dezelfde $g$- en $f$-waardes als na $i-1$ stappen. Die $g$- en $f$-waardes zijn in de $i$-de stap door het ontwikkelen van $k_1$ niet gewijzigd. Op het huidige pad van $s$ naar $k_1$ (en dan zeker op het huidige pad van $s$ naar $k_2$ is er dus na $j = 1, 2, \ldots, i-1$ stappen altijd wel een *Open* knoop geweest met dezelfde $g$- en $f$-waardes als na $i$ stappen.

In beide gevallen is de inductiestap geslaagd. Het gestelde geldt dus ook na $i$ stappen. □

We kunnen Stelling 4 eenvoudig generaliseren tot

**Stelling 5** *Stel dat er na $i \geq 1$ stappen in A*-Vertrouw een Open knoop $k_2$ is, en dat één van zijn huidige echte voorouders $k_0$ voor het laatst in de $i_0$-de stap is geëxpandeerd ($1 \leq i_0 \leq i$). Dan heeft er op het huidige (!) pad van $k_0$ naar $k_2$ na elke stap $j = i_0, i_0+1, \ldots, i$ minstens één Open knoop gestaan met dezelfde $g$- en $f$-waardes als na $i$ stappen.*

**Bewijs** Gaat volstrekt analoog aan het bewijs van Stelling 4. We dienen alleen overal in het bewijs $s$ door $k_0$ te vervangen en te beseffen dat $k_0$ na de $i_0$-de stap niet meer geëxpandeerd is. Als dan $i \geq i_0 + 1$, kan de knoop $k_1$ die in de $i$-de stap geëxpandeerd is, niet gelijk zijn aan $k_0$. □

We gaan nu situaties bekijken, waarin we in de zoekboom van A*-Vertrouw twee *Open* knopen boven elkaar op hetzelfde pad vanaf de startknoop hebben liggen, waarbij bovendien de hoger gelegen knoop een $f$-waarde heeft die minstens even hoog is als de $f$-waarde van de lager gelegen knoop. Voor zulke situaties bewijzen we enkele eigenschappen.

**Lemma 6** *Stel dat er na $i \geq 1$ stappen in A*-Vertrouw voor het eerst een Open knoop $k_2$ in de zoekboom staat met een echte voorouder $k_0$ die ook Open is, zo dat $f(k_2) \leq f(k_0)$. Dan is $k_0$ later (voor het laatst) Open gemaakt dan $k_2$.*

**Bewijs** We merken allereerst op dat het niet mogelijk is dat $k_0$ en $k_2$ allebei in dezelfde stap voor het laatst *Open* zijn gemaakt. Dan zou immers de ouder $k_1$ van $k_2$ ook de ouder van $k_0$ zijn. Dat zou tot gevolg hebben dat $k_0$ nooit een voorouder van $k_2$ zou kunnen zijn.

Stel nu dat $k_0$ *Open* was op het moment dat $k_2$ voor het laatst *Open* werd gesteld en dat $k_0$ sindsdien niet opnieuw *Open* is gemaakt. Dan is de huidige ouder $k_1$ van $k_2$ ontwikkeld nadat $k_0$ voor het laatst *Open* werd gesteld. Laat dit in de $j$-de stap gebeurd zijn voor zekere $1 \leq j \leq i$. Op dat moment was $k_1$ dus *Open*. Hij kan niet gelijk zijn aan $k_0$, want anders zou $k_0$ nu niet meer *Open* zijn. Evenmin kan $k_1$ onmiddellijk voorafgaand aan de $j$-de stap (na $j-1$ stappen dus) $k_0$ als echte voorouder gehad hebben. Op dat moment gold immers $f(k_1) \leq f(k_0)$ (anders zou $k_1$ niet in de $j$-de stap ontwikkeld zijn) en volgens aanname komt het pas na $i$ stappen voor het eerst voor dat een *Open* nakomeling van een *Open* knoop een $f$-waarde heeft die niet hoger is dan die van de betreffende voorouder.

Na $i$ stappen heeft $k_1$ echter wel $k_0$ als echte voorouder omdat zijn kind $k_2$ dat heeft. Knoop $k_1$ is dus, nadat hij zelf ontwikkeld is, alsnog een nakomeling van $k_0$ geworden. Omdat $k_0$ zelf niet meer geëxpandeerd is, kan dit alleen gebeurd zijn doordat in de $j'$-de stap ($j < j' \leq i$) bij het ontwikkelen van een nakomeling $k_0'$ van $k_0$ een beter pad naar $k_1$ of naar één van diens voorouders is gevonden. In dat geval moet echter na $j'-1$ stappen $f(k_0') \leq f(k_0)$ zijn geweest, terwijl $k_0$ toch een echte voorouder van $k_0'$ was en is. Dit is in tegenspraak met de veronderstelling dat zoiets pas na $i$ stappen voor het eerst voorkomt.

De knoop $k_0$ is derhalve *Open* gemaakt nadat dit voor het laatst met $k_2$ gebeurde. □

**Lemma 7** *Stel dat er na $i \geq 1$ stappen in A*-Vertrouw voor het eerst een Open knoop $k_2$ in de zoekboom staat met een echte voorouder $k_0$ die ook Open is, zo dat $f(k_2) \leq f(k_0)$. Dan is $k_0$ in de $i$-de stap Open geworden.*

**Bewijs** Stel dat $k_0$ in de $j$-de stap voor het laatst *Open* is geworden met $j < i$. We weten dan volgens Lemma 6 dat $k_2$ al eerder voor het laatst *Open* is gesteld. Na de $j$-de stap zijn dus zowel $k_0$ als $k_2$ *Open* en voor hun $f$-waardes geldt dan al $f(k_2) \leq f(k_0)$. Volgens de veronderstelling is zoiets pas na $i$ stappen voor het eerst het geval met $k_2$ een echte nakomeling van $k_0$. Daarom is $k_2$ pas na de $j$-de stap een echte nakomeling van $k_0$ geworden.

Nu kunnen we hetzelfde argument gebruiken als aan het eind van Lemma 6. Omdat $k_0$ zelf niet meer geëxpandeerd is na de $j$-de stap, kan $k_2$ alleen een nakomeling van hem geworden zijn doordat in de $j'$-de stap ($j < j' \leq i$) bij het ontwikkelen van een nakomeling $k_0'$ van $k_0$ een beter pad naar $k_2$ of naar één van diens voorouders is gevonden. In dat geval moet echter na $j'-1$ stappen $f(k_0') \leq f(k_0)$ zijn geweest, terwijl $k_0$ toch een echte voorouder van $k_0'$ was en is. Dit is in tegenspraak met de veronderstelling dat zoiets pas na $i$ stappen voor het eerst voorkomt.

De knoop $k_0$ is derhalve in de $i$-de stap voor het laatst *Open* geworden. □

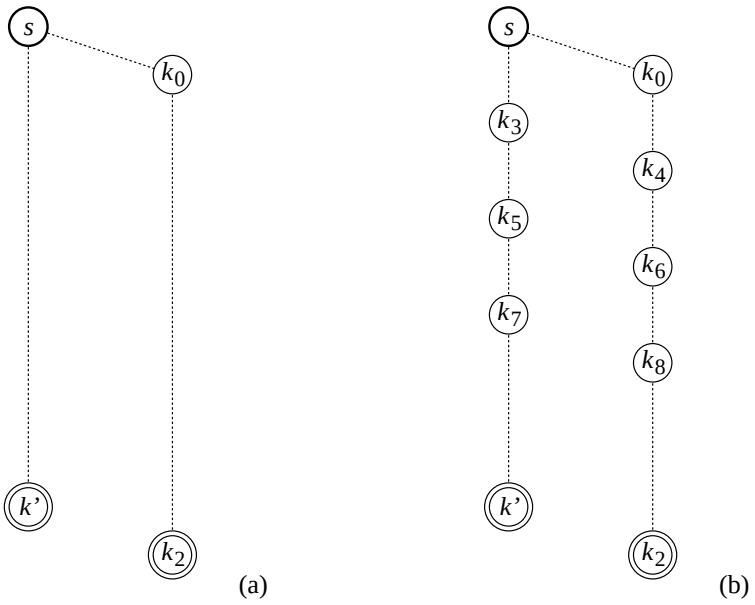
**Lemma 8** *Stel dat er na $i \geq 1$ stappen in A*-Vertrouw voor het eerst een Open knoop $k_2$ in de zoekboom staat met een echte voorouder $k_0$ die ook Open is, zo dat $f(k_2) \leq f(k_0)$. Dan was $k_2$ na $i-1$ stappen al Open en had hij toen dezelfde waarde $f(k_2)$ als na $i$ stappen. Bovendien was hij toen al een nakomeling van $k_0$ en was $k_0$ toen nog Gesloten.*

**Bewijs** Volgens Lemma 6 is $k_2$ na $k_0$, en dus zeker niet in de $i$-de stap voor het laatst *Open* geworden. Hij was dus al na $i-1$ stappen *Open* en zijn $f$-waarde is in de $i$-de stap niet gewijzigd. Wanneer $f'(k_2)$ de $f$-waarde van $k_2$ na $i-1$ stappen voorstelt, geldt dus $f(k_2) = f'(k_2)$.

We weten dankzij Lemma 7 dat in de $i$-de stap de huidige ouder $k'$ van $k_0$ is ontwikkeld. Omdat cykels een niet-negatieve lengte hebben (zie Gevolg 3), was $k'$ na $i-1$ stappen geen nakomeling van $k_0$. Het is daarom niet mogelijk dat $k_2$ pas in de $i$-de stap een nakomeling van $k_0$ geworden is. Hij was dat kennelijk al na $i-1$ stappen.

Het is dan onmogelijk dat $k_0$ na $i-1$ stappen nog *Onbekend* was. In dat geval zou hij (bij A*-Vertrouw) nog nooit geëxpandeerd zijn en dus ook nog geen nakomelingen kunnen hebben.

