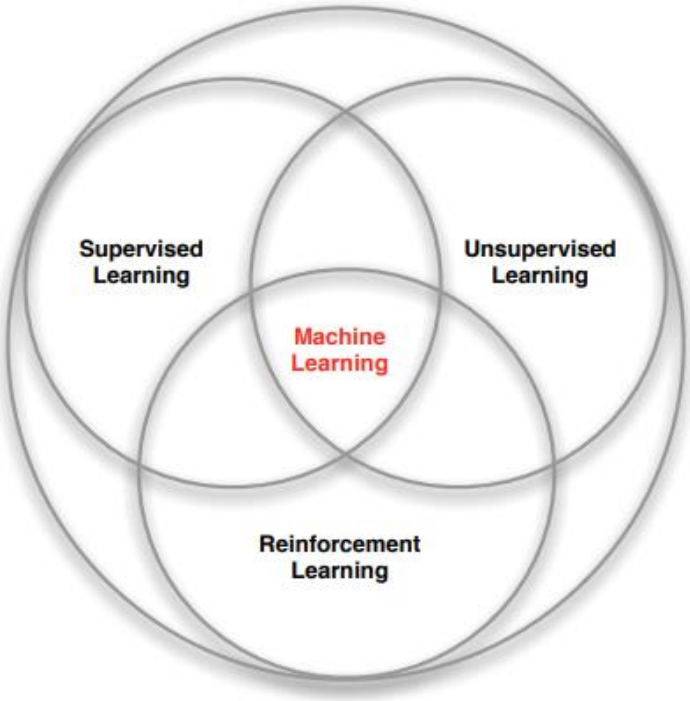
Reinforcement Learning

1. Wat is Reinforcement Learning?

1.1. Situering

Leren doen we heel vaak door interactie met onze omgeving. Neem als voorbeeld een kind dat speelt. Meestal heeft het kind geen leraar of trainer die zegt wat het moet doen, maar het leert met vallen en opstaan, letterlijk en figuurlijk. Het staat rechtstreeks in verbinding met de omgeving door te voelen, te bewegen, te ervaren. Maar het beïnvloedt ook die omgeving door bepaalde handelingen uit te voeren en zo ontstaat er een wisselwerking tussen beiden.

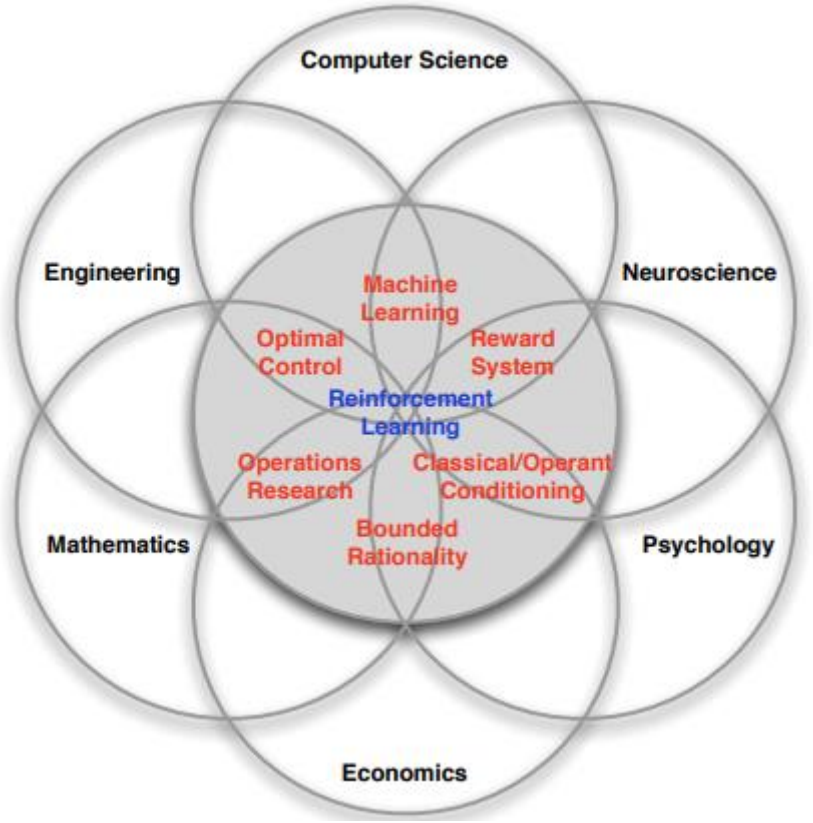
Reinforcement Learning (RL) neemt de idee van **leren door interactie** over en gaat dit wiskundig benaderen. Men implementeert hiervoor een softwareagent die zich in een bepaalde omgeving beweegt, en er leert om situaties om te zetten naar acties, met als doel een opeenvolging van beloningen te maximaliseren. Het is één van de drie fundamentele machine learning paradigma's, naast supervised en unsupervised learning. Net als vele andere woorden die eindigen op -ing, duidt RL het probleem aan van leren door interactie, verwijst het ook naar de oplossingsmethoden voor dat probleem, en is het tenslotte de discipline die dit probleem en de oplossingen bestudeert. Dit zijn drie verschillende zaken die men uit elkaar moet houden, ook al worden ze met dezelfde naam benoemd.



In RL is er geen supervisor die aangeeft welke acties goed en fout zijn. De agent krijgt enkel een beloning en het leren gaat via trial-and-error. De feedback over de genomen beslissingen krijgt hij niet onmiddellijk maar met vertraging. Op een bepaald tijdstip voert de agent een actie uit, maar het wordt pas later duidelijk of zijn beslissing om die actie uit te voeren goed of slecht was. Tijd speelt dus een cruciale rol bij RL. De beslissingen die de agent moet nemen zijn sequentieel, en na elke actie krijgt hij een beloning, maar het is de som van alle beloningen die van tel is en moet geoptimaliseerd worden. De data zijn bijgevolg niet i.i.d., wat staat voor "independent and identically distributed". In het Nederlands spreekt men van een aselecte steekproef, een aantal onafhankelijke trekkingen uit

dezelfde verdeling, wat hier dus niet het geval is. Veel andere machine learning methodes gaan wel uit van i.i.d. data. Maar in RL beïnvloedt de agent door het nemen van acties de omgeving waarin hij opereert en dus ook de data die worden gegenereerd.

Wanneer we Reinforcement Learning, afgekort tot RL, proberen te situeren binnen de wetenschappen, dan zit het in de doorsnede van verschillende disciplines, zoals op de figuur is te zien. In elk van die disciplines is er namelijk een tak die zich bezighoudt met het bestuderen van de problemen die ook in RL worden opgelost. Deze problemen gaan in het algemeen over het nemen van beslissingen, **decision making** in het Engels, waarbij men probeert om de processen te beschrijven en te optimaliseren om tot die beslissingen te komen.



In de computerwetenschappen valt het oplossen van dit soort van beslissingsproblemen onder de noemer van machine learning, in de ingenieurswetenschappen spreekt men over optimal control, en in de wiskunde bestudeert men deze optimaliseringsproblemen in de besliskunde, ook wel operationeel onderzoek of operations research genoemd.

In de psychologie staat de verzameling leerprocessen, waarbij een organisme zich aanpast aan de omgeving, bekend onder de naam conditionering. Men maakt hierin o.a. onderscheid tussen de klassieke stimulus-respons conditionering die door Pavlov werd ontdekt, en de operante conditionering, die o.a. werd bestudeerd door Thorndike en Skinner. Bij operante conditionering wordt een respons in een bepaalde context gevolgd door een bekrachtiger of een bestraffer, respectievelijk reinforcer of punisher in het Engels. In de neurowetenschappen worden de neurale structuren bestudeerd die aan de basis liggen van conditionering en deel uitmaken van het zogenaamde beloningssysteem met dopamine als belangrijkste neurotransmitter.

In de economie worden beslissingsproblemen uitvoerig bestudeerd in de speltheorie. In deze theorie gaat men er meestal van uit dat de spelers volledig rationeel handelen, wat in realiteit niet klopt.

Daarom worden ook modellen ontwikkeld die een begrensde rationaliteit veronderstellen waarin individuen niet naar optimale oplossingen zoeken, maar beslissingen nemen die hen voldoening geven.

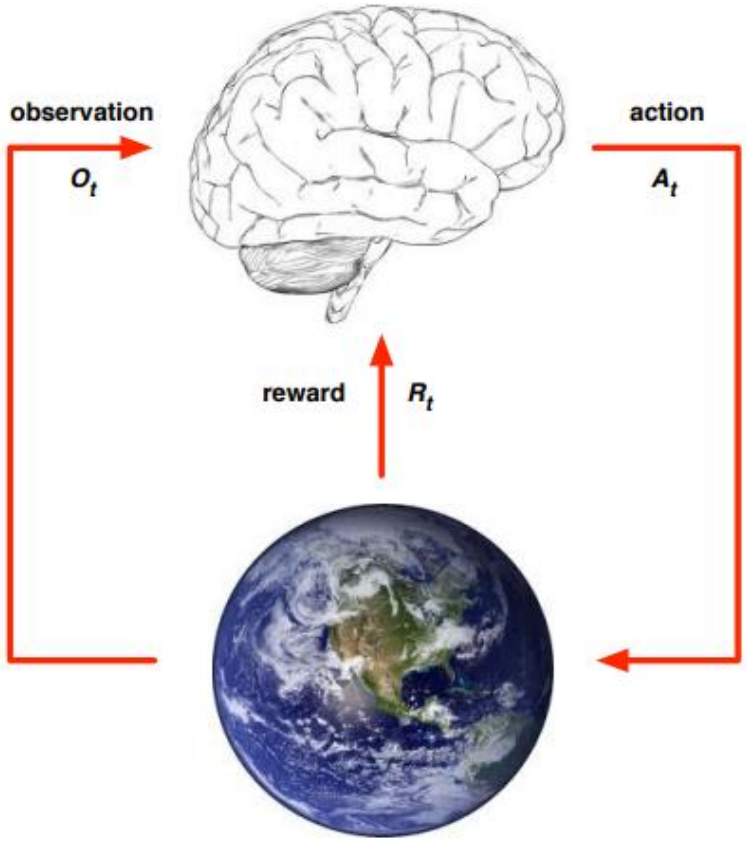
Reinforcement Learning kent veel toepassingen, maar meestal denken mensen aan allerhande spelletjes waarin menselijke spelers door artificiële intelligentie worden verslagen. Zo denken we aan bordspelen zoals Backgammon en schaken, en oude Atari videospelletjes die een RL agent in een paar dagen tijd kan leren waarbij hij even goed of zelfs beter wordt dan menselijke spelers. Een ander bekend voorbeeld is AlphaGo, die in 2016 Lee Sedol versloeg, één van 's werelds beste Go-spelers. Ook AlphaGo gebruikt RL, naast andere machine learning methodes.

Maar RL kan ook gebruikt worden in zelfrijdende auto's of bij het besturen van modelbouwhelikopters die allerlei stunts uithalen. Met RL kan men ook industriële robotten besturen of zelfs mensachtige robotten leren wandelen. In de financiële wereld kan RL toegepast worden om bijvoorbeeld een investeringsportfolio te beheren, en ingenieurs doen beroep op RL om bijvoorbeeld het aansturen van een elektriciteitscentrale te optimaliseren.

1.2. Het Reinforcement Learning probleem

Agent en Environment

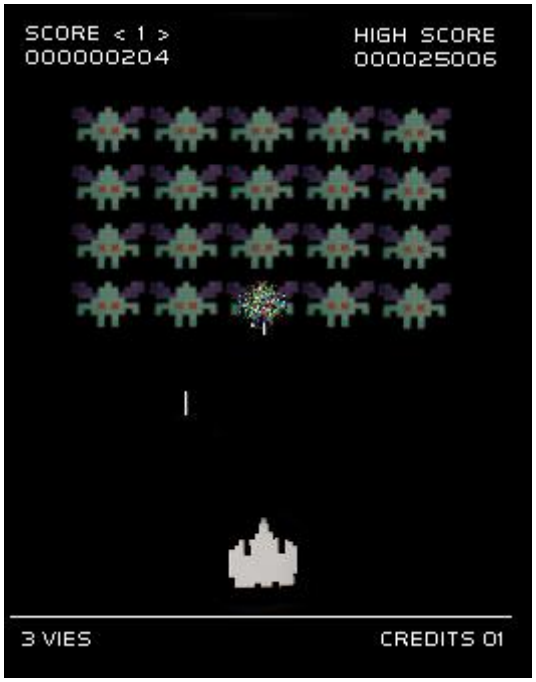
Reinforcement Learning formaliseert de interactie tussen een softwareagent (Engels: agent) en een omgeving (Engels: environment) door de wisselwerking tussen beiden op te delen in opeenvolgende tijdstappen *t*. Deze interactie kan episodisch zijn, wat wil zeggen dat ze op een gegeven moment stopt, of continu, waarbij ze geen einde kent.



Op de figuur wordt de agent voorgesteld door een brein en de omgeving door en wereldbol. In een bepaalde tijdstap *t* voert de agent een actie A_t (Engels: action) uit die een invloed heeft op de omgeving. Als gevolg van die actie ontvangt de agent een beloning R_t (Engels: reward) van de

omgeving en neemt het de nieuwe toestand van de omgeving waar via observatie O_t (Engels: observation). In de volgende tijdstap t+1 onderneemt de agent o.b.v. observatie O_t en beloning R_t een nieuwe actie A_{t+1}, ontvangt daarvoor een nieuwe beloning R_{t+1} en doet een nieuwe observatie O_{t+1}. En zo gaat de interactie tussen agent en omgeving door tot het uiteindelijke doel is bereikt.

Neem bijvoorbeeld het Atari computerspel Space Invaders waarin de speler een laserkanon bestuurt die aliens moet kapotschieten. De agent is hier de speler van het spel die het laserkanon van links naar rechts beweegt en uiteraard ook kan schieten. Op elk tijdstip t gaat de agent dus één van deze drie acties uitvoeren: naar links bewegen, naar rechts bewegen of schieten. De omgeving bestaat uit het computerscherm dat onderaan het laserkanon laat zien, en bovenaan rijen van aliens die in een steeds sneller tempo naar beneden komen en zelf ook kunnen schieten. Een observatie bestaat uit een momentopname van het scherm die de positie van het kanon en de aliens weergeeft en de laserstralen afgevuurd door het kanon en de aliens. De rewards zijn de punten die de speler krijgt na elke actie. Het doel van het spel is een zo hoog mogelijk eindscore halen. Het spel stopt wanneer het kanon drie keer is vernietigd, of wanneer de aliens de onderkant van het scherm bereiken.



Reward

De beloning, of reward in het Engels, wordt met R_t aangeduid, en is niets meer of minder dan een getal dat de softwareagent van de omgeving ontvangt op tijdstip t. Het is een feedback signaal die een indicatie geeft van hoe goed de actie is die de agent op tijdstip t heeft uitgevoerd. De taak van de agent bestaat erin om de som van alle R_t, d.i. de cumulatieve reward, te maximaliseren. Het maximaliseren van de totale of cumulatieve reward is het doel van RL, dat is gebaseerd op de reward hypothese. Die stelt dat elk doel in RL kan geformuleerd worden als de maximalisatie van de verwachte cumulatieve reward. Deze hypothese is echter controversieel en dus is niet iedereen het daar mee eens.

De reward bestaat altijd uit één getal omdat de agent op een gegeven tijdstip t één beslissing moet nemen. Ook al zijn er meerdere mogelijke acties, deze acties moeten tegenover elkaar afgewogen worden. Rewards kunnen wel negatief zijn om acties te bestraffen, bijvoorbeeld wanneer een wandelende robot valt, of wanneer een softwareagent een modelbouwhelikopter laat crashen, of wanneer veiligheidsvoorschriften worden overtreden in een door RL aangestuurde

elektriciteitscentrale. Een ander voorbeeld is wanneer een agent een bepaald traject in een zo kort mogelijke tijd moet doorlopen. In dat geval zal men per tijdstap een negatieve reward geven, en het maximaliseren van de cumulatieve reward zal ervoor zorgen dat er zo weinig mogelijk stappen worden genomen. Bij het spelen van Backgammon worden geen rewards gegeven na elke zet, maar krijgt de agent enkel een positieve reward als hij het spel wint, en een negatieve als hij verliest.

RL is dus een proces van opeenvolgende beslissingen die moeten genomen worden, sequential decision making in het Engels, met als doel acties te selecteren die de totale toekomstige reward zo groot mogelijk maken. Hieruit volgt dat deze acties gevolgen hebben op lange termijn en dat de reward soms met vertraging wordt ontvangen, waardoor het beter kan zijn om de onmiddellijke reward op te offeren in functie van een grotere winst op lange termijn. Denk bijvoorbeeld aan financiële investeringen die vaak pas na maanden of jaren echt opbrengen.

History en state

De **geschiedenis** H_t (Engels: history) op tijdstip t is de reeks van observaties, acties en beloningen die reeds hebben plaatsgevonden:

$$H_t = A_1, O_1, R_1, \dots, A_t, O_t, R_t$$

Geschiedenis H_t bevat dus alle waarneembare variabelen t.e.m. tijdstap t. Er kunnen nog andere variabelen in de omgeving meespelen, om bijvoorbeeld een reward te berekenen o.b.v. een actie, maar deze zijn niet waarneembaar voor de agent. RL gaat over algoritmes die binnenin de agent worden uitgevoerd en dus zijn enkel de variabelen relevant die door de agent waarneembaar zijn. We denken hier bijvoorbeeld aan de stroom aan sensomotorische gegevens die een robot ontvangt en op basis waarvan hij acties uitvoert.