Omdat $k_0$ in de $i$-de stap in *OPEN* is gezet, moet hij na $i-1$ stappen een hogere $f$-waarde $f'(k_0)$ hebben gehad dan zijn $f$-waarde na $i$ stappen $f(k_0)$. Maar dan was $f'(k_0)$ zeker hoger dan $f(k_2) = f'(k_2)$. Omdat zoiets volgens de veronderstelling pas na $i$ stappen voor het eerst met *Open* knopen $k_0$ en $k_2$ mogelijk is, kan $k_0$ na $i-1$




Figuur 6.1: Schets van de zoekboom na $i-1$ stappen in A*-Vertrouw als er na $i$ stappen voor het eerst een *Open* knoop $k_0$ is met een minstens even hoge $f$-waarde als zijn eveneens *Open* nakomeling $k_2$; in de $i$-de stap zal $k'$ de ouder van $k_0$ worden; (a) de situatie op het eerste gezicht; (b) de situatie bij nadere beschouwing: zowel tussen knoop $s$ en knoop $k'$ als tussen knoop $k_0$ en knoop $k_2$ bevinden zich oneindig veel andere knopen.

stappen niet *Open* geweest zijn. De enig overgebleven mogelijkheid is dus dat $k_0$ na $i-1$ stappen *Gesloten* was. □

Nu zijn we aan de hoofdstelling van deze paragraaf toe:

**Stelling 9** *Het is niet mogelijk dat op enig moment in het A*-algoritme een Open knoop $k_0$ een minstens even hoge $f$-waarde heeft als zijn eveneens Open nakomeling in de zoekboom $k_2$.*

**Bewijs** Laten we aannemen dat het gestelde niet waar is en laten we veronderstellen dat er na $i \geq 1$ stappen in A*-Vertrouw voor het eerst een *Open* knoop $k_2$ in de zoekboom staat met een echte voorouder $k_0$ die ook *Open* is, zo dat $f(k_2) \leq f(k_0)$. We beschouwen nu de situatie na $i-1$ stappen.

Volgens de Lemma's 7 en 8 is $k_0$ in de $i$-de stap *Open* geworden. Zijn ouder $k'$ is in de $i$-de stap ontwikkeld. Na $i-1$ stappen was $k'$ nog *Open*, was $k_0$ nog *Gesloten*, was $k_2$ al *Open* en was $k_2$ al een nakomeling van $k_0$. We hebben derhalve na $i-1$ stappen een situatie als in Fig. 6.1(a) is weergegeven.

Zoals we bij het bewijs van Lemma 8 hebben opgemerkt, is $k'$ geen nakomeling van $k_0$ na $i-1$ stappen, zodat de paden van $s$ naar $k'$ en van $k_0$ naar $k_2$ disjunct zijn. Verder kan $k'$ niet gelijk zijn aan $s$, omdat de startknoop alleen na 0 stappen *Open* is en op dat moment is hij de enige *Open* knoop.

Wanneer we definiëren

$$\begin{aligned} f'(k') &= \text{de } f\text{-waarde van } k' \text{ na } i-1 \text{ stappen} \\ f'(k_0) &= \text{de } f\text{-waarde van } k_0 \text{ na } i-1 \text{ stappen} \end{aligned}$$

$$\begin{aligned} f(k_0) &= \text{de } f\text{-waarde van } k_0 \text{ na } i \text{ stappen} \\ f'(k_2) &= \text{de } f\text{-waarde van } k_2 \text{ na } i-1 \text{ stappen} \\ f(k_2) &= \text{de } f\text{-waarde van } k_2 \text{ na } i \text{ stappen} \end{aligned}$$

dan weten we dat

$$f'(k') \leq f'(k_2) = f(k_2) \leq f(k_0) < f'(k_0) \tag{6.1}$$

In deze vergelijking is de eerste ongelijkheid het gevolg van het feit dat in de $i$-de stap knoop $k'$ wordt ontwikkeld en niet knoop $k_2$. De tweede ongelijkheid komt rechtsstreeks uit de aanname. De gelijkheid en de laatste ongelijkheid volgen uit Lemma 8, respectievelijk 7.

De knoop $k_0$ is na $i-1$ stappen *Gesloten*. Hij is dus eerder ooit eens geëxpandeerd. Laat hij in de $j_0$-de stap voor het laatst zijn geëxpandeerd ($2 \leq j_0 \leq i-1$). Op dat moment had hij dezelfde $f$-waarde als nu: $f'(k_0)$.

Volgens Stelling 4 is er in het huidige pad (het pad na $i-1$ stappen dus) van $s$ naar $k'$ een knoop $k_3 \neq s$ die *Open* was na $j_0 - 1$ stappen en die toen al dezelfde $f$-waarde $f'(k_3)$ had als nu na $i-1$ stappen. In de $j_0$-de stap concurreerde $k_3$ met $k_0$ om uit *OPEN* gehaald en geëxpandeerd te worden. Omdat $k_3$ kennelijk verloor, geldt er dat $f'(k_0) \leq f'(k_3)$. Wanneer we dit combineren met Vgl. 6.1, vinden we

$$f'(k') \leq f'(k_2) = f(k_2) \leq f(k_0) < f'(k_0) \leq f'(k_3) \tag{6.2}$$

Hieruit volgt dat $k_3$ niet gelijk is aan $k'$.

Na $i-1$ stappen moet $k_3$ wel *Gesloten* zijn, want anders zou op dat moment de *Open* knoop $k_3$ een *Open* nakomeling $k'$ hebben met een $f$-waarde $f'(k') \leq f'(k_3)$ (zelfs $f'(k') < f'(k_3)$) en zoiets komt pas na $i$ stappen voor het eerst voor. Merk op dat $k_3$ dan ook precies één keer geëxpandeerd is van stap $j_0$ tot en met stap $i-1$. Als $k_3$ immers in die periode meerdere keren geëxpandeerd zou zijn, zou hij na $i-1$ stappen een lagere $f$-waarde moeten hebben dan na $j_0 - 1$ stappen en dat is niet zo. Laat $k_3$ dan in de $j_3$-de stap ontwikkeld zijn met $j_0 < j_3 \leq i-1$ (voor notationele duidelijkheid slaan we $j_1$ en $j_2$ over).

Volgens Stelling 5 met $i_0 = j_0$ is er nu op het huidige pad van $k_0$ naar $k_2$ een knoop $k_4 \neq k_0$ die na $j_3 - 1$ stappen *Open* was en toen dezelfde $f$-waarde $f'(k_4)$ had die hij na $i-1$ stappen heeft. In de $j_3$-de stap wedijverde $k_4$ dus met $k_3$ om uit *OPEN* gehaald en geëxpandeerd te worden. Kennelijk won $k_3$, zodat moet gelden dat $f'(k_3) \leq f'(k_4)$. Dit kunnen we weer aan Vgl. 6.2 koppelen, zodat we de volgende ongelijkheid krijgen:

$$f'(k') \leq f'(k_2) = f(k_2) \leq f(k_0) < f'(k_0) \leq f'(k_3) \leq f'(k_4) \tag{6.3}$$

Hieruit volgt dat $k_4$ niet gelijk is aan $k_2$.

Na $i-1$ stappen moet $k_4$ *Gesloten* zijn, omdat anders dan al de *Open* knoop $k_4$ een *Open* nakomeling $k_2$ zou hebben met $f'(k_2) \leq f'(k_4)$ (zelfs $f'(k_2) < f'(k_4)$). Opnieuw volgt dat $k_4$ van de $j_3$-de tot en met de $i-1$-ste stap precies één keer ontwikkeld is. Laat dit in de $j_4$-de stap gebeurd zijn.

Dan volgt uit Stelling 5 dat er op het huidige pad van $k_3$ naar $k'$ een knoop $k_5 \neq k_3$ ligt die na $j_4 - 1$ stappen *Open* was en toen dezelfde $f$-waarde $f'(k_5)$ had als na $i-1$ stappen. In de $j_4$-de stap concurreerde $k_5$ met $k_4$ om uit *OPEN* gehaald en geëxpandeerd te worden, maar kennelijk won $k_4$. Er geldt dus dat $f'(k_4) \leq f'(k_5)$. Deze ongelijkheid kunnen we weer aan laten sluiten bij Vgl. 6.3, zodat we krijgen

$$f'(k') \leq f'(k_2) = f(k_2) \leq f(k_0) < f'(k_0) \leq f'(k_3) \leq f'(k_4) \leq f'(k_5) \tag{6.4}$$

Het is duidelijk dat $k_5$ niet gelijk is aan $k'$.

Wanneer we op deze manier doorgaan, zullen we een oneindige rij verschillende knopen $k_3, k_4, k_5, k_6, k_7, k_8, \ldots$ vinden waarbij voor de $f$-waardes na $i-1$ stappen geldt dat $f'(k_l) \leq f'(k_{l+1})$ voor $l \geq 3$. De knopen met een oneven index $l$ liggen (niet noodzakelijk onmiddellijk) na elkaar op het pad van $s$ naar $k'$; die met een even index $l$ liggen tussen $k_0$ en $k_2$ (zie Fig. 6.1(b)). Omdat we van een eindige graaf en een dito zoekboom waren uitgegaan, is dit niet mogelijk. Bovendien zouden al die knopen tussen de $j_0$-de en de $i$-de stap van het algoritme geëxpandeerd moeten zijn. Ook dat is niet mogelijk voor een oneindig aantal knopen.

We kunnen daarom concluderen dat de aanname die we aan het begin van het bewijs maakten, niet klopt. Het zal nooit voorkomen in de variant A*-Vertrouw dat een *Open* knoop $k_2$ een *Open* voorouder $k_0$ heeft en dat geldt $f(k_2) \leq f(k_0)$. □

**Gevolg 10** *In A*-Vertrouw zal een Open knoop $k_2$ met een Open voorouder $k_0$ niet geëxpandeerd worden voor $k_0$ zelf, zolang $k_0$ zijn voorouder blijft. De effectieve zoekboom van A*-Vertrouw is dus de zoekboom van het algoritme met daaruit verwijderd de knopen met een Open voorouder.*

**Bewijs** Dankzij Stelling 9 weten we dat de *Open* knoop $k_0$, zolang hij een voorouder van de *Open* knoop $k_2$ is, een lagere $f$-waarde zal hebben dan $k_2$. □

Tenslotte bewijzen we nog enkele stellingen over de correctheid van de $g$-waardes van de knopen in de zoekboom van A*-Vertrouw.

**Stelling 11** *Voor elke knoop $k$ in de zoekboom na $i \geq 0$ stappen in A*-Vertrouw geldt:*
*knoop $k$ heeft alleen Gesloten echte voorouders in de zoekboom $\Longleftrightarrow$ $g(k)$ is gelijk aan de lengte van het pad van de startknoop $s$ naar $k$ in de zoekboom;*
*knoop $k$ heeft minstens één Open voorouder in de zoekboom $\Longleftrightarrow$ $g(k)$ is groter dan de lengte van het pad van de startknoop $s$ naar $k$ in de zoekboom.*

**Bewijs** Gaat met inductie naar het aantal stappen $i$.

Als $i = 0$, bevindt alleen de startknoop $s$ zich in de zoekboom. Die heeft dan geen *Open* voorouders in de zoekboom, dus formeel heeft hij alleen *Gesloten* voorouders. Inderdaad geldt dan dat $g(s) = 0 =$ de lengte van het pad van knoop $s$ naar knoop $s$ in de zoekboom.