Wat er op tijdstip t+1 gebeurt, hangt volledig af van de geschiedenis H_t. Eerst en vooral zal de agent een actie selecteren en uitvoeren o.b.v. die geschiedenis. Het algoritme van de agent moet dus de geschiedenis H_t afbeelden op een actie A_{t+1}. En ook de omgeving selecteert observaties en rewards o.b.v. de geschiedenis. Maar hoe groter t wordt, hoe meer data geschiedenis H_t zal bevatten, en daarom is het niet nuttig om o.b.v. de geschiedenis te bepalen wat er in de volgende tijdstap moet gebeuren. Daarom gaan we de geschiedenis H_t vervangen door de **toestand** S_t (Engels: state), die de geschiedenis samenvat en dus de nodige informatie bevat om te kunnen bepalen wat er op tijdstip t+1 moet gebeuren. Formeel gezien is de toestand een functie van de geschiedenis:

$$S_t = f(H_t)$$

We kunnen drie toestanden onderscheiden. De toestand van de omgeving S_t^e is de interne voorstelling van de omgeving. Dit is de informatie binnenin de omgeving die wordt gebruikt om de rewards en de observaties te genereren. In het geval van de Atari games is dat de emulator. Vaak is de toestand van de omgeving niet zichtbaar voor de agent, en als die toch zichtbaar is, dan is die informatie niet altijd relevant voor de agent. De toestand van de agent S_t^a is de interne voorstelling van de agent en bestaat uit de informatie binnenin de agent die wordt gebruikt om de acties te kiezen. Dit is de informatie die wordt gebruikt door de RL algoritmes. De toestand van de agent moet de geschiedenis samenvatten, maar we kunnen die toestand zelf definiëren. M.a.w. kunnen we de functie f zelf kiezen:

$$S_t^a = f(H_t)$$

Tenslotte is er een wiskundige definitie van toestand, namelijk de informatietoestand, ook wel **Markov toestand** genoemd, die alle nuttige informatie uit de geschiedenis bevat. Formeel is een toestand S_t Markov als en slechts als:

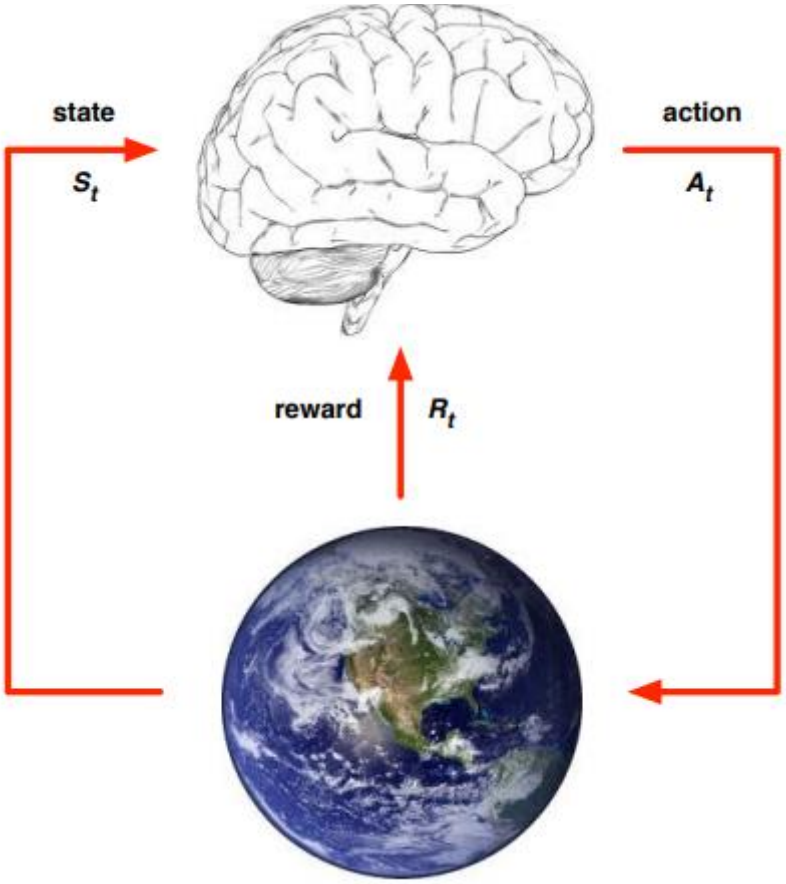
$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, S_2, \ldots, S_t]$$

$\mathbb{P}$ staat voor de kans of de probabiliteit. Een toestand S_t is dus Markov als de kans op de volgende toestand S_{t+1}, gegeven de toestand S_t, gelijk is aan de kans op de volgende toestand S_{t+1}, gegeven alle voorgaande toestanden $S_1, S_2, \ldots, S_t$. Dit betekent dat je alle voorgaande toestanden t.e.m. S_{t-1} kan weggooien omdat S_t een statistiek is die voldoende informatie bevat over de toekomst. Dit kan samengevat worden als: de toekomst $(t + 1 \rightarrow \infty)$ is onafhankelijk van het verleden $(1 \rightarrow t - 1)$ als het heden $(1t)$ is gegeven. In het voorbeeld van de helikopter is de positie en de snelheidsvector op een bepaald tijdstip een Markov toestand, omdat men op basis daarvan de volgende positie en snelheidsvector kan berekenen. Men hoeft hier dus niet alle voorgaande posities en snelheden bij te houden. Enkel de positie bijhouden volstaat niet om de volgende positie te kunnen berekenen, en dus is de positie geen Markov toestand.

Per definitie zijn zowel de toestand van de omgeving S_t^e als de geschiedenis H_t Markov, omdat ze bepalen wat er op tijdstip t+1 gaat gebeuren. Alleen is S_t^e niet altijd gekend, en H_t is niet nuttig omdat het teveel informatie bevat. Maar ze tonen wel aan dat er altijd een Markov toestand is. In het speciale geval dat de omgeving rechtstreeks waarneembaar is voor de agent, vallen toestand van omgeving en agent samen en is die bovendien Markov. Dit is een volledig waarneembare omgeving (Engels: fully observable environment) waarvoor geldt dat:

$$O_t = S_t^e = S_t^a$$

Formeel noemen we dit een **Markov beslissingsproces** (Engels: Markov Decision Process, afgekort MDP). In het geval van een MDP kunnen we observatie O_t dus vervangen door toestand S_t:

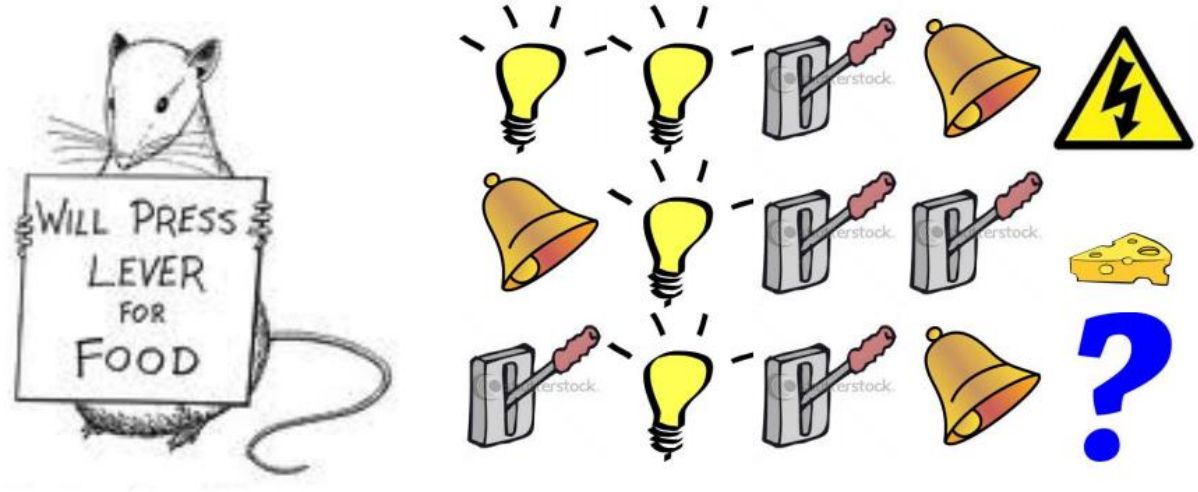


In het andere geval hebben we een gedeeltelijk waarneembare omgeving (Engels: Partially observable environment), die niet rechtstreeks door de agent kan waargenomen worden. Een

voorbeeld hiervan is een robot met een camera die zijn absolute locatie niet kent, of een agent die poker speelt en enkel de kaarten ziet die zichtbaar op tafel liggen. In deze gevallen is de toestand van de omgeving niet gelijk aan die van de agent, en dus moet de agent zelf zijn toestand bepalen, door bijvoorbeeld de volledige geschiedenis bij te houden, door gebruik te maken van een recurrent neuraal netwerk, of door een Bayesiaanse benadering toe te passen en de toestand van de omgeving af te leiden uit een kansverdeling. Formeel noemen we dit een gedeeltelijk waarneembaar Markov beslissingsproces (Engels: partially observable Markov decision process, afgekort POMDP).

Voorbeeld: experiment met rat

Om het belang van de toestand van de agent te illustreren, nemen we als voorbeeld een denkbeeldig experiment waarbij een agent een rat is die licht kan zien knipperen, een belletje kan horen rinkelen, en zelf een hendel kan bedienen. Wanneer hij twee keer het licht ziet knipperen, dan aan de hendel trekt en tenslotte het belletje hoort, krijgt hij een stroomstoot. Wanneer hij het belletje hoort, het licht één keer ziet knipperen en dan twee keer aan de hendel trekt, krijgt hij kaas. De vraag is nu wat er zal gebeuren in het laatste geval.



De toestand van de omgeving die bepaalt of er een stroomstoot of een stuk kaas wordt gegeven, is niet gekend. De toestand van de rat is een functie van de geschiedenis. Stel dat die toestand enkel de drie voorgaande gebeurtenissen bijhoudt, dan zal de rat denken dat hij een stroomstoot zal krijgen, omdat de drie voorgaande gebeurtenissen in de laatste reeks gelijk zijn aan die van de eerste reeks. Als de toestand van de rat enkel het aantal gebeurtenissen bijhoudt, dan zal de rat denken dat hij kaas zal krijgen, omdat het aantal keren dat aan de hendel is getrokken, het licht heeft geknipperd en de bel heeft gerinkeld, in de tweede en laatste reeks gelijk zijn. Maar als de toestand van de rat uit de hele geschiedenis bestaat, dan kan de rat niet voorspellen wat er zal gebeuren.

1.3. De RL agent

Tijdens het leren kan de agent beroep doen op:

- een beleid (Engels: policy): een functie van het gedrag van de agent;
- een waarde-functie (Engels: value function): een functie die aangeeft hoe goed elke toestand of actie is;
- een model (Engels: model): een voorstelling die de agent maakt van de omgeving.

De agent kan één of meer van de drie bovenstaande componenten bevatten. Hij kan ook nog andere componenten bevatten, maar deze drie zijn de belangrijkste.

Policy

Een beleid, in het Engels policy, geeft het gedrag van de agent weer. De policy wordt door de agent opgebouwd. De agent leert dus de policy. Formeel gezien is het een functie π die de toestand afbeeldt op een actie. Een **deterministische** policy wordt als volgt gedefinieerd:

$$a = \pi(s)$$

Functie π zet in dit geval eenvoudigweg toestand *s* om in actie *a*. Bij een **stochastische** policy is π de kans dat de actie *a* wordt gekozen gegeven de toestand *s*:

$$\pi(a|s) = \mathbb{P}[A = a \mid S = s]$$

In dit geval zijn *A* en *S* toevalsvariabelen die respectievelijk alle mogelijk acties en toestanden weergeven. Stochastische policies zijn nuttig omdat ze een element van willekeurigheid bevatten en op die manier beter de omgeving kunnen verkennen.

Value function

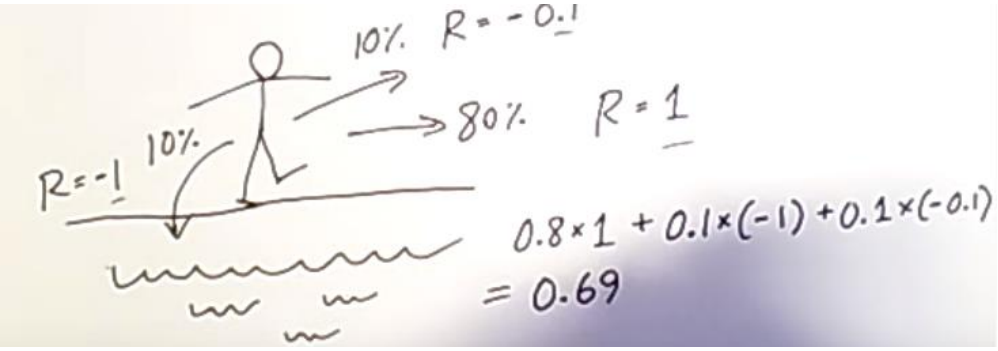
De waarde-functie, in het Engels value function, is een voorspelling van de te verwachten toekomstige reward, en wordt gebruikt om de waarde van de toestanden te evalueren. M.a.w. de value function geeft aan hoe goed of hoe slecht een toestand is. Op die manier kan de agent een keuze maken o.b.v. de toekomstige reward die hij mag verwachten. De value function *v* in toestand *s* wordt in het geval van een stochastische policy π als volgt gedefinieerd:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s]$$

De functie is dus afhankelijk van de policy π, die als subscript wordt weergegeven, omdat deze de rewards *R* bepaalt. De discount factor γ is een getal tussen 0 en 1. $\mathbb{E}$ is de verwachtingswaarde (Engels: expected value). Herinner dat de verwachtingswaarde van een discrete toevalsvariabele *X* als een gewogen gemiddelde kan gezien worden waarbij de gewichten gelijk zijn aan de kansen:

$$\mathbb{E}[X] = \sum_i \mathbb{P}[X = x_i].x_i$$

In de value function is de toevalsvariable *R* die de verschillende rewards weergeeft. Laten we dit met een eenvoudig voorbeeld verduidelijken.



Op de figuur zien we een agent, voorgesteld door een mannetje, die een pad volgt, voorgesteld door de horizontale lijn waarop het mannetje loopt. De agent bevindt zich in toestand $S_t = s$. We hebben een stochastische policy π die zegt dat er 80% kans is dat de agent rechtdoor zal gaan en het pad zal blijven volgen. In dit geval krijgt de agent een reward *R* gelijk aan 1. Maar er is ook 20% kans dat de agent van het pad af geraakt, waarbij er 10% kans is dat hij rechts in het water valt, en 10% kans dat hij links tegen een wand botst. Valt hij in het water, dan wordt hij gestraft met een reward *R* van -1,

botst hij tegen de wand, wat minder erg is, dan krijgt hij een reward *R* van -0.1. In dit geval zal de value function voor de state *s* gelijk zijn aan:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} | S_t = s] = \sum_{i=1}^{3} \mathbb{P}[R_i | S_t = s].R_i$$

met R_i de 3 mogelijke rewards die de agent in toestand *s* kan ontvangen. De figuur geeft de toestand *s* schematisch weer, samen met de berekening van de value function die gelijk is aan 0.69.

Voor de eenvoud hebben we in dit voorbeeld de discount factor γ gelijk gesteld aan 1. Dit is zeer kortzichtig, omdat we dan enkel rekening houden met de onmiddellijke reward R_{t+1} en niet met de toekomstige rewards na *t*+1. Nemen we de discount factor gelijk aan 1, dan wegen alle rewards evenveel mee. Meestal wordt echter de discount factor gelijk gesteld aan een getal tussen 0 en 1, waardoor de toekomstige rewards minder en minder gaan meewegen. Maar omdat ze nog meewegen, wordt niet alleen de volgende stap geëvalueerd, maar het volledige traject dat na toestand *s* met de grootste waarschijnlijkheid zal gevolgd worden.

In het geval van een deterministische policy ligt dat traject vast en is er in elke toestand slechts één mogelijke actie en dus één mogelijk reward. De value function in toestand *s* bij een deterministische policy π wordt dus vereenvoudigd tot:

$$v_\pi(S_t = s) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

Model

De agent kan ook een model van de omgeving opbouwen. Dit model is een voorstelling van de omgeving die voorspelt wat de omgeving zal doen in een volgende tijdstap. Het model bestaat uit twee onderdelen: een model van de transities (Engels: transitions) dat de volgende toestand voorspelt en dus de dynamiek van de omgeving weergeeft, en een model van de beloningen (Engels: rewards), dat de volgende, onmiddellijke reward voorspelt. Het transitiemodel wordt met de letter $\mathcal{P}$ aangeduid, het reward model met de letter $\mathcal{R}$. De formele definities zijn:

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s,\ A_t = a]$$

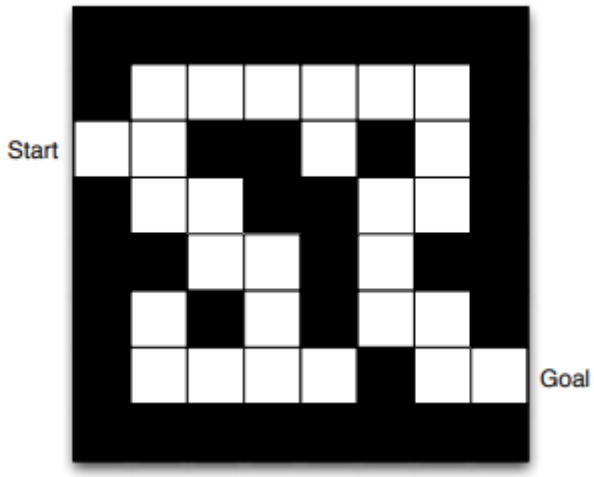
$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \mid S_t = s,\ A_t = a]$$

Het transitiemodel $\mathcal{P}^a_{ss'}$ geeft de kans weer dat de agent in de volgende tijdstap *t*+1 in toestand s' komt, gegeven dat in de huidige tijdstap *t* de toestand *s* is en actie *a* wordt gekozen. Het reward model $\mathcal{R}^a_s$ is de verwachte beloning R_{t+1} nadat in tijdstap *t* actie *a* werd gekozen in toestand *s*. Een model is echter geen vereiste en in veel algoritmen maakt de agent geen gebruik van een model om beslissingen te nemen.

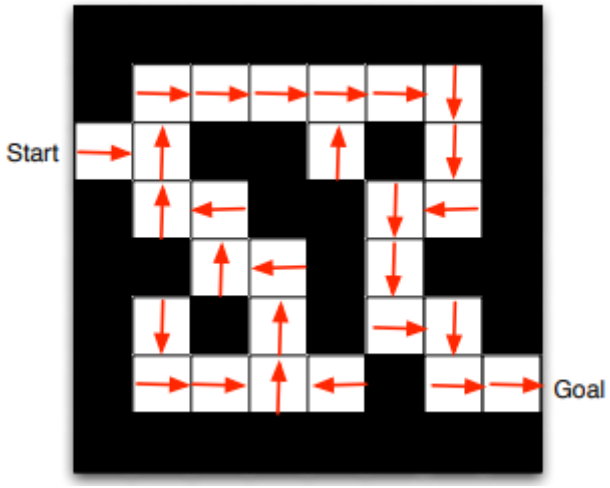
Voorbeeld: doolhof

Vooraleer we verder gaan illustreren we eerst de concepten policy, value function en model met een voorbeeld van een agent die vanaf de startpositie (Start) de kortste weg door een eenvoudig doolhof moet vinden naar de uitgang (Goal).

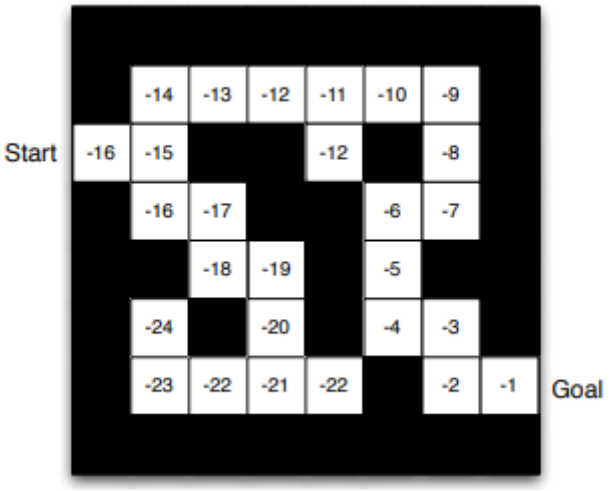
De toestanden zijn de vakjes waarin de agent zich bevindt. De mogelijke acties zijn een vakje naar boven, naar onder, naar links of naar rechts opschuiven. Zoals eerder uitgelegd geven we per stap een reward van -1 zodat de agent bij het maximaliseren van de cumulatieve reward effectief het kortste traject zal volgen.



Hieronder wordt een deterministische **policy** $\pi(s)$ weergegeven a.d.h.v. rode pijltjes die aangeven in welke richting de agent een vakje moet opschuiven.



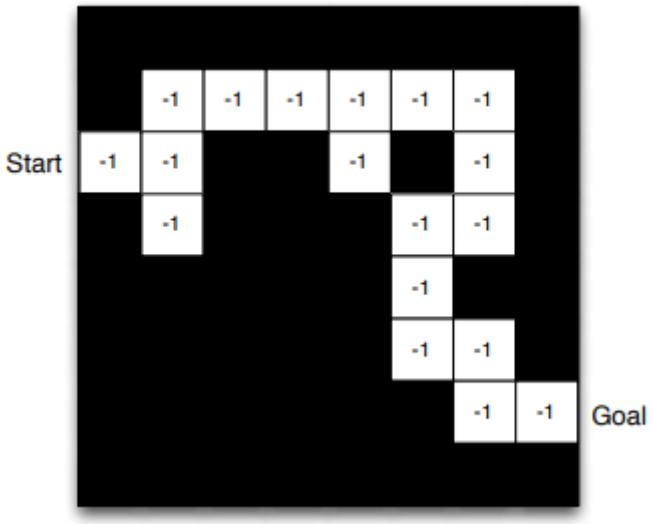
De **value function** $v_\pi(s)$ wordt hieronder weergegeven door de cijfers in elk vakje.



Wanneer de agent in het vakje links van de uitgang staat, dan geeft die positie de toestand weer en die krijgt een waarde van -1, omdat de policy de agent naar rechts laat gaan, daarbij een reward gelijk aan -1 krijgt, en daarna zijn doel heeft bereikt. Wanneer de agent in het vakje rechts van de start staat, dan krijgt deze toestand de waarde -16, omdat de agent vanuit dat vakje nog 16 stappen moet zetten

wanneer hij de rode pijltjes volgt. Na die 16 stappen bereikt hij zijn doel en krijgt hij een totale reward van -16 omdat hij per stap een reward van -1 krijgt. Op die manier geven de waarden in elk vakje de totale reward weer, en de absolute waarde van die reward is gelijk aan het aantal stappen dat de agent nog moet nemen volgens de policy vooraleer hij de uitgang bereikt.

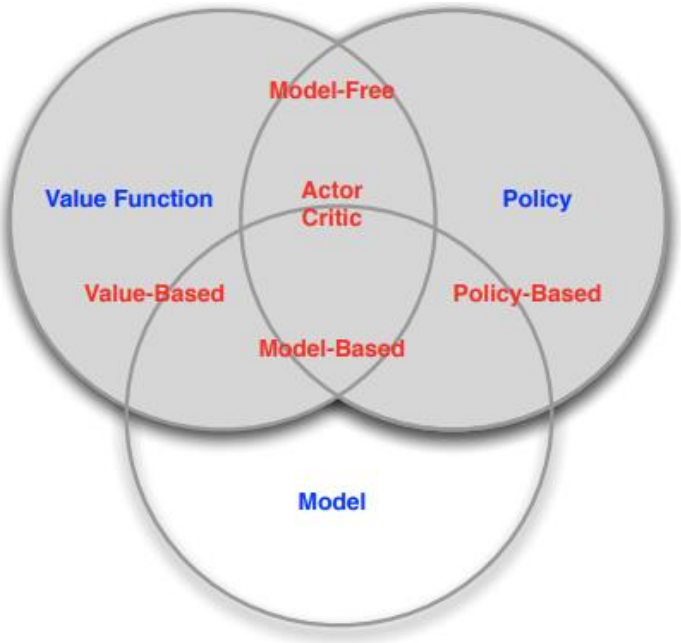
Tenslotte kan de agent een **model** van het doolhof opbouwen. Zoals hieronder is te zien hoeft dat model niet volledig te zijn.



Het transitiemodel $\mathcal{P}_{ss'}^{a}$, wordt voorgesteld door de vakjes. Elk vakje stelt een toestand voor. De aangrenzende vakjes zijn de volgende toestanden die mogelijk zijn vanuit de huidige toestand, d.i. het vakje waarin de agent momenteel staat. De cijfers in de vakjes zijn het reward model $\mathcal{R}_{s}^{a}$. Per stap wordt inderdaad een onmiddellijke reward van -1 gegeven aan de agent.

Taxonomie

Aan de hand van de componenten waaruit een RL agent bestaat, kunnen we nu volgende categorieën van algoritmen onderscheiden:



Een value-based algoritme maakt gebruik van een value function en niet van een policy. Een policy-based algoritme maakt gebruik van een policy en heeft geen value function. Een actor critic algoritme maakt gebruik van een policiy en een value function. Al deze algoritmen zijn ofwel model-based of model-free. In het eerste geval maken ze gebruik van een policy en/of een value function en een model, in het laatste geval passen ze een policy en/of value function toe maar hebben ze geen model.

1.4. Problemen in RL

Hierboven hebben we het algemene probleem geformaliseerd dat in reinforcement learning wordt bestudeerd. In dit hoofdstuk bespreken we enkele fundamentele subproblemen binnen RL.

Leren versus plannen

In het Engels spreekt men van learning en planning. Het onderscheid tussen beiden is van cruciaal belang binnen de wetenschap van “sequential decision making”. In het geval van reinforcement learning is de omgeving initieel niet gekend en leert de agent die omgeving kennen door interactie. Bij planning is de omgeving wel gekend. De agent beschikt over een model van de omgeving, en past dit toe zonder dat er een interactie is met die omgeving. In beide gevallen zal de agent de policy verbeteren. Beide problemen zijn gelinkt aan elkaar aangezien een model om te plannen kan opgebouwd worden door te leren.

Nemen we als voorbeeld een agent die een Atari spel speelt. In het geval van reinforcement learning kent de agent de regels van het spel niet en moet hij die leren door interactief het spel te spelen. Door de joystick te bedienen kan hij acties kiezen, waarna hij pixels (observaties) en scores (rewards) ontvangt. Via trial-and-error zal hij zijn policy alsmaar verbeteren tot hij een optimale policy heeft gevonden. Bij planning daarentegen kent de agent de regels van het spel en beschikt hij dus over een model van het spel. Hij kan de emulator bevragen via queries en zo voor elke actie de volgende toestand en score vinden. De optimale policy zou hij dan via een tree search kunnen bepalen zonder het spel interactief te spelen.

Exploreren versus exploiteren

In het Engels spreekt men van exploration en exploitation. Doordat RL leren is door trial-and-error moet de agent een goede balans vinden tussen beiden, want hij kan enkel een goede policy vinden door de omgeving te verkennen zonder daarbij te veel beloningen te verliezen. Bij het verkennen of exploreren verzamelt de agent informatie over de omgeving, terwijl hij bij het exploiteren die informatie gaat gebruiken om de totale reward te maximaliseren. Om tot een goede policy te komen moet de agent dus de twee gaan toepassen, want als hij altijd de actie kiest die de grootste onmiddellijke beloning oplevert, zou het kunnen dat hij een traject over het hoofd ziet waar initieel de beloningen kleiner zijn maar waar de totale beloning op het einde veel groter is.

Nemen we als voorbeeld het kiezen van een restaurant om op zondag te gaan eten. Exploiteren betekent in dit geval dat we naar ons favoriete restaurant gaan, exploreren dat we een nieuw restaurant uitproberen. Als we elke zondag zouden exploiteren, dan zouden we altijd naar hetzelfde restaurant gaan, waar we wel zeker zijn van het lekkere eten, maar waardoor we misschien wel restaurants zouden missen waar het eten nog lekkerder is. Wanneer we echter elke weekend zouden exploreren, dan zouden we altijd een nieuw restaurant kiezen, waardoor we het risico lopen om altijd minder lekker eten te krijgen dan in ons favoriete restaurant.

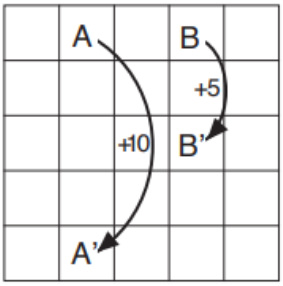
Een ander voorbeeld is het boren van olie. Exploiteren wil in dit geval zeggen dat er op meest rendabele gekende locatie wordt geboord, terwijl exploreren betekent dat er ook op onbekende locaties wordt geboord. In het voorbeeld van online adverteren via banners, waarmee machine

learning één van zijn eerste grote successen binnenhaalde, worden bij exploiteren de meest succesvolle advertenties getoond, terwijl exploreren betekent dat er nieuwe advertenties worden getoond. Een laatste voorbeeld is het spelen van spelletjes waarin een zet, waarvan men denkt dat die de beste is, gelijk aan exploiteren, terwijl het uitproberen van een andere zet gelijk is aan exploreren.

Voorspellen versus regelen

In het Engels spreekt men van prediction en control. Bij prediction is er een policy gegeven en die moet worden geëvalueerd. Men gaat in dit geval dus voorspellen hoeveel de totale reward zal zijn van de gegeven policy. Bij control gaat men proberen de beste policy te vinden. Men gaat dus de totale reward maximaliseren. Samengevat kan men stellen dat prediction de toekomst evalueert, terwijl control de toekomst optimaliseert. In RL is het vaak zo dat men iteratief policies gaat evalueren om tot de optimale policy te komen. We moeten dus het prediction probleem kunnen oplossen om het control probleem te kunnen oplossen.

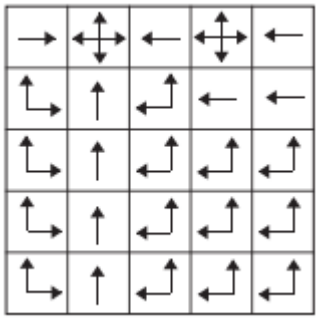
Als voorbeeld nemen we een gridworld waarin een agent vanuit het vakje waarin hij staat naar het aangrenzende vakje boven, onder, links of rechts kan bewegen. Per stap krijgt de agent een reward van -1. Enkel als hij in vakje A of B komt, krijgt hij een beloning van respectievelijk 10 en 5, en wordt hij naar de respectievelijke vakjes A' en B' geteleporteerd. Stel dat de uniforme random policy is gegeven waarin de kans om naar boven, onder, links of rechts te bewegen telkens 25% is. Prediction betekent dat we voor deze policy de value function gaan bepalen, zoals die hieronder is weergegeven (rechts).



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Control betekent dat we de optimale policy gaan zoeken die de grootste totale reward oplevert. Deze wordt hieronder weergegeven (rechts) samen met de bijhorende value function (links).

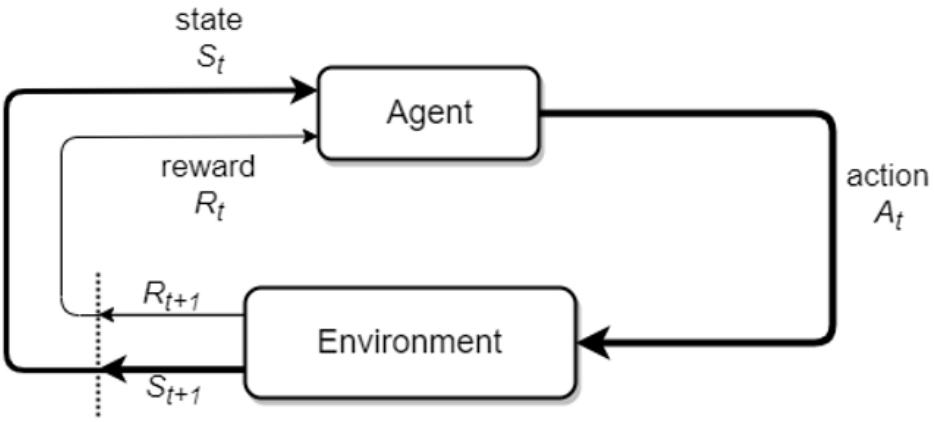
22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7



In de volgende hoofdstukken gaan we zien hoe we dit soort problemen kunnen oplossen.

1.5. Samenvatting

We sluiten dit eerste hoofdstuk af met een ander veel gebruikt schema waarmee het reinforcement learning probleem wordt samengevat:



Op de figuur zien we de agent die zich op tijdstip *t* in toestand S_t bevindt en een beloning R_t van de omgeving heeft ontvangen. Daarop gaat de agent een actie A_t uitvoeren die een invloed heeft op de omgeving. Zo komen we op het volgende tijdstip *t*+1 in de nieuwe toestand S_{t+1} waarin de omgeving de agent een beloning R_{t+1} geeft. Merk op dat de reward die volgt op actie A_t in sommige handboeken als R_t wordt aangeduid, en in andere als R_{t+1}.

Er is dus een continue interactie tussen agent en omgeving. Het doel van de agent is om via trial-and-error te leren de acties te kiezen en uit te voeren die een maximale totale beloning opleveren. Wanneer de omgeving volledig zichtbaar is, dan stelt S_t zowel de toestand van de agent als de toestand van de omgeving voor. Wanneer toestand van agent en omgeving samenvallen, dan spreken we van een Markov beslissingsproces, in het Engels Markov Decision Process (MDP). In het volgende hoofdstuk gaan we hier dieper op in.

2. Markov Decision Processes

In dit hoofdstuk bespreken we in detail Markov Decision processes (MDPs), die we gebruiken om de omgeving in reinforcement learning te beschrijven. Maar eerst bespreken we de eenvoudiger Markovketens of -processen en de Markov Reward processen. Elk van deze systemen bouwt verder op de voorgaande door er meer complexiteit aan toe te voegen.

In een MDP is de omgeving volledig waarneembaar en het proces wordt volledig gekarakteriseerd door de huidige toestand van de omgeving, die in dit geval ook de toestand van de agent is. Bijna alle RL problemen kunnen worden geformaliseerd als MDP. Zo zijn multi-armed bandits (zie verder) MDPs met één toestand, en problemen met een gedeeltelijk waarneembare omgeving kunnen worden omgezet in MDPs. Er bestaan ook continue MDPs, die bijvoorbeeld gebruikt worden in optimal control, een discipline binnen de ingenieurswetenschappen.

2.1. Markov Processes

Markoveigenschap

We zagen reeds dat een toestand S_t Markov is als en slechts als:

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, S_2, \dots, S_t]$$

We hebben dit samengevat als de toekomst die onafhankelijk is van het verleden als het heden is gegeven, omdat de toestand alle relevante informatie van de geschiedenis bevat. Als gevolg hiervan kunnen we de kans $\mathcal{P}_{ss'}$ dat een Markov toestand *s* overgaat naar toestand s' definiëren als:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

In het Engels is $\mathcal{P}_{ss}$, de state transition probability.

State transition matrix

We kunnen nu alle kansen $\mathcal{P}_{ss}$, van alle toestanden *s* en alle daaropvolgende toestanden *s'* in een matrix $\boldsymbol{\mathcal{P}}$ weergeven:

$$\boldsymbol{\mathcal{P}} = \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}$$

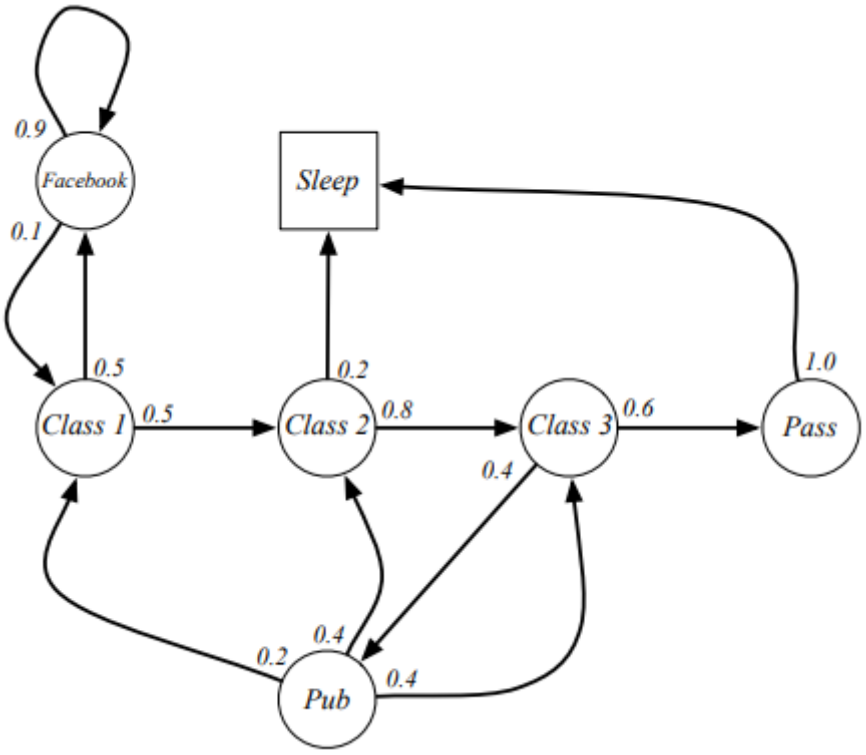
Deze matrix noemt men in het Engels de state transition matrix. In dit geval zijn er *n* toestanden die genummerd zijn van 1 tot *n*. Element $\mathcal{P}_{ij}$ in deze matrix is de kans dat toestand *i* zal overgaan in toestand *j*. Dit betekent dat de rijen overeenkomen met de huidige toestanden S_t en de kolommen met de volgende toestanden S_{t+1}. De som van elke rij is om die reden gelijk aan 1.