Veronderstel dat het gestelde waar is tot en met $i-1$ stappen in A*-Vertrouw (inductiehypothese) en dat we nog geen eindknoop hebben ontdekt. Laat $k_1$ de knoop zijn die in de $i$-de stap wordt geëxpandeerd. Volgens Gevolg 10 had $k_1$ na $i-1$ stappen alleen *Gesloten* voorouders. Volgens de inductiehypothese gold toen dus dat de $g$-waardes van $k_1$ en van zijn voorouders gelijk waren aan de lengtes van de respectievelijke paden vanaf $s$ naar hen. Omdat een cykel altijd niet-negatieve lengte heeft (zie Gevolg 3), zijn de $g$-waardes van de voorouders van $k_1$ in de zoekboom niet gewijzigd en zijn die voorouders *Gesloten* gebleven in de $i$-de stap van het algoritme. Voor $k_1$ en voor zijn voorouders geldt dus dat ze ook na $i$ stappen alleen *Gesloten* voorouders hebben en dat hun $g$-waardes gelijk zijn aan de lengtes van hun respectievelijke paden vanaf $s$.

Elk kind $k_2$ van $k_1$ na $i$ stappen in het algoritme heeft dan geen *Open* voorouders meer in de zoekboom. Zijn ouder $k_1$ heeft dat immers ook niet en is zelf inmiddels ook *Gesloten*. We moeten dus bewijzen dat $k_2$ na de $i$-de stap een $g$-waarde heeft die overeenkomt met de lengte van het pad vanaf $s$.

Knoop $k_2$ heeft bij het ontwikkelen van $k_1$ in de $i$-de stap de $g$-waarde $g(k_1)+c(k_1, k_2)$ aangeboden gekregen. Deze waarde komt overeen met de lengte van het pad van $s$ naar $k_2$ in de zoekboom na $i$ stappen. We onderscheiden nu twee gevallen:

- Knoop $k_2$ was na $i-1$ stappen al een kind van $k_1$. Zijn pad vanaf $s$ in de zoekboom na $i$ stappen is dan gelijk aan zijn pad vanaf $s$ in de zoekboom na $i-1$ stappen. Omdat hij na $i-1$ stappen de *Open* knoop $k_1$ boven zich had in dit pad, was zijn $g$-waarde toen hoger dan de lengte van dit pad. Hij heeft nu in de $i$-de stap een lagere (namelijk de correcte) $g$-waarde gekregen en is met deze lagere $g$-waarde in *OPEN* gezet.

- Knoop $k_2$ was na $i-1$ stappen nog geen kind van $k_1$. Omdat hij in de $i$-de stap $k_1$ als ouder heeft geaccepteerd, had hij na $i-1$ stappen kennelijk een hogere $g$-waarde dan de waarde die $k_1$ hem aanbood. In de $i$-de stap is hij dus met die lagere (en correcte) $g$-waarde in *OPEN* gezet.

Een knoop $k_3$ die na $i$ stappen een nakomeling van een kind $k_2$ van $k_1$ in de zoekboom is, heeft dan de *Open* knoop $k_2$ als voorouder. We moeten dus laten zien dat $k_3$ na $i$ stappen een te hoge $g$-waarde heeft.

Knoop $k_3$ was al nakomeling van $k_2$ na $i-1$ stappen, want de ontwikkelde knoop $k_1$ ligt boven $k_2$ in de zoekboom na $i$ stappen. Laat $g'(k_2)$ en $g'(k_3)$ de $g$-waardes van $k_2$, respectievelijk $k_3$ na $i-1$ stappen zijn en laat $g(k_2)$ en $g(k_3)$ de $g$-waardes van $k_2$ respectievelijk $k_3$ na $i$ stappen zijn. Dan geldt volgens Gevolg 2 $g'(k_3) \geq g'(k_2)+$ de lengte van het pad van $k_2$ naar $k_3$. Nu is dat pad in de $i$-de stap niet veranderd. Ook de $g$-waarde van $k_3$ is nog hetzelfde, want $k_3$ is geen kind van $k_1$ geworden. We hebben echter zojuist gezien dat $k_2$ in de $i$-de stap een lagere (correcte) $g$-waarde heeft gekregen. Daardoor is $g(k_3) > g(k_2)+$ de lengte van het pad van $k_2$ naar $k_3$ = de lengte van het pad van $s$ naar $k_3$ na $i$ stappen.

Tenslotte beschouwen we de knopen in de zoekboom na $i$ stappen die op dat moment noch een voorouder van $k_1$, noch $k_1$ zelf, noch een nakomeling van $k_1$ zijn. Laat $k_4$ zo'n knoop zijn. Omdat hij geen kind van $k_1$ is na $i$ stappen, stond $k_4$ na $i-1$ stappen al in de zoekboom en is zijn $g$-waarde niet veranderd.

Als het pad vanaf $s$ naar $k_4$ veranderd zou zijn, zou dat veroorzaakt moeten zijn doordat $k_4$ of een van zijn voorouders in zijn pad na $i-1$ stappen een betere $g$-waarde kreeg aangeboden bij het ontwikkelen van $k_1$. Maar dan zou $k_4$ na $i$ stappen een nakomeling van $k_1$ zijn. Omdat dat niet zo is, is het pad vanaf $s$ naar $k_4$ in de $i$-de stap niet veranderd.

Ook zijn er geen knopen in dat pad *Open* geworden terwijl ze na $i-1$ stappen *Gesloten* waren of andersom. Een verandering van status overkomt namelijk alleen $k_1$ en daarnaast hooguit de kinderen van $k_1$.

Voor $k_4$ is er in de $i$-de stap dus niets veranderd. Omdat volgens de inductiehypothese het gestelde na $i-1$ stappen al voor deze knoop gold, geldt het ook na $i$ stappen.

Op deze manier hebben we voor alle knopen in de zoekboom na $i$ stappen bewezen dat het gestelde ook op dat moment voor hen geldt. De inductiestap is dus geslaagd.

□

**Stelling 12** *Als een knoop $k_2$ eenmaal een Open voorouder in de zoekboom van A*-Vertrouw heeft, zal dat zo blijven zolang de g-waarde van $k_2$ niet is aangepast. Pas wanneer dat wel gebeurt, krijgt $k_2$ alleen maar Gesloten voorouders.*

**Bewijs** Gaat met inductie naar het aantal stappen $i$.

Stel dat knoop $k_2$ na $i-1$ stappen een *Open* voorouder heeft. Dan heeft hij volgens Stelling 11 op dat moment een $g$-waarde die hoger is dan de lengte van het pad vanaf

$s$. Als we in $i-1$ stappen nog geen eindknoop gevonden hebben, wordt in de $i$-de stap een knoop $k_1$ ontwikkeld.

Stel nu dat in de $i$-de stap de $g$-waarde van $k_2$ niet verandert. Dan onderscheiden we twee gevallen:

- Het pad vanaf $s$ naar $k_2$ in de zoekboom is in stap $i$ niet gewijzigd. Dan is de $g$-waarde na $i$ stappen nog steeds hoger dan de lengte van dit pad. Volgens Stelling 11 heeft $k_2$ dan nog steeds een *Open* voorouder.

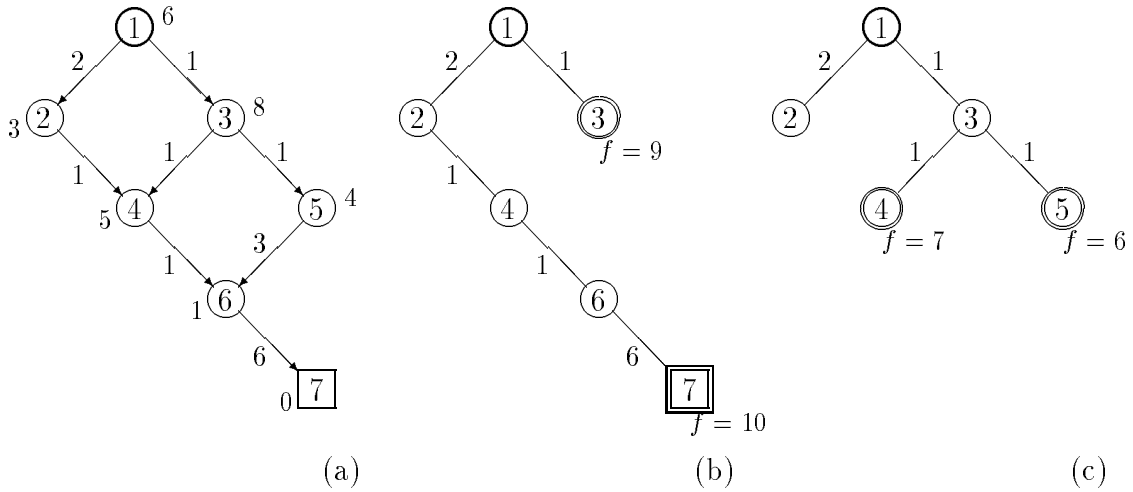
- Het pad vanaf $s$ naar $k_2$ in de zoekboom is in stap $i$ wel gewijzigd. Omdat de $g$-waarde van $k_2$ zelf niet is veranderd, heeft een van de voorouders van $k_2$ in de zoekboom na $i-1$ stappen in de $i$-de stap een betere $g$-waarde gekregen. Met die betere waarde is die voorouder dan in *OPEN* gezet. Omdat die voorouder ook na $i$ stappen nog een voorouder van $k_2$ is, heeft de laatste ook na $i$ stappen een *Open* voorouder.

Als de $g$-waarde van $k_2$ in de $i$-de stap wel verandert, moet hij in de $i$-de stap een kind van $k_1$ geworden zijn. Vanwege Gevolg 10 had $k_1$ na $i-1$ stappen geen *Open* voorouders. Omdat cykels altijd niet-negatieve lengtes hebben (zie Gevolg 3), geldt dit ook na $i$ stappen. Ook knoop $k_1$ is na de $i$-de stap, waarin hij zelf ontwikkeld werd, *Gesloten*. Derhalve heeft knoop $k_2$ na $i$ stappen alleen nog *Gesloten* voorouders in de zoekboom. □

## 6.2 Ontwikkelde Knopen bij A*-Vertrouw

Het verschil tussen A*-Verwijder en A*-Vertrouw is dat bij de laatste de nakomelingen in de zoekboom van een al geëxpandeerde knoop $k_2$ niet uit de zoekboom verwijderd worden als er een beter pad naar $k_2$ wordt ontdekt. De zoekboom van A*-Vertrouw bevat dus extra knopen in vergelijking met de zoekboom van A*-Verwijder. Als gevolg van de verbetering van het pad van de startknoop $s$ naar $k_2$ zijn de $g$- en $f$-waardes van die knopen niet meer correct. In het algemeen zitten er bij de extra knopen ook *Open* knopen. Omdat die knopen echter de *Open* knoop $k_2$ als voorouder hebben, zullen ze volgens Gevolg 10 niet geëxpandeerd worden. Ze vormen dus geen direct gevaar voor het zoekproces. Ook later vormen de knopen met verkeerde (te hoge) $g$- en $f$-waardes geen bedreiging. Volgens Stelling 12 zullen ze immers pas vrij van *Open* voorouders worden als ze nieuwe $g$- en $f$-waardes hebben. Die nieuwe waardes komen dan meteen overeen met het pad vanaf $s$.