Definitie

Een **Markovproces** is een geheugenloos, stochastisch proces dat bestaat uit een opeenvolging van toevallige toestanden met de Markoveigenschap. Formeel wordt een Markovproces gedefinieerd als een tuple $\langle \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{P}} \rangle$ met $\boldsymbol{\mathcal{S}}$ een verzameling van toestanden en $\boldsymbol{\mathcal{P}}$ een state transition matrix. Een Markovproces in discrete tijd met een eindige of aftelbaar oneindige toestandsverzameling noemt men een **Markovketen**.

Voorbeeld: student

Hieronder wordt een Markovketen weergegeven van een student die 3 lessen (Engels: class) moet doorlopen om te kunnen slagen. Daarna kan de student slapen, wat de eindtoestand is. Maar in elke les zijn er afleidingen, zoals op Facebook zitten, op café gaan of te vroeg gaan slapen. De toestanden worden dus met cirkels weergegeven, de overgangen met pijltjes. De getallen bij de pijltjes geven de kansen weer dat een student van de ene naar de andere toestand overgaat. Zo is er 50% kans dat de student van de eerste les naar Facebook gaat, en eens hij/zij op Facebook zit, is er 90% kans dat hij/zij op Facebook blijft.



De state transition matrix $\mathcal{P}$ is in dit geval gelijk aan:

$$\mathcal{P} = \begin{array}{c} \\ C1 \\ C2 \\ C3 \\ Pass \\ Pub \\ FB \\ Sleep \end{array} \begin{array}{c} \begin{array}{ccccccc} C1 & C2 & C3 & Pass & Pub & FB & Sleep \end{array} \\ \begin{bmatrix} & 0.5 & & & & 0.5 & \\ & & 0.8 & & & & 0.2 \\ & & & 0.6 & 0.4 & & \\ & & & & & & 1.0 \\ 0.2 & 0.4 & 0.4 & & & & \\ 0.1 & & & & & 0.9 & \\ & & & & & & 1 \end{bmatrix} \end{array}$$

De ontbrekende elementen zijn gelijk aan nul, wat wil zeggen dat er geen overgang is gedefinieerd. Uit deze Markovketen kunnen we verschillende samples nemen. Wanneer we Class 1 als begintoestand nemen, dan zijn er van daaruit verschillende episodes mogelijk:

- C1 C2 C3 Pass Sleep
- C1 FB FB C1 C2 Sleep
- C1 C2 C3 Pub C2 C3 Pass Sleep
- …

In het geval van een Markovketen bestaan samples dus uit willekeurige reeksen van opeenvolgende toestanden: S_1, S_2, S_3, …, S_T.

2.2. Markov Reward Process

Definitie

Een Markov beloningsproces, Markov Reward Process (MRP) in het Engels, is een Markovketen met waarden. Deze waarden noemen we beloningen (Engels: rewards). De formele definitie van een MRP is een tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ met $\mathcal{S}$ een (eindige) verzameling van toestanden, $\mathcal{P}$ een state transition matrix, $\mathcal{R}$ een reward functie, en γ een discount factor waarvoor geldt dat $0 \leq \gamma \leq 1$. We zagen reeds dat de reward functie aangeeft hoeveel beloning de agent kan verwachten als hij in een bepaalde toestand is. Dit wordt als volgt gedefinieerd:

$$\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$$

Merk op dat het hier om de onmiddellijke reward gaat. In RL is het echter de cumulatieve reward die moet gemaximaliseerd worden. Vandaar dat we een nieuwe grootheid gaan definiëren: de return.

Return

De return G_t is de totale beloning vanaf tijdstap t, waarbij men de discount factor γ in rekening brengt:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_{t+T} = R_{t+1} + \gamma G_{t+1}$$

De return kan dus op een recursieve manier worden berekend. Merk op dat het hier niet om een verwachtingswaarde gaat, omdat de return voor één episode wordt berekend, en dus niet de verwachte cumulatieve reward is van alle mogelijke episodes. Op tijdstap $t+T$ eindigt de episode en bereikt de agent de eindtoestand. In veel gevallen is het echter zo dat T oneindig is:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (0 \leq \gamma \leq 1)$$

Discount factor γ maakt de uitdrukking wiskundig eenvoudiger en zorgt ervoor dat de return eindig blijft, ook als er een oneindig aantal termen zijn. Bij een discount factor gelijk aan nul telt enkel de

onmiddellijke reward mee, en hoe groter γ wordt, hoe meer belang er wordt gehecht aan de toekomstige beloningen. Het is mogelijk om γ gelijk aan 1 te nemen, wanneer alle episodes eindigen, maar over het algemeen neemt men een factor kleiner dan 1. Er is namelijk onzekerheid over deze toekomstige beloningen en dus wil men ze niet volledig laten mee tellen. In reële voorbeelden kan de discount factor de voorkeur van dieren en mensen voor onmiddellijke beloningen beter in rekening brengen, of de grotere interesten bij onmiddellijke financiële beloningen.

Value function

De waarde-functie *v(s)* (Engels: value function) van een MRP is de verwachte return vertrekkende van toestand *s*:

$$v(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s] = \mathbb{E}[G_t \mid S_t = s]$$

De waarde-functie van toestand *s* kan gezien worden als de lange-termijn waarde van die toestand.

Bellman vergelijking

Deze vergelijking is fundamenteel in RL en ze toont aan dat de value function *v(s)* recursief kan ontbonden worden in twee delen, nl. de onmiddellijke reward R_{t+1} en de gereduceerde waarde van de volgende toestand $\gamma v(S_{t+1})$:

$$\begin{aligned} v(s) &= \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \end{aligned}$$

In de laatste stap wordt de wet van de totale verwachting toegepast, die zegt dat $\mathbb{E}(X) = \mathbb{E}(\mathbb{E}(X \mid Y))$, met *X* en *Y* toevalsvariabelen. Zie https://en.wikipedia.org/wiki/Law_of_total_expectation voor meer uitleg. Omdat de verwachtingswaarde lineair is, geldt ook dat $\mathbb{E}(X+Y) = \mathbb{E}(X)+\mathbb{E}(Y)$ en $\mathbb{E}(aX) = a\mathbb{E}(X)$, met *a* een contante. Wanneer we deze eigenschappen toepassen, kunnen we *v(s)* in functie van $\mathcal{R}_s$ en $\mathcal{P}_{ss'}$ schrijven:

$$\begin{aligned} v(s) &= \mathbb{E}[R_{t+1} \mid S_t = s] + \gamma \mathbb{E}[v(S_{t+1}) \mid S_t = s] \\ &= \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} \, v(s') \end{aligned}$$

Wanneer er *n* toestanden *s* zijn, dan hebben we *n* vergelijkingen met *n* onbekende waarden *v(s)*. In matrixvorm:

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

De *n* toestanden werden voor het gemak enkel met hun index weergegeven. Dit stelsel kan algebraïsch worden opgelost:

$$\boldsymbol{v} = \boldsymbol{\mathcal{R}} + \gamma \boldsymbol{\mathcal{P}} \boldsymbol{v}$$

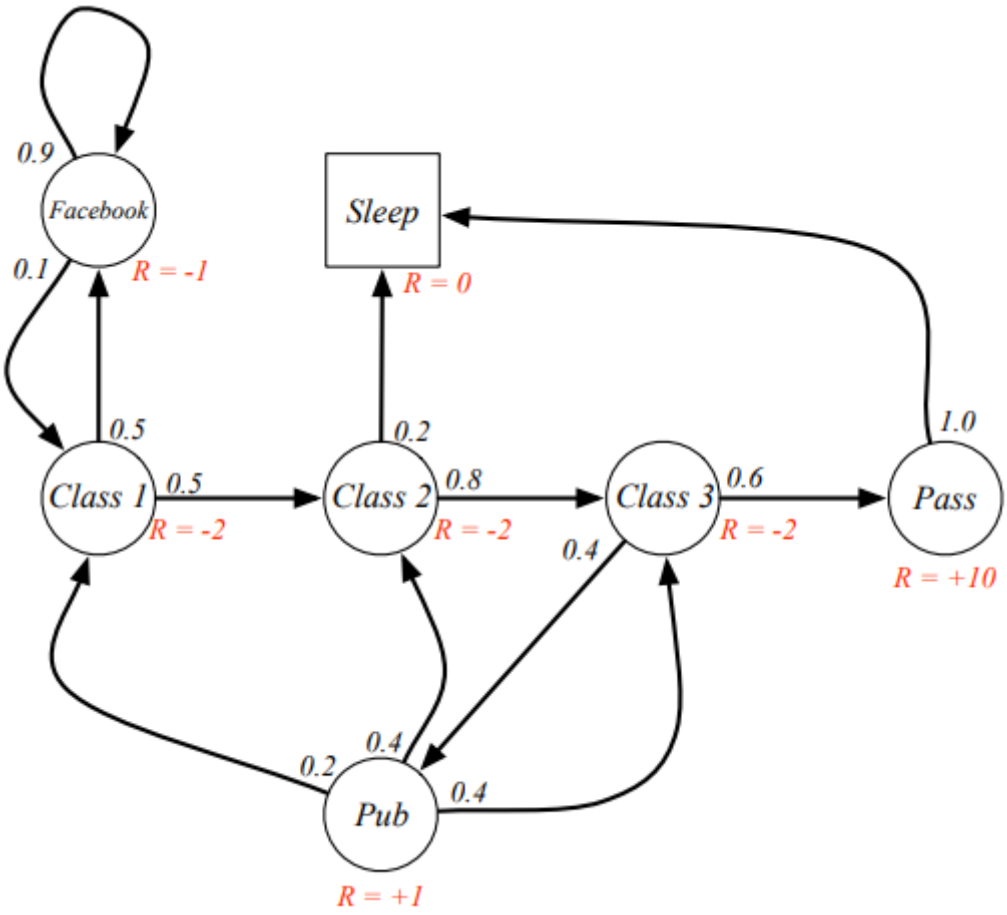
$$[\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}] \boldsymbol{v} = \boldsymbol{\mathcal{R}}$$

$$\boldsymbol{v} = [\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}]^{-1} \boldsymbol{\mathcal{R}}$$

met $\boldsymbol{I}$ de eenheidsmatrix die enen op de hoofddiagonaal bevat en nullen elders. Bij kleine MRPs kan dit matrixstelsel rechtstreeks worden opgelost, bij grote MRPs zijn er iteratieve oplossingsalgoritmen nodig, zoals dynamic programming, Monte-Carlo evaluation en Temporal-Difference learning.

Voorbeeld: student

Nemen we opnieuw het voorbeeld van de student en maken we van de Markovketen een Markov reward process door aan elke toestand een beloning te hangen (rood).



Voor willekeurige samples of episodes startend in “Class 1” en eindigend in “Sleep” kunnen we nu de return berekenen, waarbij we de discount factor gelijk nemen aan 1/2:

C1 C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 10 * \frac{1}{8}$	=	−2.25
C1 FB FB C1 C2 Sleep	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16}$	=	−3.125
C1 C2 C3 Pub C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 1 * \frac{1}{8} - 2 * \frac{1}{16} \ldots$	=	−3.41

We gaan nu de value function berekenen met verschillende discount factors door het stelsel van Bellman vergelijkingen op te lossen. We schrijven daarvoor een eenvoudig R script.

We creëren eerst een vector met de namen van de toestanden. Die gaan we gebruiken bij het benoemen van rijen en kolommen van de matrices die we gaan opstellen.

```
# states
states = c("C1", "C2", "C3", "Pass", "Pub", "FB", "Sleep")
nstates = length(states)
```

De rewards slaan we op in een kolomvector:

```
# reward vector R
R = matrix(c(-2, -2, -2, 10, 1, -1, 0), nrow=nstates, ncol=1,
           dimnames=list(rownames=states))
print(R)
```

De eenheidsmatrix definiëren we met functie `diag`:

```
# identity matrix I
I = diag(1, nrow=nstates, ncol=nstates)
print(I)
```

We gaan de oplossing voor 11 discount factors berekenen:

```
# discount factors gamma
gamma = seq(0, 1, length.out=11)
print(gamma)
```

De state transition matrix initialiseren we eerst als een matrix met allemaal nullen, en daarna vullen we de gegeven kansen in:

```
# state transition matrix P
P = matrix(0, nrow=nstates, ncol=nstates,
           dimnames=list(rownames=states, colnames=states))
P["C1", "C2"] = 0.5
P["C1", "FB"] = 0.5
P["C2", "C3"] = 0.8
P["C2", "Sleep"] = 0.2
P["C3", "Pass"] = 0.6
P["C3", "Pub"] = 0.4
P["Pass", "Sleep"] = 1
P["Pub", "C1"] = 0.2
P["Pub", "C2"] = 0.4
P["Pub", "C3"] = 0.4
P["FB", "C1"] = 0.1
P["FB", "FB"] = 0.9
#P["Sleep", "Sleep"] = 1
print(P)
print(rowSums(P))
```

Hierboven werd element `P["Sleep","Sleep"]` op 1 gezet, wat kan gezien worden als de agent dit tot in het oneindige blijft overgaan van “Sleep” naar “Sleep”, en dus in die toestand blijft. Maar bij γ=1 wordt het stelsel daardoor singulier, wat betekent dat niet alle vergelijkingen lineair onafhankelijk zijn, en er geen unieke oplossing is. Daarom laten we dat element op 0 staan. In de laatste lijn checken we of de som van alle rijen effectief gelijk is aan 1, wat hier het geval is.

Het stelsel kunnen we nu oplossen met de functie `solve`. We doen dit voor alle discount factors. Daarvoor gebruiken we een lus. De oplossing van het stelsel is de value function voor alle toestanden. Voor elke discount factor wordt de oplossing in een kolom van matrix `v` opgeslagen. Vooraleer de lus wordt doorlopen, wordt deze matrix eerst gealloceerd:

```
# solving the system for different gamma
v = matrix(nrow=nstates, ncol=length(gamma),
           dimnames=list(rownames=states, colnames=gamma))
for (i in 1:length(gamma)) {
  v[, i] = solve(I - gamma[i]*P, R)
}

# solutions
print(v)
```

Wanneer we het script runnen, dan zien we de oplossingen in de console. Bemerk dat bij γ=0 de value function gelijk is aan de reward vector. De waarden voor “Pub” en “Sleep” zijn in alle gevallen

gelijk aan hun reward, respectievelijk 10 en 0. Dit is omdat er slechts één overgang mogelijk is vanuit deze toestanden.

```
> print(v)
        colnames
rownames  0        0.1        0.2        0.3        0.4        0.5        0.6
   C1    -2 -2.1616019 -2.3395907 -2.5257034 -2.7150379 -2.9081572 -3.1167464
   C2    -2 -2.1093834 -2.1193318 -2.0310305 -1.8429994 -1.5500691 -1.1420428
   C3    -2 -1.3672920 -0.7458237 -0.1292937  0.4906268  1.1248272  1.7874109
   Pass  10 10.0000000 10.0000000 10.0000000 10.0000000 10.0000000 10.0000000
   Pub    1  0.8177009  0.6772039  0.5892189  0.5664173  0.6241359  0.7808788
   FB    -1 -1.1226550 -1.2765754 -1.4736590 -1.7321899 -2.0825597 -2.5804452
   Sleep  0  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000
        colnames
rownames        0.7         0.8        0.9           1
   C1    -3.3805907 -3.82965824 -5.0127289 -12.5432099
   C2    -0.6022709  0.09147104  0.9426553   1.4567901
   C3     2.4959449  3.26792350  4.0870212   4.3209877
   Pass  10.0000000 10.00000000 10.0000000  10.0000000
   Pub    1.0569460  1.46226093  1.9083924   0.8024691
   FB    -3.3422739 -4.66561664 -7.6376084 -22.5432099
   Sleep  0.0000000  0.00000000  0.0000000   0.0000000
```

2.3. Markov Decision Process

In de Markov Reward Processes die we in het vorige punt bespraken werden random samples genomen en op basis daarvan konden we de value function berekenen. Er was dus geen agent die het proces doorliep en beslissingen nam die tot acties leidde. Bij een Markov Decision Process is dat wel het geval, en dit formalisme is daarom de basis van reinforcement learning.

Definitie

Een Markov beslissingproces, Markov Decision Process (MDP) in het Engels, is een Markov Reward Process met beslissingen. Het is een omgeving waarin alle toestanden de Markoveigenschap bezitten. Formeel wordt een MDP gedefinieerd als een tuple $\langle \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{R}}, \gamma \rangle$ met $\boldsymbol{\mathcal{S}}$ een (eindige) verzameling van toestanden, $\boldsymbol{\mathcal{A}}$ een (eindige) verzameling van acties, $\boldsymbol{\mathcal{P}}$ een state transition matrix, $\boldsymbol{\mathcal{R}}$ een reward functie, en γ een discount factor waarvoor geldt dat $0 \leq \gamma \leq 1$. In dit geval hangen de state transition probability en de reward function ook af van de gekozen actie *a*:

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}[S_{t+1} = s' \mid S_t = s,\ A_t = a]$$

$$\mathcal{R}_{s}^{a} = \mathbb{E}[R_{t+1} \mid S_t = s,\ A_t = a]$$

Policy

Een beleid π, in het Engels policy, geeft de kans dat op tijdstip *t* actie *a* wordt gekozen, gegeven dat op dat tijdstip de toestand *s* is. De policy is dus een kansverdeling die als volgt wordt gedefinieerd:

$$\pi(a|s) = \ \mathbb{P}[A_t = a \mid S_t = s]$$

De policy legt het gedrag van de agent volledig vast. Let er wel op dat dit de definitie van een **stochastisch** beleid is. Eerder zagen we ook al dat een policy ook deterministisch kan zijn, maar in RL werkt men meestal met stochastische policies omdat men op die manier de agent toelaat om te exploreren.

Omdat de toestanden in een MDP Markov zijn, hangt de policy enkel af van de huidige toestand S_t, en niet van de geschiedenis H_t. Ook dit hebben al eerder uitvoerig besproken. Het gevolg is dat MDP policies **stationair** zijn, d.w.z. niet tijdsafhankelijk zijn. Om dezelfde reden komen de rewards niet voor in de definitie, omdat deze impliciet in rekening worden gebracht door de toestand S_t.

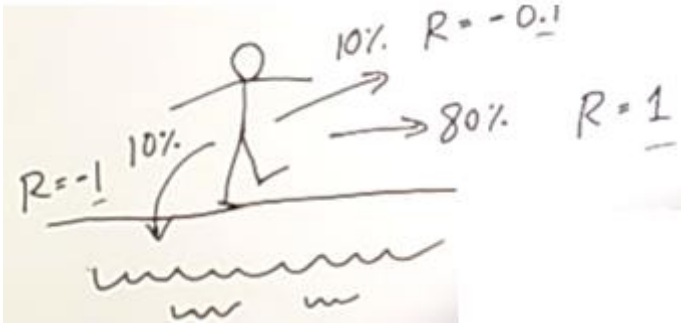
Eens een policy π is vastgelegd, kan elke MDP $\langle \boldsymbol{S}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{R}}, \gamma \rangle$ met behulp van π gereduceerd worden tot een Markov proces en een Markov Reward proces. De reeks van toestanden S_1, S_2, … die volgens de policy worden gegenereerd is inderdaad een Markov proces $\langle \boldsymbol{S}, \boldsymbol{\mathcal{P}^{\pi}} \rangle$, en de reeks van toestanden en rewards S_1, R_1, S_2, R_2, … is een Markov Reward proces $\langle \boldsymbol{S}, \boldsymbol{\mathcal{P}^{\pi}}, \boldsymbol{\mathcal{R}^{\pi}}, \gamma \rangle$. De state transition matrix en reward function worden als volgt bepaald:

$$\mathcal{P}^{\pi}_{ss'} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}^{a}_{ss'}$$

$$\mathcal{R}^{\pi}_{s} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}^{a}_{s}$$

In het eerste geval wordt de kans dat de agent vanuit toestand *s* in toestand *s'* terechtkomt na het uitvoeren van actie *a* vermenigvuldigd met de kans dat actie *a* in toestand *s* effectief wordt uitgevoerd onder policy π. In het tweede geval wordt de verwachte beloning die de agent in toestand *s* krijgt na uitvoeren van actie *a* vermenigvuldigd met de kans dat actie *a* in toestand *s* effectief wordt uitgevoerd onder policy π. In beide gevallen nemen we dan de som van alle bekomen waarden bij alle mogelijke acties. Wat we hier dus eigenlijk doen is de state transition probabilities en de verwachte beloningen in de huidige toestand uitmiddelen a.d.h.v. de vastgelegde policy.

Om dit te verduidelijken nemen we het eenvoudige voorbeeld dat we eerder zagen van het mannetje dat op een pad loopt en momenteel op positie H staat, wat dus de huidige toestand is.



Stel dat we nu een policy π hebben die zegt dat er 95% kans is dat het mannetje de actie "vooruitgaan" (V) zal kiezen, en 5% kans dat hij de actie "blijven staan" (B) selecteert. Wanneer hij vooruitgaat (V), dan is er 80% kans dat hij effectief een stap verder op het pad zal zetten (P), 10% kans dat hij rechts in het water valt (W), en 10% kans dat hij links tegen de muur botst (M). Wanneer hij blijft staan (B), dan zal hij sowieso zijn evenwicht verliezen, met 50% kans dat hij in het water valt (W), en 50% kans dat hij tegen de muur terechtkomt (M). De reward voor een stap verder op het pad te komen (P) is 1, de reward voor in het water te vallen (W) is -1, en de reward voor tegen de muur te botsten (M) is -0.1. De kans dat het mannetje vanop de huidige positie H onder dit beleid π in het water (W) zal terechtkomen is:

$$\mathcal{P}^{\pi}_{H,W} = \sum_{a=V,B} \pi(a|H) \mathcal{P}^{a}_{H,W} = 0.95 \text{ x } 0.1 + 0.05 \text{ x } 0.5 = 0.12$$

De te verwachten reward op positie H onder deze policy π is:

$$\mathcal{R}^{\pi}_{H} = \sum_{a=V,B} \pi(a|H) \mathcal{R}^{a}_{H} = \left(\pi(V|H) \sum_{s'=P,W,M} \mathbb{P}[R_{s'}|\, H, V]. R_{s'} \right) + \left(\pi(B|H) \sum_{s'=W,M} \mathbb{P}[R_{s'}|\, H, B]. R_{s'} \right)$$

$$= 0.95 \text{ x } [0.8 \text{ x } 1 + 0.1 \text{ x } (-1) + 0.1 \text{ x } (-0.1)] + 0.05 \text{ x } [0.5 \text{ x } (-1) + 0.5 \text{ x } (-0.1)] = 0.678$$

We kunnen die berekeningen uitvoeren met een eenvoudig R-scriptje:

```
P.a = matrix(c(0.8, 0.1, 0.1, 0, 0.5, 0.5), nrow=2, byrow=T,
             dimnames=list(rownames=c("vooruit", "blijven"),
                           colnames=c("pad", "water", "muur")))
print(P.a)

R = c(1, -1, -0.1)
names(R) = c("pad", "water", "muur")
print(R)

policy = c(0.95, 0.05)
names(policy) = c("vooruit", "blijven")
print(policy)

P = colSums(policy * P.a)
print(P)

R.huidig = sum(policy * rowSums(R * P.a))
print(R.huidig)
```

Als we het script runnen zien we het resultaat in de R console:

```
> print(P.a)
          colnames
rownames   pad water muur
  vooruit 0.8   0.1   0.1
  blijven 0.0   0.5   0.5
> print(R)
  pad water  muur
  1.0  -1.0  -0.1
> print(policy)
vooruit blijven
   0.95    0.05
> print(P)
  pad water  muur
 0.76  0.12  0.12
> print(R.huidig)
[1] 0.678
```

Matrix `P.a` bevat voor de twee mogelijke acties, vooruit gaan en blijven staan, de kansen op één van de drie volgende toestanden, nl. een stap verder op het pad terechtkomen, of in het water vallen, of tegen de muur botsen. Vector `R` toont de beloningen voor die drie toestanden. Vector `policy` bevat de kansen op de twee mogelijke acties. Dit zijn de gegevens waarmee dan `P` en `R.huidig` worden berekend. Vector `P` toont de kansen $\mathcal{P}^{\pi}_{H,P}$, $\mathcal{P}^{\pi}_{H,W}$ en $\mathcal{P}^{\pi}_{H,M}$. Dat zijn de kansen dat het mannetje respectievelijk een stap verder op het pad of in het water of tegen de muur terechtkomt. Variabele `R.huidig` is de verwachte beloning $\mathcal{R}^{\pi}_{H}$ in de huidige toestand.

Value function

Daar waar we in een Markov Reward Process de value function hebben gedefinieerd aan de hand van random samples, moeten we die in een Markov Decision Process definiëren aan de hand van de gegeven policy π. Bij een MRP was er slechts één verwachte cumulatieve reward, bij een MDP zijn er meerdere policies mogelijk, en elke policy heeft een andere verwachte cumulatieve reward. Omdat er in een MDP ook acties zijn, kunnen we nu zelfs twee value functions definiëren, één voor de toestanden en één voor de acties: de state-value en de action-value function.

De toestandswaarde-functie $v_\pi(s)$ van een MDP, in het Engels **state-value function**, is de verwachte return *G* startend in toestand *s*, waarbij policy π wordt gevolgd:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

Deze functie geeft aan hoe goed het is voor de agent om in toestand *s* te zijn onder policy π.

De actiewaarde-functie $q_\pi(s, a)$, in het Engels **action-value function**, is de verwachte return *G* wanneer actie *a* wordt geselecteerd in toestand *s* en daarbij policy π volgen:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

Deze functie geeft aan hoe goed het is voor de agent om actie *a* te kiezen wanneer hij in toestand *s* is en policy π volgt.

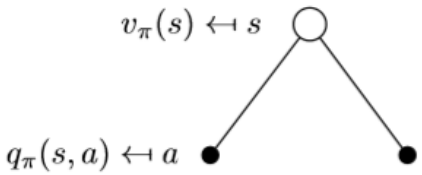
Bellman expectation equation

Net als bij een MRP kunnen we de value functions van een MDP eveneens recursief ontbinden:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \end{aligned}$$

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \end{aligned}$$

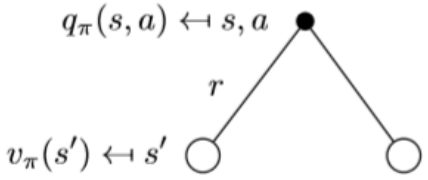
In woorden zeggen deze recursieve definities dat de value function gelijk is aan de verwachte onmiddellijke beloning plus de value function van de toestand waarin de agent terechtkomt. Beide functies zijn gerelateerd aan elkaar. Stel dat de agent in toestand *s* is, van waaruit hij verschillende acties *a* kan kiezen. In het onderstaande schema zijn dat er twee (zwarte bolletjes):



In dit geval kunnen we $v_\pi(s)$ schrijven als:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s). \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a)$$

Als de agent in toestand *s* actie *a* kiest, dan kan hij in verschillende volgende toestanden *s'* komen, waarna hij een reward *r* krijgt. In onderstaand schema worden er twee weergegeven (witte cirkels):



In elke toestand *s'* hij precies zal komen hangt af van de state transition probability $\mathcal{P}^a_{ss'}$. Gebruik makend van de lineariteit van verwachtingswaarde $\mathbb{E}$, gaan we eerst $q_\pi(s, a)$ schrijven als:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} | S_t = s, A_t = a] + \gamma \mathbb{E}_\pi[q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

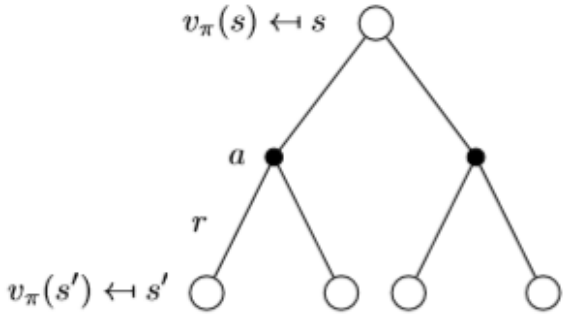
De eerste term is de definitie van de reward functie $\mathcal{R}^a_s$. De tweede term kunnen we in functie van $\mathcal{P}^a_{ss'}$ schrijven:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v(s')$$

In de volgende stap gaan we de gevonden uitdrukking voor $q_\pi(s,a)$ substitueren in de uitdrukking voor $v_\pi(s)$:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v(s') \right)$$

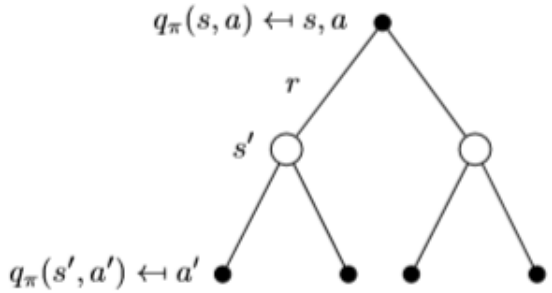
Schematisch stellen we dit voor door de twee vorige schema's samen te brengen:



De eerste uitdrukking voor $v_\pi(s)$ kunnen we eveneens in de uitdrukking voor $q_\pi(s,a)$ substitueren:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v(s') = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

Schematisch:



Samenvattend zijn de Bellman verwachtingsvergelijkingen, in het Engels Bellman expectation equations:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v(s') \right)$$
$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

We zagen eerder dat een MDP tot een MRP kan gereduceerd worden eens de policy π vastligt. Laten we de eerste Bellman expectation equation herschrijven en de definities van $\mathcal{P}_{ss'}^\pi$ en $\mathcal{R}_s^\pi$ toepassen:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a v(s') = \mathcal{R}_s^\pi + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^\pi v(s')$$

Op die manier hebben we deze vergelijking gereduceerd tot de Bellman vergelijking van een MRP. Als we alle toestanden beschouwen, kunnen we het stelsel vergelijkingen in matrixvorm schrijven:

$$\boldsymbol{v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi}$$

Zoals we reeds zagen, kunnen we dit matrixstelsel algebraïsch oplossen:

$$\boldsymbol{v_\pi = [I - \gamma \mathcal{P}^\pi]^{-1} \mathcal{R}^\pi}$$

De Bellman vergelijking voor de action-value function kunnen we ook in matrixvorm schrijven, maar omdat een MRP geen acties bevat, kunnen we de MDP niet reduceren tot een MRP. Toch kunnen we tot een gelijkaardige matrixvergelijking komen. Eerst en vooral herschrijven we de vergelijking:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \sum_{a' \in \mathcal{A}} \mathcal{P}_{ss'}^a \, \pi(a'|s') \, q_\pi(s',a')$$

Wanneer we de kans $\mathcal{P}_{ss'}^{aa'}$ definiëren als de kans dat de agent in toestand s actie a kiest waarna hij in toestand s' komt en daar actie a' kiest, dan geldt:

$$\mathcal{P}_{ss'}^{aa'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, \, A_t = a] \, \mathbb{P}[A_{t+1} = a' \mid S_{t+1} = s'] = \mathcal{P}_{ss'}^a \, \pi(a'|s')$$

Gebruik makend van $\mathcal{P}_{ss'}^{aa'}$ kunnen we inderdaad de Bellman vergelijking voor de action-value function in dezelfde vorm schrijven als die voor de state-action function:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{\substack{(s',a') \\ \in \mathcal{S} \times \mathcal{A}}} \mathcal{P}_{ss'}^{aa'} \, q_\pi(s',a')$$

De $q_\pi(s,a)$ gaan we nu ook in een vector plaatsen door alle mogelijke combinaties van toestanden *s* en acties *a* te beschouwen:

$$\boldsymbol{q_\pi} = \begin{bmatrix} q_\pi(s_1,a_1) \\ \vdots \\ q_\pi(s_m,a_n) \end{bmatrix}$$

met *m* het aantal toestanden en *n* het aantal acties. Vector $\boldsymbol{q_\pi}$ heeft dus *m* x *n* elementen. Hetzelfde kunnen we doen voor de reward function $\mathcal{R}_s^a = \mathcal{R}(s,a)$:

$$\boldsymbol{\mathcal{R}_q^\pi} = \begin{bmatrix} \mathcal{R}(s_1,a_1) \\ \vdots \\ \mathcal{R}(s_m,a_n) \end{bmatrix}$$

Vector $\boldsymbol{\mathcal{R}_q^\pi}$ heeft ook *m* x *n* elementen. Tenslotte moeten we nog de matrix opstellen met kansen $\mathcal{P}_{ss'}^{aa'} = \mathcal{P}(s,a,s',a')$:

$$\boldsymbol{\mathcal{P}_q^\pi} = \begin{bmatrix} \mathcal{P}(s_1,a_1,s_1,a_1) & \cdots & \mathcal{P}(s_1,a_1,s_m,a_n) \\ \vdots & \ddots & \vdots \\ \mathcal{P}(s_m,a_n,s_1,a_1) & \cdots & \mathcal{P}(s_m,a_n,s_m,a_n) \end{bmatrix}$$

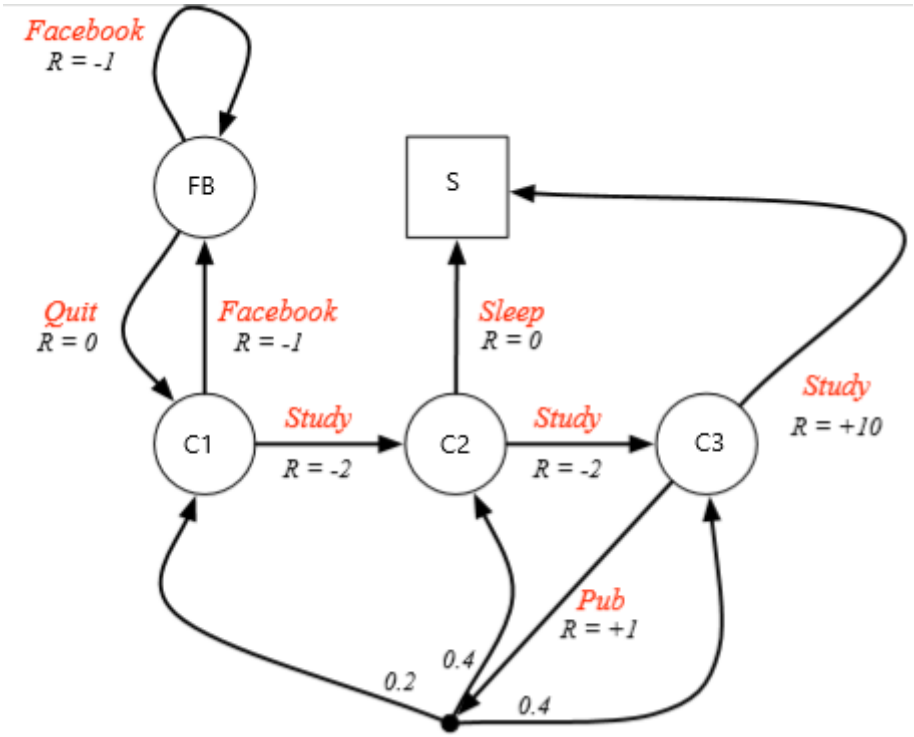
Matrix $\boldsymbol{\mathcal{P}_q^\pi}$ is een vierkante matrix met (*m* x *n*) x (*m* x *n*) elementen. Zo kunnen we een gelijkaardig matrixstelsel opstellen als voor de state-value function, dat we ook algebraïsch kunnen oplossen:

$$\boldsymbol{q_\pi = \mathcal{R}_q^\pi + \gamma \mathcal{P}_q^\pi q_\pi}$$

$$\boldsymbol{q_\pi = \left[I - \gamma \mathcal{P}_q^\pi\right]^{-1} \mathcal{R}_q^\pi}$$

Voorbeeld: student

In dit voorbeeld hebben we van ons eerder besproken student MRP een MDP gemaakt door acties (rood) toe te voegen:



In dit geval hebben 4 tussentijdse toestanden (cirkels) en 1 eindtoestand (vierkant). De toestanden noemen we zoals in de MRP van dit voorbeeld C1, C2, C3, FB en S. In elke tussentijdse toestand kan de student twee acties kiezen die hem/haar met 100% zekerheid naar een andere toestand zal brengen. Enkel bij de actie “Pub” is er een element van willekeurigheid toegevoegd en zijn er 3 mogelijke volgende toestanden die elk met een gekende waarschijnlijkheid worden bereikt. Deze kansen worden op de figuur weergegeven. De beloning *R* die de student krijgt na het uitvoeren van een actie, staat eveneens op de figuur onder de actie aangeduid.

Stel nu dat de student de policy $\pi(a|s)$ = 0.5 volgt, wat betekent dat in elke tussentijdse toestand *s* de kans 50% is dat hij/zij één van de twee mogelijke acties *a* kiest. We kunnen nu de state-value function $v_\pi(s)$ berekenen door het opstellen en oplossen van het stelsel van Bellman vergelijkingen. We gaan dit in R doen en daarbij gebruik maken van matrices, zoals we dat eerder bij het voorbeeld van de MRP deden.