Aanvankelijk staan nu, afgezien van de kansloze knopen die toch een *Open* voorouder hebben, alle knopen die bij A*-Vertrouw in *OPEN* staan met dezelfde $g$- en $f$-waardes ook bij A*-Verwijder in *OPEN*. Stel vervolgens dat dit op een bepaald moment geldt, waarna in de de eerstvolgende stap van A*-Vertrouw een knoop $k_1$ geëxpandeerd wordt. Dan zal, vanwege de overeenkomst tussen de lijsten *OPEN*, $k_1$ ook bij A*-Verwijder geëxpandeerd worden en er zullen dezelfde opvolgers $k_2$ gegenereerd worden. Omdat de $g$-waarde van $k_1$ bij beide varianten hetzelfde is, krijgt zo'n opvolger $k_2$ ook dezelfde $g$-waardes aangeboden. We nemen nu even aan dat de oude $f$-waarde van $k_2$ bij A*-Verwijder nooit lager zal zijn bij A*-Vertrouw. Als $k_2$ dus bij A*-Vertrouw $k_1$ als ouder aanvaardt, zal dit zeker ook bij A*-Verwijder gebeuren. Op die manier staan ook ná het ontwikkelen van $k_1$ de *Open* knopen bij A*-Vertrouw (voor zover ze geen *Open* voorouder hebben) ook bij A*-Verwijder in *OPEN*.




Figuur 6.2: (a) Voorbeeldgraaf waarvoor er bij A*-Verwijder knopen in *OPEN* terechtkomen die er bij A*-Vertrouw niet in komen: (b) de bijbehorende zoekboom na 4 stappen in het A*-algoritme; in de volgende stap wordt het kortere pad naar $k_0 = 4$ gevonden (via knoop 3); (c) nadat de subzoekboom met knoop 4 als wortel is verwijderd (op knoop 4 zelf na), vinden we in de volgende stap van $A^*$-Verwijder een pad van $s = 1$ via $k_1 = 5$ naar $k_2 = 6$ dat slechter is dan het oude pad via knoop 4; dat oude pad kennen we echter niet meer zodat we dit nieuwe pad (voorlopig) als beste pad van $s$ naar knoop 6 onthouden.

Het is wel mogelijk dat *OPEN* bij A*-Verwijder daarnaast nog andere knopen bevat. Het kan namelijk zo zijn dat de knoop $k_2$ die als opvolger van $k_1$ gegenereerd is, bij A*-Verwijder ooit uit de zoekboom is verwijderd, omdat er een beter pad naar zijn voorouder $k_0$ was gevonden. Daarbij heeft $k_2$ een $f$-waarde $\infty$ gekregen, terwijl hij bij A*-Vertrouw met zijn eindige waarde in de zoekboom bleef staan. Als die $f$-waardes nadien niet meer zijn gewijzigd, heeft $k_2$ bij A*-Vertrouw een lagere $f$-waarde dan bij A*-Verwijder. Het gevolg daarvan kan zijn dat het aanbod van $k_1$ aan $k_2$ bij A*-Verwijder wel wordt aanvaard, terwijl dat bij A*-Vertrouw niet gebeurt. In Fig. 6.2 is zo'n situatie uitgebeeld.

Zoiets kan echter niet gebeuren als het ontwikkelen van $k_1$ past in het indirect propageren van de verbetering in het pad naar $k_0$ in de richting van $k_2$. Laat immers $g'(k_2)$ de lengte van het oude pad van $s$ via $k_0$ naar $k_2$ zijn en $\tilde{g}(k_2)$ de lengte van het verbeterde pad van $s$ via $k_0$ naar $k_2$. Dan geldt dat $\tilde{g}(k_2) < g'(k_2)$. Dan is in dit geval de nieuwe $g$-waarde $g(k_2)$ die $k_2$ krijgt aangeboden gelijk aan $\tilde{g}(k_2)$, terwijl $k_2$ tot nu toe de te hoge waarde $g'(k_2)$ aanhield. In dit geval zal ook bij A*-Vertrouw $k_2$ het aanbod van $k_1$ dus accepteren.

Indien echter het ontwikkelen van $k_1$ niets te maken heeft met het propageren van de verbetering in het pad naar $k_0$, gaat deze redenering niet op. Dan is het dus wel mogelijk dat $k_2$ bij A*-Verwijder het aanbod van $k_1$ als een verbetering beschouwt, terwijl hij dat bij A*-Vertrouw niet doet.

Dit verschil tussen de twee varianten kan echter geen desastreuse gevolgen hebben voor de correctheid van A*-Vertrouw. Omdat de aangeboden $g$-waarde $g(k_2)$ minstens zo groot is als de oude $g$-waarde $g'(k_2)$ en die op zijn beurt weer groter is dan de werkelijke lengte $\tilde{g}(k_2)$ van het verbeterde pad van $s$ via $k_0$ naar $k_2$, heeft het helemaal

geen zin om, zoals in A*-Verwijder gedaan wordt, die aangeboden waarde te accepteren. Uiteindelijk is er tenslotte toch nog een beter pad van $s$ naar $k_2$ en dat zal zonodig later alsnog ontdekt worden.

Het aantal knopen dat bij A*-Verwijder (zonder nut) extra in *OPEN* gezet wordt in vergelijking met A*-Vertrouw, is beperkt. In principe heeft zo'n knoop $k_2$ met de te hoge waarde wel een kans om op zijn beurt ook geëxpandeerd te worden, zoals ook met knoop 6 in Fig. 6.2 zal gebeuren. Wanneer hij echter een *Open* nakomeling van $k_0$ was juist voordat het betere pad naar $k_0$ werd gevonden, verdwijnt die kans. Hij heeft namelijk een $g$-waarde $g(k_2)$ die minstens zo hoog is als de lengte $g'(k_2)$ van het oude pad van $s$ via $k_0$ naar $k_2$. Dankzij Gevolg 10 weten we dat die laatste $g$-waarde al niet laag genoeg zou zijn om in A*-Vertrouw een expansie te bewerkstelligen, omdat er nog een voormalig voorouder van $k_2$ in *OPEN* gereed staat om de verbetering van het pad naar $k_0$ in de richting van $k_2$ door te geven. Dan zal zeker $g(k_2)$ daar niet laag genoeg voor zijn.

Ook wanneer knoop $k_2$ *Gesloten* was op het moment dat er een korter pad naar zijn voorouder $k_0$ werd ontdekt, zal hij bij A*-Verwijder met zijn nieuwe waarde $g(k_2)$ niet meer nakomelingen in de zoekboom kunnen krijgen dan hij had toen hij destijds uit de zoekboom verwijderd werd.[1]

Kortom, hoewel in A*-Verwijder knopen in *OPEN* gezet kunnen worden die dat in A*-Vertrouw niet zou overkomen, kunnen we met die knopen niet 'door de bodem van de subboom van voormalige nakomelingen van $k_0$ heen zakken'.

Dat verzekert ons er tevens van dat er bij A*-Verwijder geen knopen gegenereerd worden die bij A*-Vertrouw nog *Onbekend* blijven. Onze aanname dat de $f$-waarde van een knoop bij A*-Verwijder niet lager zal zijn dan bij A*-Vertrouw, wordt hierdoor dus niet geschonden. Omdat de aanname aan het begin van de beide varianten geldig is, en er ook nergens anders een situatie gecreëerd wordt waarin de aanname geschonden kan worden, mogen we hem inderdaad algemeen geldig verklaren.

Bij de eerste presentatie van de variant A*-Vertrouw, in paragraaf 5.3, hielden we er nog rekening mee dat er in die variant onnodig knopen met incorrecte $g$- en $f$-waardes ontwikkeld zouden worden die bij A*-Verwijder niet ontwikkeld worden. We hebben nu aangetoond dat het eerder omgekeerd is. Alle knopen die in A*-Vertrouw ontwikkeld worden, worden ook in A*-Verwijder ontwikkeld. Wel kunnen er bij A*-Verwijder nog extra knopen ontwikkeld worden, maar dat zijn dan toch nutteloze knopen. Het eindresultaat (het pad dat A*-Verwijder oplevert) wordt er niet door beïnvloed. Het algoritme wordt er alleen maar mee vertraagd.

Hieruit kunnen we concluderen dat A*-Vertrouw hetzelfde resultaat geeft als A*-Verwijder: het kortste pad van de startknoop $s$ naar het doel. De variant A*-Vertrouw is dus ook correct.

[1] Dit is niet helemaal correct. Stel dat er behalve naar $k_0$ ook naar een knoop $k_0'$ een beter pad is gevonden, zodat diens nakomelingen ook uit de zoekboom zijn verwijderd. Dan is het in principe mogelijk dat één van die nakomelingen, die nu een oneindig grote $f$-waarde heeft, knoop $k_2$ als vader zal accepteren als die geëxpandeerd wordt. Er zullen hierdoor echter nooit knopen met nutteloze $g$- en $f$-waardes in *OPEN* terechtkomen, die er nog nooit eerder in gestaan hebben.

# Hoofdstuk 7

# Vergelijking van de Varianten

In dit hoofdstuk gaan we de efficiëntie van de drie beschreven varianten van het A*-algoritme met elkaar vergelijken. We zullen dit eerst vanuit een theoretische invalshoek, vooral een beetje intuïtief doen. Vervolgens beschouwen we enkele geconstrueerde voorbeelden van grafen. Voor sommige voorbeelden blijkt de ene variant beter te zijn en voor andere voorbeelden de andere. Tenslotte meten we de hoeveelheid werk die elke variant nodig heeft voor een aantal willekeurige grafen met 25, 50 en 100 knopen.

We meten de efficiëntie van de varianten allereerst door het aantal takken uit de graaf te tellen dat we in elk van de varianten aflopen om het kortste pad van de startknoop naar het doel te bepalen. In het A*-algoritme bezoeken we in feite iedere keer dat we bij het expanderen van een knoop een kind genereren, een tak. Daarnaast gebruiken we in A*-Propageer takken uit de graaf om een verbetering van het pad naar $k_2$ aan de andere knopen door te geven en bij A*-Verwijder om de nakomelingen van $k_2$ uit de zoekboom te verwijderen.