We starten met het definiëren van de toestanden (`states`) en de acties (`actions`) en houden ook het aantal toestanden (`nstates`) en aantal acties (`nactions`) bij. De acties worden afgekort met hun eerste letter, tenzij actie “Sleep” die we met “Z” aanduiden.

```
# states
states = c("C1", "C2", "C3", "FB", "S")
nstates = length(states)

# actions
actions = c("S", "Z", "F", "Q", "P")
nactions = length(actions)
```

De policy $\pi(a|s)$ = 0.5 definiëren we aan de hand van een matrix die we alloceren met nullen:

```
# policy
policy = matrix(0, nrow=nstates, ncol=nactions,
                dimnames=list(rownames=states,
                              colnames=actions))
policy["C1", "S"] = 0.5
policy["C1", "F"] = 0.5
policy["C2", "S"] = 0.5
policy["C2", "Z"] = 0.5
policy["C3", "S"] = 0.5
policy["C3", "P"] = 0.5
policy["FB", "F"] = 0.5
policy["FB", "Q"] = 0.5
print(policy)
```

Wanneer actie "a" mogelijk is vanuit toestand "s", dan krijgt het element `policy["s","a"]` de waarde 0.5. Anders blijft het element op nul staan. Op dezelfde manier gaan we de state transition matrix opbouwen. Deze bevat de elementen $\mathcal{P}_{ss'}^{a}$. Maar omdat elke kans van 3 variabelen afhangt, namelijk de huidige toestand *s*, de actie *a*, en de volgende toestand *s'*, moeten we een 3D matrix opstellen, die we in R als `array` definiëren. Samengevat komt het er dus op neer dat de state transition probability $\mathcal{P}_{ss'}^{a}$ hier dus wordt opgeslagen als `P.a["s", "a", "s'"]`:

```
# state transition matrix
P.a = array(0, dim=c(nstates, nactions, nstates),
            dimnames=list(from=states,
                          action=actions,
                          to=states))
P.a["C1", "S", "C2"] = 1
P.a["C1", "F", "FB"] = 1
P.a["C2", "S", "C3"] = 1
P.a["C2", "Z", "S"] = 1
P.a["C3", "S", "S"] = 1
P.a["C3", "P", "C1"] = 0.2
P.a["C3", "P", "C2"] = 0.4
P.a["C3", "P", "C3"] = 0.4
P.a["FB", "F", "FB"] = 1
P.a["FB", "Q", "C1"] = 1
print(P.a)
```

Deze state transition matrix gaan we nu reduceren tot matrix $\boldsymbol{\mathcal{P}^{\pi}}$ die de elementen $\mathcal{P}_{ss'}^{\pi}$ bevat:

```
# reduced state transition matrix
P = matrix(nrow=nstates, ncol=nstates,
           dimnames=list(rownames=states,
                         colnames=states))
for (state in states) {
  P[state, ] = policy[state, ] %*% P.a[state, , ]
}
print(P)
```

De lus doorloopt de verschillende begintoestanden *s* en gaat voor elk van deze toestanden $\mathcal{P}_{ss'}^{\pi}$ berekenen met behulp van matrixvermenigvuldiging, wat neerkomt op de formule die we hierboven hebben gegeven:

$$\mathcal{P}_{ss'}^{\pi} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^{a}$$

De reward matrix `R.a` met elementen $\mathcal{R}_{s}^{a}$ gaan we op dezelfde manier reduceren tot kolomvector $\boldsymbol{\mathcal{R}^{\pi}}$:

```
# reward matrix
R.a = matrix(0, nrow=nstates, ncol=nactions,
             dimnames=list(rownames=states,
                           colnames=actions))
R.a["C1", "S"] = -2
R.a["C1", "F"] = -1
R.a["C2", "S"] = -2
R.a["C2", "Z"] = 0
R.a["C3", "S"] = 10
R.a["C3", "P"] = 1
R.a["FB", "F"] = -1
R.a["FB", "Q"] = 0
print(R.a)

# reduced reward vector
R = rowSums(policy * R.a)
print(R)
```

Vector `R` bevat dus de elementen $\mathcal{R}_s^\pi$, die zoals we zagen als volgt worden berekend:

$$\mathcal{R}_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s)\mathcal{R}_s^a$$

Tenslotte hebben we nog de eenheidsmatrix $\boldsymbol{I}$ nodig:

```
# identity matrix
I = diag(1, nstates, nstates)
```

Met behulp van de R functie `solve` kunnen we nu het stelsel oplossen voor verschillende discount factors γ, variërend van 0 tot en met 1:

```
# discount factors
gamma = seq(0, 1, length.out=11)

# state value function
v = matrix(nrow=nstates, ncol=length(gamma),
           dimnames=list(rownames=states,
                         colnames=gamma))
for (i in 1:length(gamma)) {
  v[, i] = solve(I - gamma[i]*P, R)
}

# solutions
print(v)
```

In de lus wordt het stelsel met Bellman vergelijkingen opgelost:

$$\boldsymbol{v_\pi} = [\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}^\pi}]^{-1}\boldsymbol{\mathcal{R}^\pi}$$

De oplossing van het stelsel is een kolomvector $\boldsymbol{v_\pi}$ met de state value function $v_\pi(s)$ voor de 5 toestanden. Deze kolomvectoren bewaren we in matrix `v`.

In de console zien we uiteindelijk de oplossingen:

```
> print(v)
         colnames
rownames    0        0.1        0.2        0.3       0.4        0.5        0.6
      C1 -1.5 -1.5664842 -1.6167446 -1.6516823 -1.671722 -1.6766623 -1.6653622
      C2 -1.0 -0.7209226 -0.4322526 -0.1315065  0.184321  0.5189048  0.8768051
      C3  5.5  5.5815477  5.6774740  5.7899565  5.921605  6.0756193  6.2560169
      FB -0.5 -0.6087623 -0.7351938 -0.8797086 -1.042930 -1.2255541 -1.4280124
      S   0.0  0.0000000  0.0000000  0.0000000  0.000000  0.0000000  0.0000000
         colnames
rownames       0.7       0.8       0.9         1
      C1 -1.635044 -1.579578 -1.484477 -1.307692
      C2  1.263800  1.687440  2.158158  2.692308
      C3  6.467999  6.718600  7.018129  7.384615
      FB -1.649639 -1.886385 -2.123663 -2.307692
      S   0.000000  0.000000  0.000000  0.000000
```

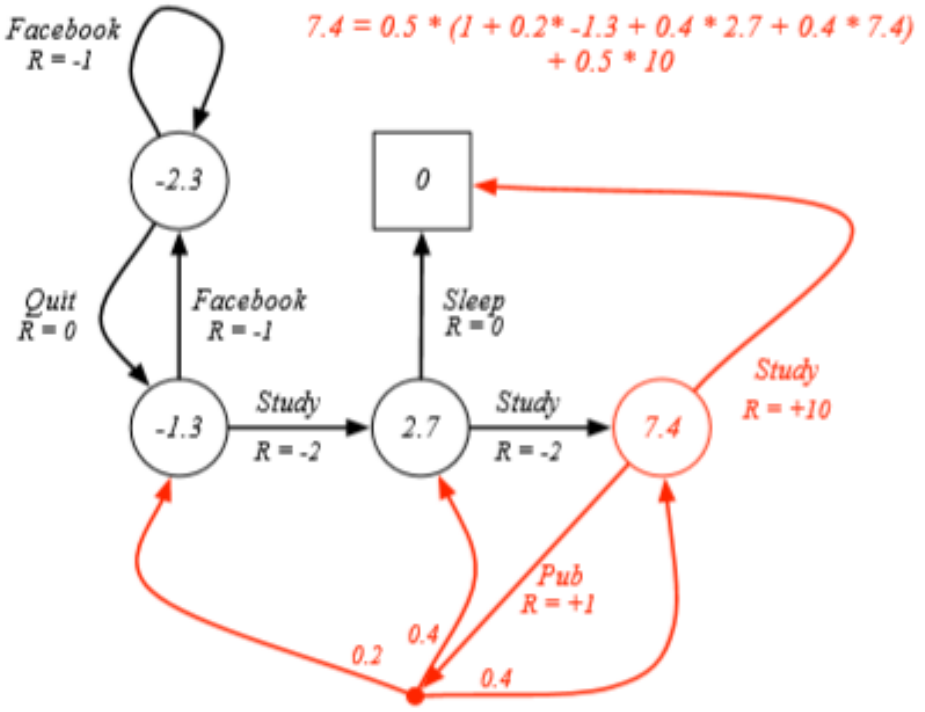
We bekijken één oplossing in detail, nl. de waarde 7.4 van toestand C3 voor γ = 1. De Bellman vergelijking is:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$

In dit geval moeten we dus de volgende berekening maken:

$$v_\pi(C3) = \pi(Pub|C3) \left[\mathcal{R}_{C3}^{Pub} + \gamma \left(\mathcal{P}_{C3,C1}^{Pub} v(C1) + \mathcal{P}_{C3,C2}^{Pub} v(C2) + \mathcal{P}_{C3,C3}^{Pub} v(C3) \right) \right] + \pi(Study|C3) \left[\mathcal{R}_{C3}^{Study} + \gamma \mathcal{P}_{C3,S}^{Study} v(S) \right]$$

Deze berekening wordt hieronder in het rood uitgevoerd:



Wanneer we de Bellman vergelijking hierboven volledig uitwerken totdat er geen haakjes meer in voorkomen, dan kunnen we de vergelijking schrijven in functie van $\mathcal{P}_{ss'}^\pi$ en $\mathcal{R}_s^\pi$:

$$v_\pi(C3) = \mathcal{R}_{C3}^\pi + \gamma \mathcal{P}_{C3,C1}^\pi v(C1) + \gamma \mathcal{P}_{C3,C2}^\pi v(C2) + \gamma \mathcal{P}_{C3,C3}^\pi v(C3) + \gamma \mathcal{P}_{C3,S}^\pi v(S)$$

met

$$\mathcal{R}^{\pi}_{C3} = \pi(Pub|C3)\mathcal{R}^{Pub}_{C3} + \pi(Study|C3)\mathcal{R}^{Study}_{C3}$$

$$\mathcal{P}^{\pi}_{C3,C1} = \pi(Pub|C3)\mathcal{P}^{Pub}_{C3,C1}$$

$$\mathcal{P}^{\pi}_{C3,C2} = \pi(Pub|C3)\mathcal{P}^{Pub}_{C3,C2}$$

$$\mathcal{P}^{\pi}_{C3,C3} = \pi(Pub|C3)\mathcal{P}^{Pub}_{C3,C3}$$

$$\mathcal{P}^{\pi}_{C3,S} = \pi(Study|C3)\mathcal{P}^{Study}_{C3,S}$$

Dit is de vergelijking die in het script wordt toegepast.

Op dezelfde manier gaan we nu de action-value function bepalen, waarbij we verder werken met de inputvariabelen die we voor het berekenen van de state-value function hebben gedefinieerd. We overlopen ze nog eens:

```
# states
states = c("C1", "C2", "C3", "FB", "S")
nstates = length(states)

# actions
actions = c("S", "Z", "F", "Q", "P")
nactions = length(actions)



# policy
policy = matrix(0, nrow=nstates, ncol=nactions,
                dimnames=list(rownames=states,
                              colnames=actions))
policy["C1", "S"] = 0.5
policy["C1", "F"] = 0.5
policy["C2", "S"] = 0.5
policy["C2", "Z"] = 0.5
policy["C3", "S"] = 0.5
policy["C3", "P"] = 0.5
policy["FB", "F"] = 0.5
policy["FB", "Q"] = 0.5
print(policy)



# reward matrix
R.a = matrix(0, nrow=nstates, ncol=nactions,
             dimnames=list(rownames=states,
                           colnames=actions))
R.a["C1", "S"] = -2
R.a["C1", "F"] = -1
R.a["C2", "S"] = -2
R.a["C2", "Z"] = 0
R.a["C3", "S"] = 10
R.a["C3", "P"] = 1
R.a["FB", "F"] = -1
R.a["FB", "Q"] = 0
print(R.a)
```

```
# state transition matrix
P.a = array(0, dim=c(nstates, nactions, nstates),
            dimnames=list(from=states,
                          action=actions,
                          to=states))
P.a["C1", "S", "C2"] = 1
P.a["C1", "F", "FB"] = 1
P.a["C2", "S", "C3"] = 1
P.a["C2", "Z", "S"] = 1
P.a["C3", "S", "S"] = 1
P.a["C3", "P", "C1"] = 0.2
P.a["C3", "P", "C2"] = 0.4
P.a["C3", "P", "C3"] = 0.4
P.a["FB", "F", "FB"] = 1
P.a["FB", "Q", "C1"] = 1
print(P.a)
```

```
# discount factors
gamma = seq(0, 1, length.out=11)
```

Bij de theoretische uitwerking van het matrixstelsel die we hierboven hebben uiteengezet, beschouwden we alle mogelijke combinaties van toestanden en acties. In praktijk kunnen we ons echter beperken tot diegene die effectief voorkomen in de MDP. Die gaan we opslaan in een vector `sanames` (waarbij "sa" de afkorting is voor state-action). Daarnaast gaan we ook een list `sa` definiëren met in de velden de overeenkomstige state-action koppels waarvan het eerste element de toestand is en het tweede element de actie. Dit zal ons toelaten om eenvoudig de elementen uit de inputmatrices op te vragen.

```
# relevant state-actions
sanames = c("C1_S", "C1_F", "C2_S", "C2_Z", "C3_S",
            "C3_P", "FB_F", "FB_Q")
nsa = length(sanames)
sa = strsplit(sanames, "_")
names(sa) = sanames
```

De identity matrix $\boldsymbol{I}$ moeten we opnieuw definiëren, omdat die in dit geval niet hetzelfde aantal rijen en kolommen heeft:

```
# identity matrix
I.q = diag(1, nsa, nsa)
```

We kunnen nu eenvoudig de reward vector $\boldsymbol{\mathcal{R}_q^\pi}$ construeren die de rewards $\mathcal{R}_S^a$ bevat door de input matrix `R.a` te herschikken:

```
# reward vector
R.q = rep(0, nsa)
names(R.q) = sanames
for (rname in sanames) {
  state = sa[[rname]][1]
  action = sa[[rname]][2]
  R.q[rname] = R.a[state, action]
}
print(R.q)
```

De matrix $\boldsymbol{\mathcal{P}_q^\pi}$ met kansen $\mathcal{P}_{ss'}^{aa'} = \mathcal{P}_{ss'}^{a}\, \pi(a'|s')$ leiden we af van inputmatrices `policy` en `P.a`:

```
# probability matrix
P.q = matrix(0, nrow=nsa, ncol=nsa,
             dimnames=list(rownames=sanames,
                           colnames=sanames))
for (rname in sanames) {
  for (cname in sanames) {
    fromstate = sa[[rname]][1]
    fromaction = sa[[rname]][2]
    tostate = sa[[cname]][1]
    toaction = sa[[cname]][2]
    P.q[rname, cname] = P.a[fromstate, fromaction, tostate] *
                        policy[tostate, toaction]
  }
}
print(P.q)
```

Nu we alle nodige matrices hebben gedefinieerd, kunnen we de action-value function voor de verschillende discount factors gaan berekenen:

```
# action value function
q = matrix(nrow=nsa, ncol=length(gamma),
           dimnames=list(rownames=sanames,
                         colnames=gamma))
for (i in 1:length(gamma)) {
  q[, i] = solve(I.q - gamma[i]*P.q, R.q)
}

# solutions
print(q)
```

In de lus wordt het stelsel van Bellman vergelijkingen opgelost:

$$\boldsymbol{q_\pi} = \left[\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}_q^\pi}\right]^{-1} \boldsymbol{\mathcal{R}_q^\pi}$$

De oplossing van het stelsel is een kolomvector $\boldsymbol{q_\pi}$ met de action value function $q_\pi(s, a)$ voor de 8 state-action koppels. Deze kolomvectoren bewaren we in matrix `q`. In de console zien we uiteindelijk de oplossingen:

```
> print(q)
         colnames
rownames  0        0.1        0.2        0.3        0.4        0.5        0.6
    C1_S -2 -2.0720923 -2.0864505 -2.0394520 -1.9262716 -1.7405476 -1.4739170
    C1_F -1 -1.0608762 -1.1470388 -1.2639126 -1.4171722 -1.6127771 -1.8568074
    C2_S -2 -1.4418452 -0.8645052 -0.2630130  0.3686421  1.0378096  1.7536101
    C2_Z  0  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000  0.0000000
    C3_S 10 10.0000000 10.0000000 10.0000000 10.0000000 10.0000000 10.0000000
    C3_P  1  1.1630953  1.3549479  1.5799131  1.8432105  2.1512386  2.5120338
    FB_F -1 -1.0608762 -1.1470388 -1.2639126 -1.4171722 -1.6127771 -1.8568074
    FB_Q  0 -0.1566484 -0.3233489 -0.4955047 -0.6686888 -0.8383312 -0.9992173
         colnames
rownames       0.7        0.8         0.9          1
    C1_S -1.115340 -0.6500479 -0.05765792  0.6923077
    C1_F -2.154747 -2.5091083 -2.91129706 -3.3076923
    C2_S  2.527599  3.3748802  4.31631573  5.3846154
    C2_Z  0.000000  0.0000000  0.00000000  0.0000000
    C3_S 10.000000 10.0000000 10.00000000 10.0000000
    C3_P  2.935997  3.4372004  4.03625717  4.7692308
    FB_F -2.154747 -2.5091083 -2.91129706 -3.3076923
    FB_Q -1.144531 -1.2636625 -1.33602974 -1.3076923
```

Laten we opnieuw één van die oplossingen in detail bekijken. Nemen we $q_\pi(C3, P)$ bij γ = 1. We komen hiervoor een waarde van 4.77 uit. Dit is dus de waarde van de actie "Pub" uitgevoerd vanuit toestand "C3". De Bellman vergelijking is:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')$$

In dit geval moeten we dus de volgende berekening maken:

$$\begin{aligned} q_\pi(C3, P) = \mathcal{R}_{C3}^P \\ &+ \gamma \big[\mathcal{P}_{C3,C1}^P (\pi(S|C1) q_\pi(C1, S) + \pi(F|C1) q_\pi(C1, F)) \\ &+ \mathcal{P}_{C3,C2}^P (\pi(S|C2) q_\pi(C2, S) + \pi(Z|C2) q_\pi(C2, Z)) \\ &+ \mathcal{P}_{C3,C3}^P (\pi(S|C3) q_\pi(C3, S) + \pi(P|C3) q_\pi(C3, P)) \big] \end{aligned}$$

Wanneer we de waarden invullen, krijgen we inderdaad:

$$\begin{aligned} q_\pi(C3, P) = 1 + 1 \text{ x } [0.2 \text{ x } (0.5 \text{ x } 0.69 + 0.5 \text{ x } (-3.31)) + 0.4 \text{ x } (0.5 \text{ x } 5.38 + 0.5 \text{ x } 0) \\ + 0.2 \text{ x } (0.5 \text{ x } 10 + 0.5 \text{ x } 4.77)] = 4.77 \end{aligned}$$

Dit is een eenvoudig voorbeeld met een klein aantal vergelijkingen waardoor het rechtstreeks algebraïsch oplossen van het stelsel zonder problemen lukt. Wanneer er echter een zeer groot aantal vergelijkingen zijn, dan zijn iteratieve oplossingsalgoritmen efficiënter.

Optimal value functions

Met de Bellman expectation equations kunnen we de value functions bepalen voor een gegeven policy. Maar in RL is het de bedoeling dat de agent zich optimaal gaat gedragen. Om dit probleem op te lossen definiëren we eerst de optimale waarde-functies. Ook hier definiëren we een optimale state-value function en een optimale action-value function.

De **optimal state-value function** is de maximale state-value function over alle policies:

$$v_*(s) = \max_\pi v_\pi(s)$$

De **optimal action-value function** is de maximale action-falue function over alle policies:

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Deze definities zeggen nog niets over de optimale policy π_* maar wel over de best mogelijke acties in de MDP. In praktijk is het zelfs zo dat het MDP probleem is opgelost wanneer $q_*(s,a)$ is gekend.

Optimal policy

Om over een optimale policy te kunnen spreken, moeten we formeel een partiële ordening op de verzameling van alle policies in de MDP definiëren:

$$\pi \geq \pi' \text{ als } \forall s: v_\pi(s) \geq v_{\pi'}(s)$$

Een partiële ordening kennen we als de logische operator "kleiner of gelijk aan". Meer uitleg over wat een partiële ordening juist is, vinden we op wikipedia: https://nl.wikipedia.org/wiki/Parti%C3%ABle_orde.

Toegepast op policies is policy π groter of gelijk aan policy π' als de state-value function van toestand s onder policy π groter of gelijk is aan die onder policy π', en dit voor alle toestanden in de MDP.

Het volgende **theorema** is fundamenteel in RL:

Voor elke MDP bestaat er minstens één optimale policy π_* die beter is dan of even goed is als alle andere policies:

$$\pi_* \geq \pi, \qquad \forall \pi$$

Alle optimale policies bereiken de optimale state-value function en optimale action-value function:

$$v_{\pi_*}(s) = v_*(s)$$
$$q_{\pi_*}(s,a) = q_*(s,a)$$

De optimale policy vinden we door de optimale action-value function $q_*(s,a)$ te maximaliseren:

$$\pi_*(a|s) = \begin{cases} 1 \text{ als } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s,a) \\ 0 \qquad\qquad\qquad \text{anders} \end{cases}$$

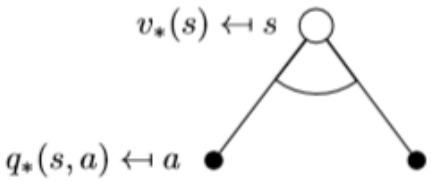
Elke MDP heeft een deterministische optimale policy π_* zoals hierboven gedefinieerd. Hieruit volgt ook dat we onmiddellijk deze optimale policy π_* kunnen bepalen wanneer we de optimale action-value function $q_*(s,a)$ hebben gevonden.

Bellman optimality equation

Ook voor de optimale value functions kunnen we recursieve vergelijkingen opstellen doordat beide functies gerelateerd zijn aan elkaar. Laten we starten bij toestand *s* met optimale waarde $v_*(s)$. Vanuit die toestand kiezen we de actie *a* met de grootste waarde:

$$v_*(s) = \max_a q_*(s,a)$$

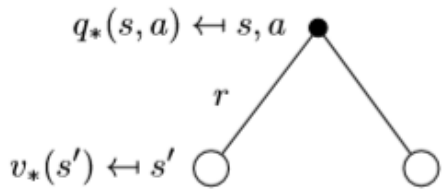
Schematisch geven we dat als volgt weer, waarbij de boog onder toestand *s* het maximum voorstelt:



Na het selecteren van die actie *a* krijgen we een reward $\mathcal{R}_s^a$. De volgende toestand *s'* wordt bepaald door het systeem en hangt af van de kansen $\mathcal{P}_{ss'}^a$. Hiermee gaan we het gewogen gemiddelde berekenen van de optimale waarden van de mogelijke volgende toestanden. De waarde van de actie *a* is dus:

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a\, v_*(s')$$

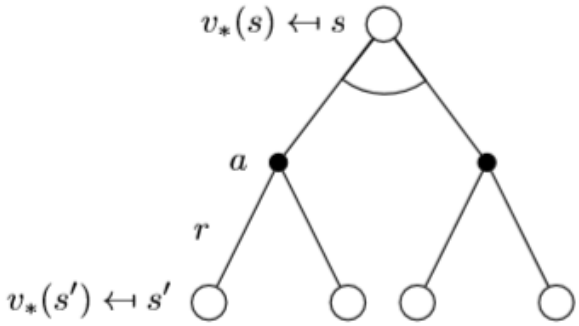
Schematisch, met *r* de reward na het nemen van actie *a*:



De bekomen uitdrukking voor $q_*(s,a)$ substitueren we nu in de uitdrukking voor $v_*(s)$:

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$

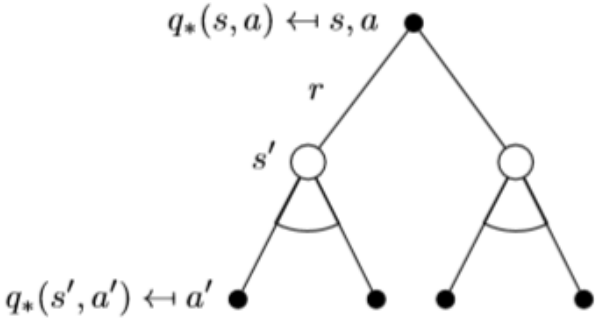
Schematisch stellen we dit voor door de twee vorige schema's samen te voegen:



Omgekeerd kunnen we ook de uitdrukking voor $v_*(s)$ substitueren in de uitdrukking voor $q_*(s,a)$:

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

Schematisch:



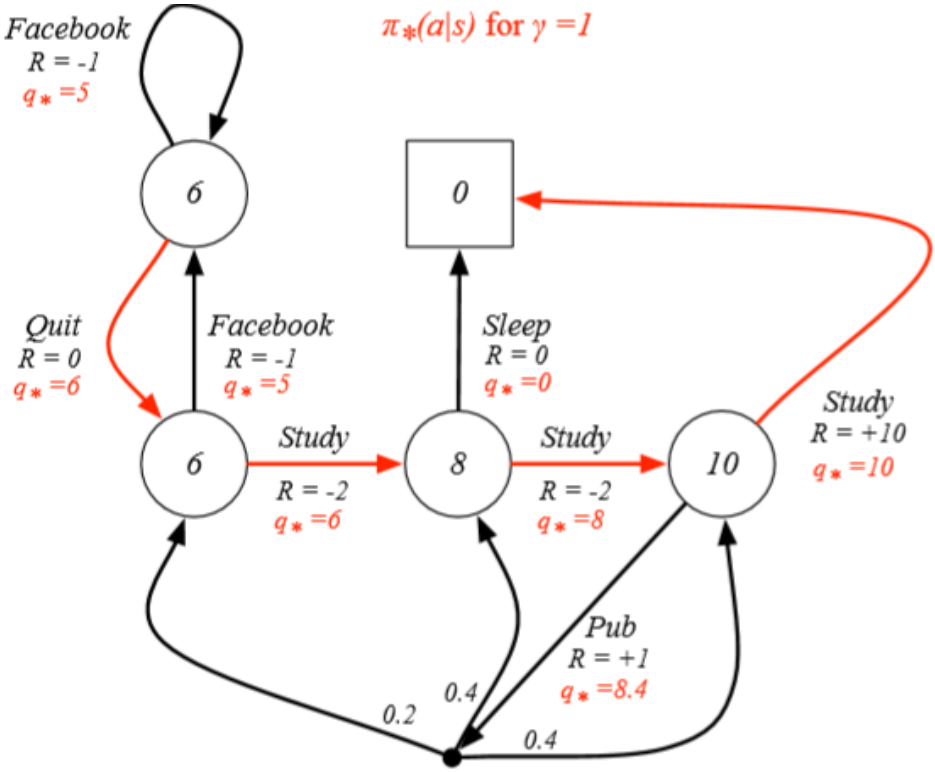
Wanneer men het in RL over de Bellman vergelijkingen heeft, dan bedoelt men meestal de recursieve vergelijkingen die we hier hebben afgeleid:

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$
$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

De idee achter die vergelijkingen is eigenlijk vrij simpel: de optimale waarde van een bepaalde toestand vind je door de actie te kiezen die tot de grootste cumulatieve reward leidt, en de optimale waarde van een actie is gelijk aan de reward van die actie plus de maximale reward die men kan verwachten na het nemen van die actie. In tegenstelling tot de Bellman expectation vergelijkingen zijn deze vergelijkingen niet lineair. Over het algemeen is er ook geen zogenaamde "closed-form" oplossing (zie https://en.wikipedia.org/wiki/Closed-form_expression voor meer uitleg). Om die redenen is men aangewezen op iteratieve oplossingsmethoden zoals Value Iteration, Policy Iteration, Q-learning of Sarsa, die we in de volgende hoofdstukken zullen bespreken.

Voorbeeld: student

Hieronder tonen we de student MDP uit onze eerdere voorbeelden met daarop de optimale policy weergegeven (rood) en de bijhorende optimale state-value en action-value functions. Om het eenvoudig te houden is de discount factor γ gelijk aan 1. Op die manier kunnen we de optimale waarden van de toestanden op zicht berekenen. Op de figuur zijn ze weergegeven in de cirkels die de toestanden voorstellen. De maximale cumulatieve reward die we vanuit toestand C3 kunnen behalen is inderdaad 10 door voor de actie "Study" te kiezen. De maximale state-value voor deze toestand is dus 10. In C2 krijgen we de maximale cumulatieve reward door twee keer voor "Study" te kiezen. Zo krijgen we -2 + 10 = 8 als maximale state-value. In C1 is de maximale state-value gelijk aan 6 die we bekomen door drie keer voor "Study" te kiezen. In toestand FB is de maximale state-value eveneens 6 en die bekomen we door voor "Quit" te kiezen met een reward van 0, en daarbij de maximale state-value van C1 op te tellen.



Voor het bepalen van de maximale action-value function starten we bij actie "Study" die vanuit C3 vertrekt. Deze actie heeft een reward van 10 en komt aan in de eindtoestand met waarde 0, dus de maximale waarde van die actie is 10. Hetzelfde geldt voor de actie "Sleep" die vanuit C2 vertrekt en ook in de eindtoestand eindigt. De reward voor deze actie is 0, dus de action-value is eveneens 0. De actie "Study" die vanuit C2 vertrekt, heeft als reward -2, en komt aan in C3. Daarbij moeten we de maximale waarde van de mogelijke acties tellen die vanuit C3 vertrekken, en dat is 10, dus de action-value is 8. Op dezelfde manier kunnen we de waarden voor de overige actions bepalen. Enkel voor de actie "Pub" is de berekening iets ingewikkelder. We werken de Bellman vergelijking uit:

$$
\begin{aligned}
q_*(C3, Pub) &= \mathcal{R}_{C3}^{Pub} + \gamma \sum_{s'=C1,C2,C3} \mathcal{P}_{C3,s'}^{Pub} \max_{a'} q_*(s', a') \\
&= \mathcal{R}_{C3}^{Pub} + \gamma \left(\mathcal{P}_{C3,C1}^{Pub} q_*(C1, Study) + \mathcal{P}_{C3,C2}^{Pub} q_*(C2, Study) + \mathcal{P}_{C3,C1}^{Pub} q_*(C3, Study) \right) \\
&= 1 + 1 \text{ x } (0.2 \text{ x } 6 + 0.4 \text{ x } 8 + 0.4 \text{ x } 10) = 8.4
\end{aligned}
$$

Merk op dat we in dit geval $q_*(C3, Pub)$ moeten kennen om $q_*(C3, Pub)$ te kunnen berekenen. Dus in principe kunnen we dit probleem niet op die manier gaan oplossen. In het volgende hoofdstuk zullen we zien hoe we dit wel kunnen oplossen.

Nu we de optimale action-value function q_* hebben gevonden, kunnen we eenvoudig de optimale policy π_* bepalen. We starten in de eindtoestand en kiezen van alle acties die in deze toestand toekomen, die met de grootste waarde. Dit is actie "Study" die vertrekt in C3. In C3 kiezen we opnieuw de actie met de grootste waarde. Dit is actie "Study" die vertrekt in C2. En zo gaan we verder tot we in toestand FB komen.

2.4. MDP uitbreidingen

We vermeldden eerder al dat het discrete, eindige Markov Decision Process dat we hier hebben gedefinieerd, kan uitgebreid worden naar een continue en/of oneindige MDP. Partially observable MDPs (POMDPs) hebben we ook al vernoemd, en daarbij vermeld dat die kunnen omgezet worden naar een volledig waarneembare MDP, zoals we hier hebben besproken. Dan zijn er ook nog MDPs waarbij men de reward niet gaat reduceren met een discount factor, maar gaat uitmiddelen. Een bespreking van deze uitbreidingen valt echter buiten de scope van dit document.

2.5. Samenvatting

In dit hoofdstuk hebben we het Markov Decision Process formeel beschreven door eerst het Markovproces te bekijken, een systeem dat bestaat uit een verzameling met toestanden $\boldsymbol{\mathcal{S}}$, en een state transition matrix $\boldsymbol{\mathcal{P}}$ die aangeeft wat de kans is om van de ene toestand naar de andere over te gaan. Wanneer een Markovproces eindig en discreet is, dan noemen we dit een Markovketen.

Het Markovproces hebben we dan uitgebreid naar een Markov Reward Process (MRP) door een reward functie $\boldsymbol{\mathcal{R}}$ te definiëren die aan elke toestand een beloning geeft. De return G_t hebben we gedefinieerd als de totale beloning vanaf tijdstap t, waarbij discount factor γ tussen 0 en 1 werd gebruikt om de toekomstige rewards te verminderen zodat de return eindig blijft. De value function $v(s)$ is de verwachte return vertrekkende van toestand s, en geeft de lange-termijn waarde van die toestand. Deze functie kan recursief gedefinieerd worden m.b.v. de Bellman vergelijking:

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}\, v(s')$$

Wanneer we deze vergelijking voor alle toestanden opstellen, dan kunnen we het bekomen stelsel in matrixvorm noteren en algebraïsch oplossen:

$$\boldsymbol{v} = [\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}}]^{-1} \boldsymbol{\mathcal{R}}$$

Tenslotte hebben we het Markov Reward Process uitgebreid naar een Markov Decision Process (MDP) door middel van een verzameling $\boldsymbol{\mathcal{A}}$ met acties die kunnen gekozen worden in een bepaalde toestand. De reward functie $\boldsymbol{\mathcal{R}}$ en de state transition matrix $\boldsymbol{\mathcal{P}}$ zijn daardoor ook afhankelijk van deze acties. De agent kiest een actie volgens een bepaalde policy π. Bij een deterministische policy ligt het 100% vast welke actie wordt gekozen. Bij een stochastische policy kunnen verschillende acties gekozen worden met een gekende probabiliteit. Een tweede element van willekeurigheid zit in het feit dat het systeem bepaalt in welke toestand de agent komt na het kiezen van die actie. Deze kansen worden gegeven door de state transition matrix $\boldsymbol{\mathcal{P}}$.

Bij een MDP kunnen we twee soorten value functions definiëren, die beiden afhangen van de gevolgde policy. De state-value function $v_\pi(s)$ is de verwachte return startend in toestand s, de action-value function $q_\pi(s, a)$ is de verwachte return wanneer actie a wordt geselecteerd in

toestand s. Deze functies kunnen opnieuw recursief gedefinieerd worden m.b.v. de Bellman expectation equations:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

Wanneer we deze vergelijkingen voor alle toestanden en acties opstellen, dan kunnen we de bekomen stelsels in matrixvorm noteren en algebraïsch oplossen:

$$\boldsymbol{v_\pi} = [\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}^\pi}]^{-1} \boldsymbol{\mathcal{R}^\pi}$$

$$\boldsymbol{q_\pi} = \left[\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}_q^\pi}\right]^{-1} \boldsymbol{\mathcal{R}_q^\pi}$$

In RL is het doel echter om de optimale policy te vinden die aangeeft welke acties de agent moeten kiezen om een maximale cumulatieve reward te verkrijgen. Een fundamenteel theorema zegt dat voor elke MDP er minstens één optimale policy π_* bestaat die beter is dan of even goed is als alle andere policies. Al deze optimale policies bereiken bovendien de optimale state-value function $v_*(s)$ en optimale action-value function $q_*(s,a)$:

$$v_{\pi_*}(s) = v_*(s) = \max_\pi v_\pi(s)$$
$$q_{\pi_*}(s,a) = q_*(s,a) = \max_\pi q_\pi(s,a)$$

De optimale state-value en action-value function zijn respectievelijk de maximale state-value function en de maximale action-value function over alle policies. De optimale policy is deterministisch en vinden we door de optimale action-value function $q_*(s,a)$ te maximaliseren:

$$\pi_*(a|s) = \begin{cases} 1 \text{ als } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s,a) \\ 0 \qquad\qquad\qquad \text{anders} \end{cases}$$

Hieruit volgt dat we onmiddellijk deze optimale policy π_* kunnen bepalen wanneer we de optimale action-value function $q_*(s,a)$ hebben gevonden.

Ook voor de optimale value functions kunnen we recursieve vergelijkingen opstellen, de Bellman optimality equations:

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s',a')$$

Wanneer men het in RL heeft over de Bellman vergelijkingen, dan bedoelt men meestal deze vergelijkingen, die niet lineair zijn, en met iteratieve algoritmes moeten opgelost worden. Deze worden in de volgende hoofdstukken besproken.

3. Plannen met Dynamic Programming

In het vorige hoofdstuk hebben we het Markov Decision Process (MDP) uitgebreid besproken, en uitgelegd hoe we kleine MDPs algebraïsch kunnen oplossen. Maar we hebben ook aangegeven dat deze oplossingswijze niet mogelijk is voor grote MDPs. In de volgende hoofdstukken bespreken we iteratieve oplossingsmethodes waarmee we wel grote MDPs kunnen oplossen. Cruciaal hierbij is een goed begrip van Dynamic Programming. In dit hoofdstuk bespreken we daarom eerst deze fundamentele wiskundige methode, die we gaan toepassen voor het oplossen van problemen i.v.m. planning. D.w.z. dat bij deze problemen het model van de omgeving, het MDP, gekend is. Daarbij gaan we twee soorten problemen oplossen: prediction en control. Zoals we reeds zagen wordt bij prediction een gegeven policy geëvalueerd. Dit is het eerste type probleem dat we gaan oplossen. De methode die we daarvoor gebruiken, kunnen we ook iteratief toepassen bij control problemen waarin we de optimale policy wordt bepaald. Twee algoritmes die hiervoor worden gebruikt zijn policy iteration en value iteration.