Het is helaas niet zo dat de totale hoeveelheid werk die in het A*-algoritme verricht wordt, per se naar boven wordt begrensd door een constante maal het aantal in het algoritme bezochte takken. Als we immers een tak hebben afgelopen en het eindpunt van die tak moet met een nieuwe of verbeterde $f$-waarde in *OPEN* gezet worden, dan kan het nog heel wat tijd vergen om de juiste positie voor die knoop in *OPEN* te bepalen. Die tijd hangt, behalve van de kwaliteit van $h$, gemiddeld gesproken af van de grootte van de lijst *OPEN*. Deze grootte hangt in het algemeen weer af van het aantal knopen $N$ in de graaf. De tijd die het kost om een knoop in *OPEN* te zetten zal dus in het algemeen niet constant zijn, maar groter worden als $N$ groter wordt.

Daar moeten we rekening mee houden als we de efficiëntie van de verschillende varianten van het A*-algoritme met elkaar willen vergelijken. Daarom kijken we ook naar het aantal vergelijkingen dat we moeten maken om de positie in *OPEN* te bepalen waar een zojuist gegenereerde knoop neergezet moet worden om er voor te zorgen dat de oplopende volgorde van de $f$-waardes van de knopen in *OPEN* behouden blijft.

We tellen het aantal vergelijkingen van de $f$-waarde van de in *OPEN* in te voegen knoop met $f$-waardes van andere knopen in *OPEN*. Als we een knoop dus achter in de lijst moeten zetten, kost het volgens deze telmethode geen vergelijking extra om vast te stellen dat we aan het eind van de lijst zijn aangekomen; zo kunnen we bijvoorbeeld aan het begin van het algoritme zonder moeite knoop $s$ in de (dan nog lege) lijst *OPEN* zetten. Verder doorlopen we de lijst *OPEN* voor elke knoop van voren af aan; we gebruiken dus geen informatie over bijvoorbeeld de $f$-waarde en de positie in *OPEN* van de vorige knoop die we *Open* gemaakt hebben.

Voor gegeven eindige grafen wordt de totale hoeveelheid werk in het A*-algoritme

wel naar boven begrensd door een constante maal de som van het totale aantal bezochte takken en het totale aantal gemaakte vergelijkingen in *OPEN*.

## 7.1 Theoretische Vergelijking

Omdat de drie varianten verschillen in wat er gebeurt na de ontdekking van een beter pad van de startknoop $s$ naar een al *Gesloten* knoop $k_2$, richten we daar onze aandacht op.

Laten we de subzoekboom met wortel $k_2$ definiëren als de subboom met wortel $k_2$ van de zoekboom op het moment dat we een beter pad naar de al ontwikkelde knoop $k_2$ vinden. Hierin zitten dus alle nakomelingen van $k_2$ in de zoekboom met respectievelijke paden vanaf $k_2$. Laten we verder de subgraaf met wortel $k_2$ definiëren als de subzoekboom met wortel $k_2$ aangevuld met de knopen buiten die subzoekboom voor wie de verbetering van het pad van $s$ naar $k_2$ ook een beter pad vanaf $s$ tot gevolg heeft.

Als we dan A*-Propageer gebruiken, lopen we de subgraaf met wortel $k_2$ af om de verbetering van het pad naar $k_2$ door te geven. Bij A*-Verwijder lopen we eerst de subzoekboom met wortel $k_2$ af om alle nakomelingen van $k_2$ te verwijderen. Daarna bezoeken we in principe stapsgewijs de subgraaf met wortel $k_2$ om de verbetering indirect toch te propageren naar de knopen in die subgraaf (we verontachtzamen hierbij voor het gemak de mogelijkheid dat we eerder voor een van die knopen een pad vanaf $s$ vinden waarin $k_2$ helemaal niet voorkomt). Bij A*-Vertrouw, tenslotte, slaan we het verwijderen over. Al knopen ontwikkelend, doorlopen we dus alleen de subgraaf met wortel $k_2$ om de verbetering indirect door te geven.

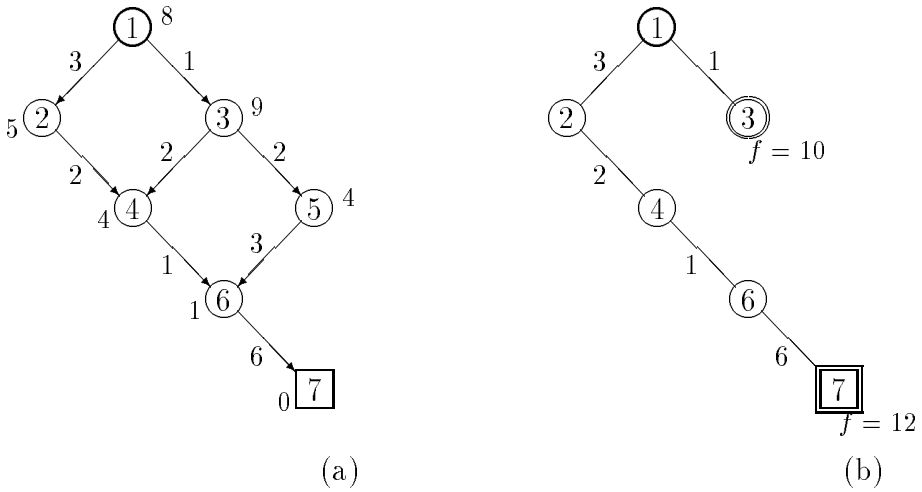
Uit deze weinig verfijnde analyse kunnen we concluderen dat in het algemeen bij A*-Verwijder het grootste aantal takken bezocht zal worden. Behalve de subgraaf met wortel $k_2$, die we ook bij de andere twee varianten doorlopen, bezoeken we daar ook de subzoekboom met wortel $k_2$. De andere twee varianten zijn dan ongeveer even goed.

We kunnen ook de wetenschap gebruiken die we in hoofdstuk 6 hebben afgeleid: elke knoop die in A*-Vertrouw wordt geëxpandeerd, wordt ook in A*-Verwijder geëxpandeerd. Daarnaast is het mogelijk dat we bij A*-Verwijder nog extra knopen expanderen. Bovendien verwijderen we daar ook nog knopen uit de zoekboom als we een beter pad naar een knoop $k_2$ hebben gevonden.

Hieruit volgt dat voor elke instantie A*-Verwijder minstens evenveel takken bezoekt als A*-Vertrouw. Wat dat betreft is A*-Vertrouw dus altijd minstens zo goed als A*-Verwijder.

Voor elk van de andere twee paren varianten ({A*-Propageer, A*-Verwijder} en {A*-Propageer, A*-Vertrouw}) blijkt het niet mogelijk om te zeggen dat de ene altijd minstens zoveel takken bezoekt als de andere of andersom. Er zijn instanties waarvoor één van de twee varianten uit het paar meer takken bezoekt en er zijn instanties waarvoor de ander meer takken bezoekt. Hiervan zullen we in de volgende paragraaf enkele voorbeelden zien.

Bij A*-Propageer wordt een eenmaal *Gesloten* knoop nooit opnieuw in *OPEN* gezet, terwijl dit bij A*-Verwijder en A*-Vertrouw wel mogelijk is. Hierdoor zal bij A*-Propageer in het algemeen het kleinste aantal knopen in *OPEN* gezet hoeven te worden. Het ligt dan ook voor de hand dat A*-Propageer gemiddeld het kleinste aantal vergelijkingen in de lijst *OPEN* nodig zal hebben. Van de andere twee varianten worden er bij A*-Verwijder nog wel eens knopen (al dan niet tijdelijk) uit *OPEN* verwijderd die




Figuur 7.1: (a) Voorbeeldgraaf voor het A*-algoritme, waarvoor A*-Propageer van de drie varianten de minste takken bezoekt en ook de minste vergelijkingen in *OPEN* uitvoert; (b) de bijbehorende zoekboom van het A*-algoritme na 4 stappen; in de volgende stap wordt het kortere pad naar knoop 4 ontdekt.
A*-Propageer komt uiteindelijk uit op 10 bezochte takken en 5 vergelijkingen, A*-Verwijder op 14 takken en 7 vergelijkingen en A*-Vertrouw op 11 takken en 9 vergelijkingen.

er bij A*-Vertrouw in blijven staan. Het is dus te verwachten dat het aantal knopen in *OPEN* en dan ook het aantal vergelijkingen dat in *OPEN* wordt uitgevoerd, bij A*-Vertrouw in het algemeen het grootst zal zijn.

Er blijkt echter geen algemene regel te zijn volgens welke één variant, voor zover het dit criterium betreft, altijd minstens zo goed zou zijn als een andere.

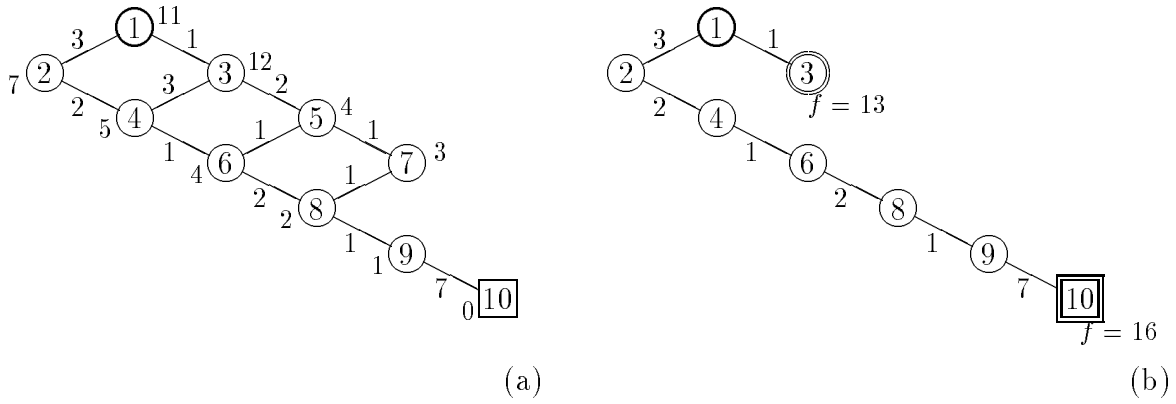
## 7.2 Enkele Voorbeelden

Zoals we opmerkten in de vorige paragraaf, presteert de ene variant beter voor één graaf, terwijl voor een andere graaf de andere variant weer beter werkt. We zullen nu enkele (geconstrueerde) voorbeelden laten zien die die opmerking bevestigen.

Eerst nemen we het aantal door de verschillende varianten bezochte takken onder de loep.

Fig. 7.1(a) bevat een voorbeeldgraaf waarvoor A*-Propageer het minste aantal takken afloopt, gevolgd door A*-Vertrouw en daarna A*-Verwijder. De zoekboom die in de eerste 4 stappen van het A*-algoritme wordt gecreëerd staat in Fig. 7.1(b). In de volgende stap wordt een beter pad naar knoop 4 ontdekt. Wanneer deze verbetering in A*-Propageer naar alle nakomelingen van deze knoop wordt gepropageerd, blijken we meteen het kortste pad te hebben gevonden. De knopen op dit pad krijgen dan ook hun optimale waardes. Weliswaar wordt nu knoop 5 nog wel geëxpandeerd voordat de eindknoop 7 uit *OPEN* gehaald wordt, maar dit levert uiteraard geen verbetering meer op voor de knopen op het kortste pad.

Bij de andere twee varianten weten de knopen 6 en 7 bij het expanderen van knoop 5 hun optimale waardes nog niet. Daardoor accepteert knoop 6 tijdelijk knoop 5 als




Figuur 7.2: (a) Voorbeeldgraaf voor het A*-algoritme, waarvoor A*-Propageer van de drie varianten de meeste takken bezoekt; (b) de bijbehorende zoekboom van het A*-algoritme na 6 stappen; in de volgende stap wordt het kortere pad naar knoop 4 ontdekt.
A*-Propageer komt uiteindelijk uit op 21 bezochte takken, A*-Verwijder op 20 takken en A*-Vertrouw op 16 takken.

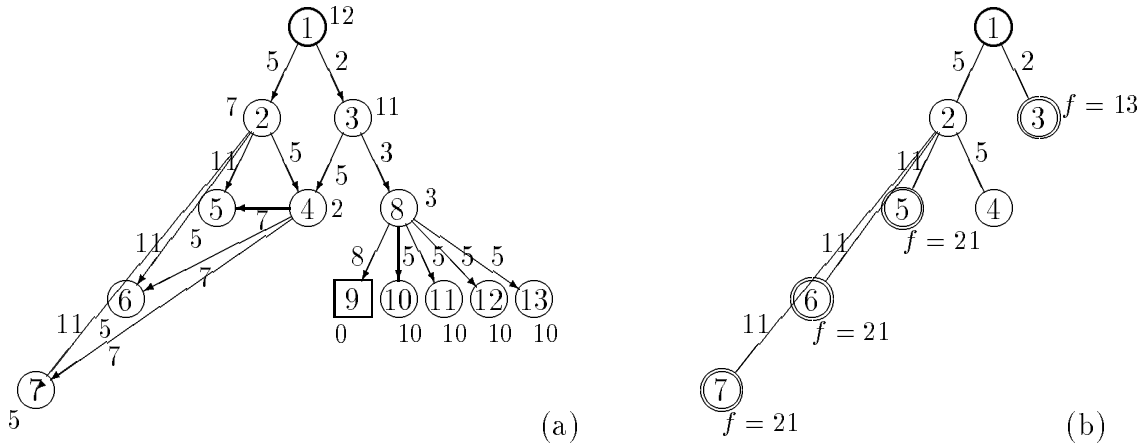
vader. Hij wordt zelfs met de nog niet optimale $f$-waarde ontwikkeld, vóórdat de optimale waarde doordringt. Het gevolg is dat er bij A*-Vertrouw en dan zeker bij A*-Verwijder meer takken worden bezocht dan bij A*-Propageer.

Dat het echter ook andersom kan zijn, blijkt uit Fig. 7.2. In de eerste 6 stappen in het A*-algoritme voor de graaf in Fig. 7.2(a) hebben we de zoekboom in Fig. 7.2(b) opgebouwd. Het pad naar de eindknoop dat we tot nu toe hebben gevonden, verschilt behoorlijk van het kortste pad $(1, 3, 5, 7, 8, 9, 10)$, maar vanwege de relatief hoge $h$-waarde van knoop 3 hebben we wel eerst dit langere pad helemaal afgelopen. In de zevende stap komen we er achter dat knoop 3 toch zo gek nog niet is, want we vinden het betere pad via die knoop naar knoop 4. Bij A*-Propageer wordt deze verbetering over het hele verdere gedeelte van het oude pad naar knoop 7 gepropageerd. Omdat dit pad nog lang niet goed is, is dit volkomen zinloos.

Wanneer vervolgens knoop 5 wordt ontwikkeld, vinden we een verbetering van het pad naar knoop 6. Ook nu wordt die verbetering door A*-Propageer (onnodig) doorgegeven tot aan de eindknoop. Bij het ontwikkelen van knoop 7 wordt immers pas het optimale pad gevonden.

In tegenstelling tot wat A*-Propageer doet, loopt A*-Verwijder na de zesde stap nog maar één keer het aanvankelijk gevonden pad naar de eindknoop door, om het in z'n geheel uit de zoekboom te vewijderen. Daarna loopt A*-Verwijder recht op het doel af. Nadat knoop 8 is bereikt, worden knoop 4 en knoop 6 nog wel eerst geëxpandeerd. Echter, omdat er geen verbetering meer wordt gevonden, geeft dit één bezochte tak per ontwikkelde knoop extra. Aldus bezoekt A*-Verwijder voor deze instantie minder takken dan A*-Propageer. A*-Vertrouw hoeft nog minder takken af te lopen om het kortste pad te vinden.

Wat betreft het aantal vergelijkingen van $f$-waardes in de lijst *OPEN*, spraken we in de vorige paragraaf het vermoeden uit dat A*-Propageer gemiddeld het kleinste aantal van zulke vergelijkingen nodig zal hebben, gevolgd door A*-Verwijder en daarna A*-Vertrouw. Voor de graaf in Fig. 7.1 is dit inderdaad het geval.




Figuur 7.3: (a) Voorbeeldgraaf voor het A*-algoritme waarvoor A*-Propageer van de drie varianten de meeste vergelijkingen van $f$-waardes in *OPEN* uitvoert; (b) de bijbehorende zoekboom van het A*-algoritme na 3 stappen; in de volgende stap wordt het kortere pad naar knoop 4 ontdekt.
A*-Propageer voert 34 vergelijkingen uit voordat hij het kortste pad heeft gevonden; A*-Verwijder en A*-Vertrouw hebben aan 32 vergelijkingen genoeg.

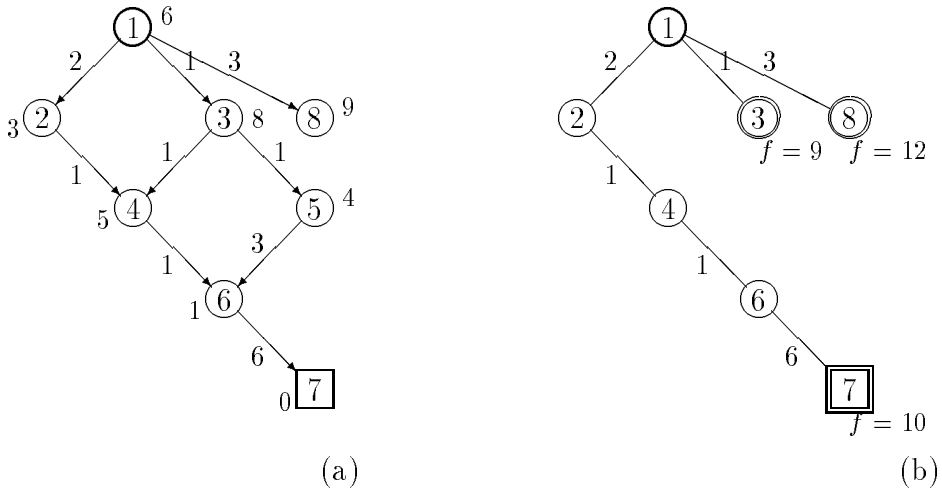
De graaf in Fig. 7.3 is echter een voorbeeld waarvoor A*-Propageer van de drie varianten de meeste vergelijkingen in *OPEN* uitvoert.

Na 3 stappen hebben we de zoekboom uit Fig. 7.3(b). In de volgende stap ontdekken we een korter pad naar knoop 4, namelijk via knoop 3. Bij A*-Propageer wordt die verbetering meteen naar de knopen 5, 6 en 7 gepropageerd, waardoor die met een lagere $f$-waarde (19 in plaats van 21) in *OPEN* komen te staan. Wanneer vervolgens knoop 8 geëxpandeerd wordt, worden onder meer de knopen 10, 11, 12 en 13 met een $f$-waarde 20 gegenereerd. Vanwege de verbeterde $f$-waardes van de knopen 5, 6 en 7, komen de knopen 10, 11, 12 en 13 nu helemaal achter in *OPEN* terecht. Ze moeten 'over de knopen 5, 6 en 7 heengetild worden'.

Bij de varianten A*-Verwijder en A*-Vertrouw hoeft dit niet, want daar staan die knopen op het moment dat knoop 8 ontwikkeld wordt nog met hun oude waarde 21 in *OPEN*. Weliswaar krijgen ze later alsnog de waarde 19, maar daar hebben de knopen 10, 11, 12 en 13 geen last meer van. Op het moment dat zíj geopend worden, hoeven ze minder ver naar achteren in *OPEN* gezet te worden. Het gevolg daarvan is dat we minder $f$-waardes met elkaar hoeven te vergelijken.

Voor dit voorbeeld is het aantal vergelijkingen in *OPEN* dat bij A*-Vertrouw wordt uitgevoerd nog even groot als bij A*-Verwijder. Ook voor het voorbeeld in Fig. 6.2 is dit nog het geval. Weliswaar voert A*-Verwijder daar een stap meer uit dan A*-Vertrouw, maar doordat *OPEN* op cruciale momenten leeg is, levert dit alles bij elkaar nog geen hoger aantal vergelijkingen van $f$-waardes in *OPEN* op.

We kunnen echter met behulp van Fig. 6.2 eenvoudig een graaf construeren waarvoor A*-Vertrouw echt minder vergelijkingen in *OPEN* uitvoert dan A*-Verwijder. Zo'n graaf is te zien in Fig. 7.4(a). Het enige verschil met de graaf uit Fig. 6.2 is de toegevoegde knoop 8 met de tak 1–8. Voor *OPEN* heeft dit echter tot gevolg dat die na elke stap minstens één knoop bevat. A*-Verwijder kan dan dus niet meer profiteren van een lege lijst *OPEN*, zodat deze variant nu wel meer vergelijkingen in die lijst




Figuur 7.4: (a) Voorbeeldgraaf voor het A*-algoritme waarvoor A*-Verwijder meer vergelijkingen in *OPEN* uitvoert dan A*-Vertrouw; het enige verschil met Fig. 6.2 is de toegevoegde knoop 8 met een tak er naar toe; (b) de bijbehorende zoekboom na 4 stappen in het A*-algoritme; in de volgende stap wordt het kortere pad naar $k_0 = 4$ gevonden (via knoop 3);
A*-Verwijder komt hier uiteindelijk uit op 14 vergelijkingen, terwijl A*-Vertrouw aan 11 vergelijkingen genoeg heeft.

uitvoert dan A*-Vertrouw.

## 7.3 Willekeurige Grafen

We hebben nu gezien dat het niet meevalt om iets algemeen geldigs te zeggen over de efficiëntie van de verschillende varianten van het A*-algoritme in vergelijking met elkaar. Het zou echter niet eerlijk zijn om puur op basis van enkele voorbeelden te concluderen dat het gemiddeld weinig uit zal maken welke variant we gebruiken. Sommige voorbeelden waren namelijk wel heel erg geconstrueerd. Als we er niet echt naar gezocht zouden hebben, waren we ze waarschijnlijk niet tegengekomen.

Daarom willen we de drie varianten ook aan de hand van een groter aantal willekeurige grafen met elkaar vergelijken. De betreffende grafen zijn als volgt gegenereerd.

Voor een gegeven waarde van $N$ worden willekeurig uit de knopen $1, 2, \ldots, N$ een startknoop en een bepaald aantal doelknopen gekozen. Het is in principe mogelijk dat de startknoop ook in het doel zit. Vervolgens krijgt elke mogelijke gerichte tak tussen twee knopen dezelfde kans om daadwerkelijk in de graaf opgenomen te worden. Als dat gebeurt, krijgt zo'n tak een willekeurig gewicht tussen een niet-negatieve ondergrens en een bovengrens. Nadat alle takken bepaald zijn, kan de functie $h^*$ voor elke knoop berekend worden. De heuristische functie $h$ ontstaat dan door elke $h^*$-waarde met een bepaald, al dan niet willekeurig getal te vermenigvuldigen. Om er voor te zorgen dat $0 \leq h(k) \leq h^*(k)$ voor elke knoop $k$, moet dit getal uit het interval $[0, 1]$ afkomstig zijn.

In alle grafen die we in deze paragraaf gebruiken, is het aantal doelknopen gelijk aan 1. Verder heeft elke tak in principe een kans van $\frac{3}{N-1}$ om ook echt in de graaf

voor te komen. Het verwachte aantal uitgaande takken van een knoop is dus 3. De gewichten op de takken zijn gehele getallen tussen (en inclusief) 1 en 10.

Om een verschil in prestatie te kunnen ontdekken tussen de varianten van het A*-algoritme, is het van belang dat er in het algoritme ooit een beter pad wordt gevonden naar een knoop die al eerder ontwikkeld is. We moeten dus proberen om het A*-algoritme te misleiden, zodat het niet direct het kortste pad zal vinden. Dit proberen we voor elkaar te krijgen door voor elke knoop $k$ op het kortste pad $h(k)$ gelijk te stellen aan $h^*(k)$, terwijl we voor de andere knopen $k$ $h^*(k)$ met een willekeurig getal tussen 0.0 en 0.1 vermenigvuldigen om $h(k)$ te krijgen.

De instanties die hierdoor ontstaan noemen we instanties van type 1. Omdat we ons niet al te zeer willen beperken, creëren we nog vier andere types instanties door steeds op één punt van de beschrijving van type 1 af te wijken.

Bij type 2 worden de $h$-waardes van knopen die niet op het optimale pad liggen vermenigvuldigd met een getal tussen 0.0 en 0.5. Type 3 onderscheidt zich van type 1 doordat de $h$-waardes van *alle* knopen met een willekeurig getal tussen 0.0 en 0.1 worden vermenigvuldigd. Het is te verwachten dat zowel bij instanties van type 2 als bij instanties van type 3 het algoritme minder vaak een verkeerd pad afloopt dan bij instanties van type 1. Het kortste pad wordt namelijk, relatief dan wel absoluut, aantrekkelijker gemaakt.

In de instanties van type 4 kunnen de takken alleen gewichten tussen (en inclusief) 5 en 10 aannemen. Het vijfde en laatste type kenmerkt zich door een hogere takdichtheid van de graaf. Elke tak heeft dan namelijk een kans $\frac{5}{N-1}$ om in de graaf opgenomen te worden.

Voor $N = 25$, $N = 50$ en $N = 100$ werden voor elk van de vijf types 10 grafen gegenereerd. Soms kwam het voor dat er voor een bepaalde instantie geen pad en dus ook geen kortste pad van de startknoop naar de eindknoop bestond. Dan werd die instantie vervangen door een nieuwe willekeurige instantie waarvoor er wel zo'n pad was. Al deze instanties hebben we opgelost met zowel A*-Propageer als A*-Verwijder en A*-Vertrouw. Daarbij hebben we het aantal bezochte takken en het aantal vergelijkingen in de lijst *OPEN* geteld. De resultaten zijn, gemiddeld over de 10 instanties voor elk type en elk aantal knopen, weergegeven in Tab. 7.1.

De algemene indruk die de resultaten geven is dat A*-Propageer en A*-Vertrouw even goed zijn als het aankomt op het aantal bezochte takken. Het kwam voor *geen enkele* van de 150 instanties voor dat A*-Propageer minder takken afliep dan A*-Vertrouw. Een enkele keer bezocht A*-Vertrouw er iets minder. Dit is aangegeven met '1)' in de kolom 'bijzonderheden' voor elke instantie waarin dat het geval was. Het beeld dat A*-Vertrouw het hier dus iets beter lijkt te doen dan A*-Propageer kan echter enigszins vertekend zijn. Bij het genereren van willekeurige grafen hielden we namelijk kunstmatig de $h$-waardes van de knopen op het kortste pad hoog. Als gevolg daarvan komen situaties als in Fig. 7.2, waarbij A*-Propageer onnodig veel werk doet, misschien onevenredig vaak voor.

Uit de tabel is verder af te leiden dat A*-Verwijder voor de 150 instanties steeds evenveel takken bezocht bij het ontwikkelen van knopen als A*-Vertrouw. Omdat A*-Vertrouw nooit een knoop ontwikkelt als A*-Verwijder dat niet ook doet, kunnen we concluderen dat bij A*-Verwijder steeds precies dezelfde knopen werden ontwikkeld als bij A*-Vertrouw. Een situatie als in Fig. 6.2, waarbij knoop 6 bij A*-Verwijder één keer meer werd ontwikkeld dan bij A*-Vertrouw, kwam dus kennelijk niet voor.

De variant A*-Verwijder bekeek voor elke instantie minstens zo veel takken als A*-Propageer (en uiteraard ook als A*-Vertrouw). Dit komt overeen met wat we al

| $N$ | type | aantal beter pad | A*-Propageer | | | A*-Verwijder | | | A*-Vertrouw | | bijz.heden |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | aantal tak1 | aantal tak | aantal vgln | aantal tak1 | aantal tak | aantal vgln | aantal tak | aantal vgln | |
| 25 | 1 | 4 | 40.1 | 47.3 | 72.0 | 47.3 | 48.7 | 78.1 | 47.3 | 77.8 | |
| 25 | 2 | 4 | 33.3 | 37.7 | 65.7 | 37.7 | 39.1 | 67.0 | 37.7 | 68.3 | 2) |
| 25 | 3 | 0 | 38.2 | 38.2 | 80.3 | 38.2 | 38.2 | 80.3 | 38.2 | 80.3 | |
| 25 | 4 | 5 | 41.5 | 43.4 | 77.6 | 43.4 | 43.8 | 79.7 | 43.4 | 79.2 | |
| 25 | 5 | 2 | 42.7 | 43.4 | 132.5 | 43.4 | 43.8 | 132.4 | 43.4 | 132.8 | 2) |
| 50 | 1 | 5 | 88.0 | 110.0 | 344.9 | 107.9 | 115.1 | 379.4 | 107.9 | 370.5 | 1),1),1),2) |
| 50 | 2 | 4 | 61.5 | 70.5 | 281.1 | 70.5 | 75.1 | 292.3 | 70.5 | 291.1 | |
| 50 | 3 | 0 | 77.4 | 77.4 | 300.7 | 77.4 | 77.4 | 300.7 | 77.4 | 300.7 | |
| 50 | 4 | 7 | 87.1 | 98.5 | 340.1 | 98.5 | 101.7 | 350.0 | 98.5 | 351.0 | |
| 50 | 5 | 2 | 102.0 | 108.9 | 451.3 | 108.9 | 110.9 | 459.2 | 108.9 | 455.7 | |
| 100 | 1 | 6 | 122.3 | 136.0 | 909.2 | 135.6 | 141.2 | 927.6 | 135.6 | 931.3 | 1) |
| 100 | 2 | 6 | 75.0 | 80.5 | 540.3 | 80.5 | 84.1 | 542.3 | 80.5 | 543.9 | 3),3) |
| 100 | 3 | 0 | 144.8 | 144.8 | 1083.3 | 144.8 | 144.8 | 1083.3 | 144.8 | 1083.3 | |
| 100 | 4 | 7 | 153.4 | 172.3 | 1130.8 | 172.3 | 179.7 | 1174.7 | 172.3 | 1175.7 | |
| 100 | 5 | 3 | 228.8 | 237.0 | 2064.3 | 237.0 | 239.7 | 2069.4 | 237.0 | 2072.2 | 2) |

Tabel 7.1: Resultaten van de drie varianten van het A*-algoritme voor 3 waardes voor $N$ en 5 types instanties. Voor elke combinatie ($N$, type) werden 10 instanties gegenereerd waarvoor inderdaad een pad van de startknoop naar de eindknoop bestond.
De kolom 'aantal beter pad' bevat het aantal instanties (van de 10) waarvoor het A*-algoritme ooit een beter pad naar een al eerder ontwikkelde knoop ontdekte. De gemeten verschillen tussen de varianten zijn het gevolg van die instanties.
In de kolom 'aantal tak' staat voor elke variant het totaal aantal takken dat die variant gemiddeld voor een instantie bezocht. Daarnaast werd voor A*-Propageer en A*-Verwijder het aantal takken steeds opgesplitst in twee aantallen. Het eerste aantal was het aantal takken dat bezocht werd bij het genereren van de kinderen van knopen die ontwikkeld werden. Het tweede aantal was het aantal takken dat bezocht werd bij het propageren van een verbetering van het pad naar een knoop (bij A*-Propageer) of bij het verwijderen van de nakomelingen van een knoop waarnaar een beter pad was gevonden (bij A*-Verwijder). Het eerste aantal is voor beide varianten in de kolom 'aantal tak1' opgenomen. Het tweede aantal kan dan eenvoudig uitgerekend worden door die kolom van de corresponderende kolom 'aantal tak' af te trekken.

intuïtief verwachtten. Erg groot worden de verschillen echter nooit: het maximale relatieve verschil voor één instantie tussen het aantal takken bezocht door A*-Vertrouw en A*-Verwijder bedroeg ongeveer 15.5%.

Voor wat betreft het aantal vergelijkingen in *OPEN*, blijkt A*-Propageer het beste te zijn. Waar we in paragraaf 7.1 de verwachting uitspraken dat A*-Verwijder in het algemeen minder vergelijkingen zou uitvoeren dan A*-Vertrouw, blijkt dat niet uit de cijfers in de tabel. De ene keer komt de verwachting uit, de andere keer juist niet. Alles bij elkaar genomen kunnen we niet zeggen dat de ene methode (van de twee) gemiddeld minder vergelijkingen nodig heeft dan de andere. Op een paar uitzonderingen na voerden A*-Verwijder en A*-Vertrouw voor elke instantie minstens zoveel vergelijkingen in *OPEN* uit als A*-Propageer. De instanties waarvoor alleen A*-Verwijder minder

vergelijkingen uitvoerde dan A*-Propageer zijn per geval aangegeven met '2)' in de kolom 'bijzonderheden'. Een '3)' in die kolom houdt in dat voor een instantie zowel A*-Verwijder als A*-Vertrouw minder vergelijkingen nodig had dan A*-Propageer. Het kwam geen enkele keer voor dat A*-Vertrouw wel, maar A*-Verwijder niet minder vergelijkingen uitvoerde dan A*-Propageer.

De verschillen tussen de aantallen vergelijkingen die de drie varianten uitvoerden werden soms niet alleen absoluut, maar ook relatief vrij groot. Het kwam voor dat er door A*-Verwijder en/of A*-Vertrouw voor een bepaalde instantie meer dan 30% meer vergelijkingen werden uitgevoerd dan door A*-Propageer. Wanneer we echter de gemiddelde verschillen bekijken, blijken die ook als het gaat om het aantal vergelijkingen in *OPEN* niet enorm groot te zijn.

Zoals opgemerkt, bevat Tab. 7.1 alleen de resultaten voor grafen waarin een pad van start naar doel bestond. De drie varianten zijn daarnaast ook toegepast op enkele willekeurige grafen waarin dat niet het geval was. Voor zulke grafen is de $h^*$-waarde van elke knoop die bereikbaar is vanaf de startknoop, eigenlijk oneindig groot. In de praktijk werd die waarde gelijk gesteld aan 500 000. Net als bij de andere knopen werd deze waarde met een willekeurig getal uit het interval $[0.0, 0.1]$ vermenigvuldigd om een $h$-waarde te krijgen. Zo kon het gebeuren dat knopen die bereikbaar waren vanaf de startknoop, $h$-waardes kregen die in verschillende mate 'oneindig groot' waren. Hierdoor kon het tevens voorkomen dat er in de loop van het A*-algoritme een beter pad naar een al eerder ontwikkelde knoop werd ontdekt.

In al zulke gevallen bleek A*-Vertrouw het kleinste aantal takken uit de graaf te bezoeken. Dat sluit dus aan bij de resultaten voor grafen waarin wel een pad van de startknoop naar de doelknoop bestond. Het kwam nu echter ook een keer voor dat A*-Propageer meer takken bekeek dan A*-Verwijder. Wat betreft het aantal vergelijkingen in de lijst *OPEN* komen de resultaten globaal overeen met die in Tab. 7.1: in het algemeen, maar niet altijd had A*-Propageer van de drie varianten de minste vergelijkingen nodig; A*-Verwijder en A*-Vertrouw waren gemiddeld ongeveer even goed.

Bij de grafen waarvoor er geen pad van de startknoop naar de doelknoop bestond, was het echter opvallend dat de verschillen tussen de drie varianten, zowel in aantallen bezochte takken als in aantallen vergelijkingen in *OPEN*, betrekkelijk groot konden worden, groter tenminste dan bij gelijksoortige grafen waarvoor wel zo'n pad bestond. Bij de bezochte takken was het verschil in aantal tussen A*-Vertrouw en A*-Verwijder een keer 19.7%; voor diezelfde instantie had A*-Verwijder 48.5% meer vergelijkingen in *OPEN* nodig dan A*-Propageer. Wellicht kunnen deze grotere verschillen als volgt verklaard worden.

In de bepaling van de $f$-waarde van een vanaf $s$ bereikbare knoop, vervult de (min of meer) oneindig grote $h$-waarde de hoofdrol. De $g$-waarde telt daarbij nauwelijks mee. Wanneer nu een bereikbare knoop $k_1$ toevallig een (ook relatief) erg hoge $h$-waarde heeft, zal die knoop pas heel laat ontwikkeld worden. Tegen die tijd kunnen zijn opvolgers al lang een keer ontwikkeld zijn; ze hebben immers een lagere $h$-waarde en de lengtes van de paden naar hen die op een bepaald moment als kortste paden te boek staan, zijn nauwelijks van belang. Wanneer nu eindelijk $k_1$ ontwikkeld wordt en er worden daarbij betere paden naar zijn opvolgers gevonden, is het goed mogelijk dat er relatief veel herstelwerk in de zoekboom te verrichten is. Daarbij kunnen de verschillen tussen de drie varianten ook relatief sterk uitkomen.

# Hoofdstuk 8

# Conclusies

In dit verslag hebben we drie varianten van het A*-algoritme beschreven: A*-Propageer, A*-Verwijder en A*-Vertrouw. Tevens hebben we de correctheid van deze varianten aangetoond. Met name voor de variant A*-Vertrouw was de correctheid nog niet helemaal triviaal, maar met behulp van enkele stellingen kon ze worden gereduceerd tot de correctheid van A*-Verwijder. We hebben daarbij bewezen dat elke knoop die bij A*-Vertrouw wordt geëxpandeerd ook bij A*-Verwijder wordt geëxpandeerd. In het algemeen geldt dit niet omgekeerd.

Om de efficiëntie van de drie varianten te vergelijken, hebben we gekeken naar het aantal takken uit een graaf dat elke variant afloopt en naar het aantal vergelijkingen dat moet worden uitgevoerd om knopen op de goede positie in de lijst *OPEN* te zetten. Samen vormen deze twee aantallen een goede maat voor de totale hoeveelheid werk die in het A*-algoritme voor eindige grafen wordt verricht.

Wellicht is het aantal bezochte takken een iets belangrijker criterium dan het aantal vergelijkingen in *OPEN*. Wanneer we gebruik maken van een adjacency-list representatie van de graaf, komt bij A*-Propageer en A*-Vertrouw elke tak die bezocht wordt overeen met een pointer die gevolgd moet worden en het aflopen van een pointer kost relatief veel tijd (bij A*-Verwijder geldt dit niet voor het verwijderen van knopen uit de zoekboom; de zoekboom kan voor een eindige graaf namelijk met behulp van arrays worden opgeslagen). De lijst *OPEN* kan daarentegen voor een eindige graaf in zijn geheel in een array worden opgeslagen.

Omdat elke knoop die bij A*-Vertrouw wordt geëxpandeerd, ook bij A*-Verwijder wordt geëxpandeerd, loopt A*-Verwijder voor elke instantie minstens zoveel takken af als A*-Vertrouw. Afgezien hiervan kunnen we, noch voor wat betreft het aantal bezochte takken, noch voor wat betreft het aantal vergelijkingen in *OPEN*, zeggen dat één variant voor elke instantie minstens zo goed is als een andere variant. Er zijn altijd wel grafen te construeren die zo'n uitspraak zouden weerleggen.

Wanneer we naar het gemiddelde gedrag van de varianten kijken, kunnen we echter wel verschillen ontdekken. Dan is A*-Verwijder, als het aankomt op het aantal bezochte takken, minder goed dan A*-Propageer en A*-Vertrouw. Van de laatste twee is A*-Vertrouw wellicht iets beter, maar die indruk kan vertekend zijn.

De variant A*-Propageer blijkt gemiddeld minder vergelijkingen in *OPEN* uit te voeren dan A*-Verwijder en A*-Vertrouw. De laatste twee blijken gemiddeld ongeveer even goed te zijn.

Voor alle verschillen die we ontdekt hebben, geldt dat ze niet spectaculair groot zijn. Bovendien treden ze pas op als er in de loop van het A*-algoritme ooit een keer een korter pad wordt gevonden naar een knoop die al eerder ontwikkeld is. Zo'n situatie

komt bij ècht willekeurige grafen minder vaak voor dan bij de grafen die wij gegenereerd hebben. Daarbij probeerden we tenslotte om het kortste pad aanvankelijk zo weinig mogelijk aantrekkelijk te maken.

Tot slot nog een waarschuwing: in dit verslag hebben we ons louter beziggehouden met het A*-algoritme voor eindige grafen die we van tevoren volledig kennen. In de praktijk hebben we bij het zoeken in een toestandsruimte, waarmee we dit verslag begonnen, echter vaak te maken met grafen die in de loop van het zoekproces worden opgebouwd. Aanvankelijk kennen we dan alleen de begintoestand en de eisen waaraan een eindtoestand moet voldoen. Bij het expanderen van toestanden worden vervolgens echt nieuwe, tot dan onbekende toestanden gecreëerd. Dit heeft onder meer tot gevolg dat we niet, zoals we in de algoritmes in dit verslag gedaan hebben, de status van elke mogelijke toestand (*Onbekend*, *Open* of *Gesloten*) eenvoudigweg in een array kunnen opslaan. We zullen (echte) lijsten van *Open* en *Gesloten* toestanden moeten bijhouden en daarin zullen we moeten nakijken of we een zojuist gegenereerde toestand al eerder zijn tegengekomen. Dit kan soms nog een behoorlijk tijdrovende bezigheid zijn.

De bereikte resultaten voor eindige, van tevoren bekende grafen zijn derhalve niet allemaal zonder meer over te dragen op anderssoortige grafen.

# Referenties

[Nau,Kumar,Kanal1984] Nau, D.S., Kumar, V. en Kanal, L.: General Branch and Bound, and Its Relation to A* and AO*, *Artificial Intelligence*, Vol. 23, 1984, blz. 29-58.

[Nilsson1982] Nilsson, N.J.: *Principles of Artificial Intelligence*, Springer, Berlin, 1982.

[Rich1983] Rich, E.: *Artificial intelligence*, McGraw-Hill Book Co., Singapore, 1983.

[Sprinkhuizen-Kuyper1990] Sprinkhuizen-Kuyper, I.G.: *Kunstmatige Intelligentie*, collegedictaat, magazijnuitgifte nr. 3, vakgroep Informatica, Rijksuniversiteit te Leiden, 1990.