3.1. Wat is Dynamic Programming?

Dynamic programming (DP) is zowel een wiskundige optimalisatiemethode als een programmeermethode, ontwikkeld door Richard Bellman in de jaren ’50 van de vorige eeuw. “Dynamic” verwijst naar het feit dat het om problemen gaat met een sequentiële of tijdsafhankelijke component. “Programming” moet in eerste instantie wiskundig geïnterpreteerd worden als een optimalisatiemethode, een term die we ook in linear programming terugvinden. Wiskundigen bedoelen hiermee het optimaliseren van een “program”, wat hetzelfde is als dat wat men in RL een policy noemt. Ondertussen is dynamic programming ook een programmeermethode geworden, maar de originele betekenis is dus de wiskundige. Op wikipedia kan je een stuk lezen uit Bellman’s biografie waarin hij uitlegt waarom hij die naam heeft gekozen: https://en.wikipedia.org/wiki/Dynamic_programming#History.

DP is een zeer algemene oplossingsmethode, die niet alleen in RL wordt toegepast, maar ook in bioinformatica (bv. lattice modellen, sequence alignment), grafentheorie (bv. kortste-pad algoritme), grafische modellen (bv. Viterbi algoritme), scheduling, enzovoort. Met DP kunnen complexe problemen worden opgelost door deze problemen op te delen in subproblemen, deze subproblemen op te lossen, en de oplossingen van deze subproblemen samen te voegen tot de algemene oplossing. De problemen die met DP kunnen worden opgelost moeten aan twee eigenschappen voldoen. Ten eerste moet er een optimale substructuur zijn waardoor Bellman’s “principle of optimality” toepasbaar is. Dit wil zeggen dat het probleem in twee of meer onderdelen kan opgedeeld worden en dat de oplossing voor die onderdelen weergeeft hoe de algemene oplossing kan gevonden worden. Wanneer dit principe toepasbaar is, dan kan het probleem effectief in subproblemen opgesplitst worden, en kan men een verdeel-en-heers strategie toepassen. Ten tweede moeten de subproblemen elkaar overlappen, of anders gezegd, ze moeten verschillende keren terugkomen, waardoor we hun oplossingen kunnen opslaan en hergebruiken.

Met DP kunnen recursieve problemen opgelost worden zoals het vinden van het kortste pad (Engels: shortest path) in een graaf (Engels: graph). Als we de kortste weg tussen A en B willen vinden, dan kunnen we dit probleem intuïtief oplossen door eerst de kortste weg te vinden tussen A en een punt tussen A en B, om daarna de kortste weg te zoeken tussen dat tussenliggende punt en B. Dit is ook het principe dat we bij het oplossen van een MDP gaan toepassen m.b.v. de recursieve Bellman vergelijkingen. De Bellman optimality equation splitst het probleem effectief op in twee subproblemen, door eerst het optimale gedrag voor één stap te gaan bepalen en daarbij het optimale gedrag op te tellen dat volgt op deze eerste stap. De value function slaat de oplossingen van de verschillende stappen op en hergebruikt ze ook bij het bepalen van de algemene oplossing.

Vandaar ook dat de value function een belangrijk concept is in RL. Een MDP voldoet dus aan de twee eigenschappen om DP te kunnen toepassen en dit is wat we ook gaan doen.

DP kan gebruikt worden bij planning, wat veronderstelt dat de MDP volledig is gekend. Eerst en vooral kunnen we daarbij een gegeven policy evalueren (i.e. prediction). In dit geval bestaat de input uit een MDP $\langle \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{R}}, \gamma \rangle$ en een policy π, die kan gereduceerd worden tot een MRP $\langle \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{P}}^{\boldsymbol{\pi}}, \boldsymbol{\mathcal{R}}^{\boldsymbol{\pi}}, \gamma \rangle$. De output bestaat uit de value function v_π. Maar we kunnen DP ook toepassen om de optimale policy te vinden (i.e. control). In dit geval is de input een MDP $\langle \boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{P}}, \boldsymbol{\mathcal{R}}, \gamma \rangle$ en bestaat de output uit de optimale value function v_* en de optimale policy π_*. Voor het oplossen van dit laatste probleem gaan we iteratief te werk, waarbij we tussentijdse policies gaan evalueren a.d.h.v. de methode die we bij policy evaluation hebben toegepast.

3.2. Policy Evaluation

Algoritme

Wanneer we een gegeven policy voor een gegeven MDP evalueren, dan betekent dat concreet dat we de value-function v_π gaan bepalen die bij die policy hoort. In het vorige hoofdstuk hebben we gezien dat we dat m.b.v. de **Bellman expectation equation** kunnen doen:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$

We zagen ook dat we de MDP kunnen reduceren tot een MRP door de bovenstaande vergelijking te herschikken:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a v(s') = \mathcal{R}_s^\pi + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^\pi v(s')$$

Als we alle toestanden beschouwen, dan bekomen we een matrixstelsel:

$$\boldsymbol{v_\pi} = \boldsymbol{\mathcal{R}^\pi} + \gamma \boldsymbol{\mathcal{P}^\pi} \boldsymbol{v_\pi}$$

Dat kunnen we algebraïsch oplossen:

$$\boldsymbol{v_\pi} = [\boldsymbol{I} - \gamma \boldsymbol{\mathcal{P}^\pi}]^{-1} \boldsymbol{\mathcal{R}^\pi}$$

Voor kleine MDPs is deze oplossingsmethode uiterst geschikt, maar bij grote MDPs kan men beter een **iteratieve procedure** toepassen. We starten met willekeurige waarden voor de value function v, vullen die in in de Bellman vergelijking, en krijgen zo een set nieuwe waarden voor v. Dit herhalen we tot we de oplossing v_π hebben gevonden:

$$\forall s: \quad v_0(s) \to v_1(s) \to v_2(s) \to \cdots \to v_\pi(s)$$

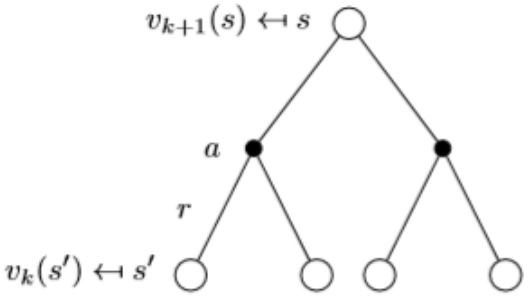
De initiële waarden zijn $v_0(s)$. In dit algoritme gaan we dus in elke iteratiestap de value function *v(s)* van alle toestanden *s* updaten aan de hand van de value function *v(s')* van de volgende toestanden *s'*:

$$\forall s: \quad v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

met *k* de iteratiestap. Omdat we telkens alle toestanden *s* beschouwen, noemen we dit **synchronous backups**. Het toepassen van asynchronous backups bespreken we later. In matrixvorm kunnen we dit als volgt noteren:

$$\boldsymbol{v}^{k+1} = \boldsymbol{\mathcal{R}^\pi} + \gamma \boldsymbol{\mathcal{P}^\pi} \boldsymbol{v}^k$$

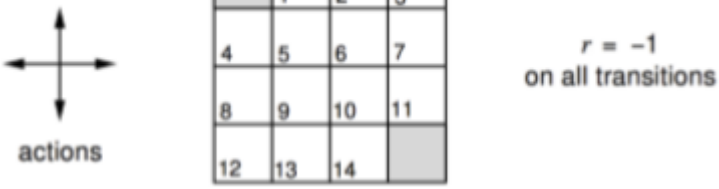
Schematisch stellen we dit als volgt voor:



Met behulp van "contraction mapping" kan men wiskundig bewijzen dat dit iteratieve schema steeds convergeert naar v_π, ongeacht de initiële waarden die men voor v gebruikt.

Voorbeeld: 4 x 4 gridworld

In dit voorbeeld gaan we een 4 x 4 gridworld bekijken waarin een agent zich voortbeweegt volgens een random walk policy. D.w.z. dat hij telkens 1 vakje naar boven (north), naar beneden (south), naar rechts (east) of naar links (west) zal opschuiven, en dat met een kans van 25%. Wanneer de agent tegen de rand botst, dan blijft hij op het vakje staan. Er zijn 14 tussentijdse toestanden, genummerd van 1 t.e.m. 14, en 2 eindtoestanden, die op de figuur grijs zijn gekleurd. Bij elke stap krijgt de agent een reward *r* van -1.



We gaan nu een R script opstellen om deze random policy te evalueren door de state-value function *v* af te leiden. Dit gaan we doen met de directe algebraïsche oplossingsmethode uit het vorige hoofdstuk en met de iteratieve methode die we zojuist hebben uiteengezet.

We starten met de gegevens, nl. de toestanden, de acties, de policy en de reward:

```
# grid dimension
n = 4

# states
ns = n^2
states = seq(0, ns-1, length.out=ns)

# actions
actions = c("n", "e", "s", "w")
na = length(actions)

# policy
policy = 0.25

# reward
r = -1
```

We definiëren ook een functie om het vakje met coördinaat (rij, kolom) om te zetten naar een lineaire index startend bij 1 en eindigend bij 16. De rijen worden van boven naar beneden genummerd, de kolommen van links naar rechts, beiden startend bij 1. Op de figuur worden de

vakjes van 0 t.e.m. 15 genummerd, en dus zal de functie vakje 9 op de figuur in rij 3 en kolom 2 een index gelijk aan 10 geven. We beginnen hier vanaf 1 te tellen omdat R dat ook doet, en dus komt vakje 9 overeen met de 10de toestand. De eindtoestanden hebben dus index 1 en 16.

```
# function to convert (row, col) into state index
state = function(row, col) (row-1)*n + col
```

Aan de hand van deze functie en gebruik makend van 2 lussen die de rijen en kolommen doorlopen, kunnen we nu eenvoudig de state transition matrix $\mathcal{P}$ definiëren:

```
# state transition matrix
P.a = array(0, dim=c(ns, na, ns),
            dimnames=list(from=states,
                          action=actions,
                          to=states))
for (row in 1:n) {
  for (col in 1:n) {
    from = state(row, col) # current state
    # action "north"
    if (row == 1) P.a[from, "n", from] = 1
    else P.a[from, "n", state(row-1, col)] = 1
    # action "east"
    if (col == n) P.a[from, "e", from] = 1
    else P.a[from, "e", state(row, col+1)] = 1
    # action "south"
    if (row == n) P.a[from, "s", from] = 1
    else P.a[from, "s", state(row+1, col)] = 1
    # action "west"
    if (col == 1) P.a[from, "w", from] = 1
    else P.a[from, "w", state(row, col-1)] = 1
  }
}
P.a[1, , ] = 0
P.a[ns, , ] = 0
```

Vervolgens gaan we aan de hand van de policy de gereduceerde state transition matrix $\mathcal{P}^{\pi}$ bepalen:

```
# reduced state transition matrix
P = matrix(nrow=ns, ncol=ns,
           dimnames=list(rownames=states,
                         colnames=states))
for (i in 1:ns) {
  P[i, ] = colSums(policy * P.a[i, , ])
}
print(P)
print(rowSums(P))
```

De gereduceerde reward vector $\mathcal{R}^{\pi}$ is eenvoudig te bepalen:

```
# reduced reward vector
R = rep(r, ns)
R[c(1, ns)] = 0
print(R)
```

Tenslotte hebben we nog de discount factor γ nodig, die we hier aan 1 gelijk stellen:

```
# discount factor
gamma = 1
```

Toepassen van de directe algebraïsche oplossingsmethode geeft de exacte oplossing voor v_{π}:

```
# direct solution
I = diag(1, ns, ns)
v.pi = solve(I - gamma*P, R)
print(matrix(v.pi, nrow=n, byrow=T))
```

De oplossing v_π zien we in de console:

```
> print(matrix(v.pi, nrow=n, byrow=T))
     [,1] [,2] [,3] [,4]
[1,]    0  -14  -20  -22
[2,]  -14  -18  -20  -20
[3,]  -20  -20  -18  -14
[4,]  -22  -20  -14    0
```

Omdat de reward bij elke stap -1 is, geeft de absolute waarde van de berekende value function het aantal stappen aan dat de agent gemiddeld nodig zal hebben om vanuit een vakje in één van de twee eindvakjes terecht te komen wanneer hij een random walk toepast. Vanuit vakje 9 in rij 3 en kolom 2, bijvoorbeeld, zal de agent dus gemiddeld 20 stappen nodig hebben om in één van de grijze vakjes terecht te komen.

Laten we nu de value function iteratief bepalen, waarbij we 1000 iteratiestappen doorlopen:

```
# iterative policy evaluation
niter = 1000
v = matrix(0, nrow=ns, ncol=niter+1,
           dimnames=list(rownames=states,
                         colnames=0:niter))
for (i in 1:niter) {
  v[, i+1] = R + gamma * P %*% v[, i]
}
print(v[, c(1:4, 11, niter+1)])
```

De initiële waarden van de value function *v* stellen we gelijk aan nul. Enkel voor de eindtoestanden is deze waarde vereist, voor de andere toestanden kan dit om het even welke waarde zijn. De tussentijdse oplossingen van *v*, die we na elke iteratie bekomen, slaan we op in de kolommen van matrix `v`. We printen de initiële waarden uit, de tussentijdse oplossing na iteratie 1, 2, 3 en 10, en de eindoplossing na 1000 iteraties. Deze laatste is visueel gelijk aan de exacte oplossing bekomen via de directe oplossingsmethode. In plaats van 1000 iteraties te nemen, hadden we ook met een sluitingscriterium kunnen werken, waarbij we het iteratieve proces stoppen als de maximale wijziging in `v` tussen twee opeenvolgende iteraties kleiner is dan een opgegeven kleine waarde.

```
> print(v[, c(1:4, 11, niter+1)])
        colnames
rownames 0  1     2       3         10 1000
      0  0  0  0.00  0.0000  0.000000    0
      1  0 -1 -1.75 -2.4375 -6.137970  -14
      2  0 -1 -2.00 -2.9375 -8.352356  -20
      3  0 -1 -2.00 -3.0000 -8.967316  -22
      4  0 -1 -1.75 -2.4375 -6.137970  -14
      5  0 -1 -2.00 -2.8750 -7.737396  -18
      6  0 -1 -2.00 -3.0000 -8.427826  -20
      7  0 -1 -2.00 -2.9375 -8.352356  -20
      8  0 -1 -2.00 -2.9375 -8.352356  -20
      9  0 -1 -2.00 -3.0000 -8.427826  -20
      10 0 -1 -2.00 -2.8750 -7.737396  -18
      11 0 -1 -1.75 -2.4375 -6.137970  -14
      12 0 -1 -2.00 -3.0000 -8.967316  -22
      13 0 -1 -2.00 -2.9375 -8.352356  -20
      14 0 -1 -1.75 -2.4375 -6.137970  -14
      15 0  0  0.00  0.0000  0.000000    0
```

3.3. Policy Iteration

Algoritme

In dit hoofdstuk bespreken we hoe we een gegeven policy kunnen verbeteren. Dit gaan we opnieuw op een iteratieve manier doen, en in elke iteratie gaan we twee stappen uitvoeren:

1) **Evaluation (E):** policy π evalueren door v_π te bepalen:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \end{aligned}$$

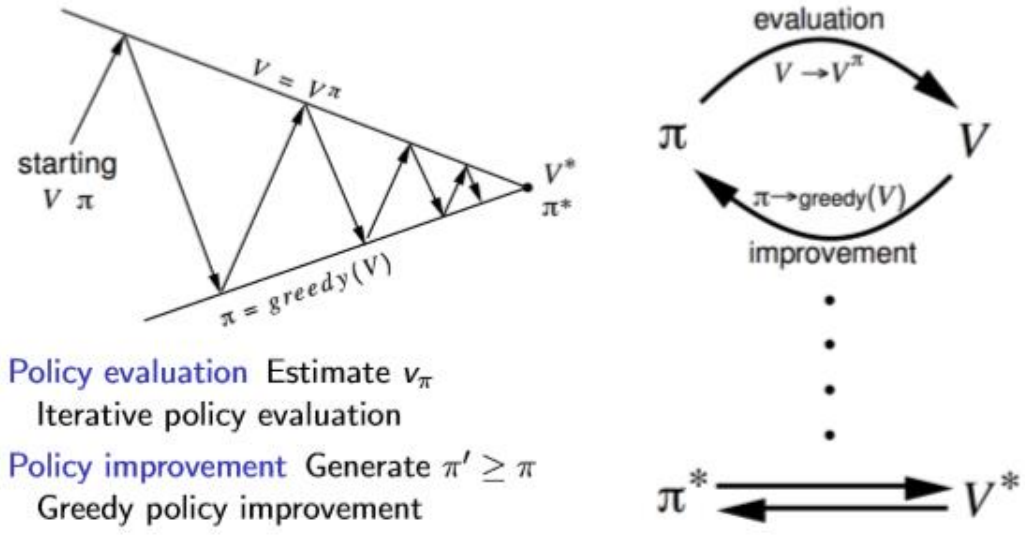
2) **Improvement (I):** policy π verbeteren door "gulzig" (Engels: greedily) te handelen t.o.v. v_π:

$$\pi' = \text{greedy}(v_\pi)$$

Over het algemeen zijn meerdere iteraties van evalueren en verbeteren nodig om de optimale policy π_* te vinden:

$$\pi_0 \overset{E}{\rightarrow} v_{\pi_0} \overset{I}{\rightarrow} \pi_1 \overset{E}{\rightarrow} v_{\pi_1} \overset{I}{\rightarrow} \pi_2 \overset{E}{\rightarrow} v_{\pi_2} \overset{I}{\rightarrow} \dots \overset{I}{\rightarrow} \pi_* \overset{E}{\rightarrow} v_*$$

met π_0 de gegeven policy. Dit iteratieve proces, dat men policy iteration noemt, convergeert altijd tot de optimale policy π_*. Policies $\pi_{k>0}$ komen dus overeen met policy π' in de tweede stap, en zijn dus tussentijdse, deterministische policies, die na vele iteraties uiteindelijk π_* benaderen. In onderstaande figuur wordt dit schematisch voorgesteld, waarbij *V* de state-value function v voorstelt, en *V** de optimale state-value function v_*. Merk op dat de eerste stap van policy evaluation het proces is dat we in vorig punt 3.2 hebben besproken.

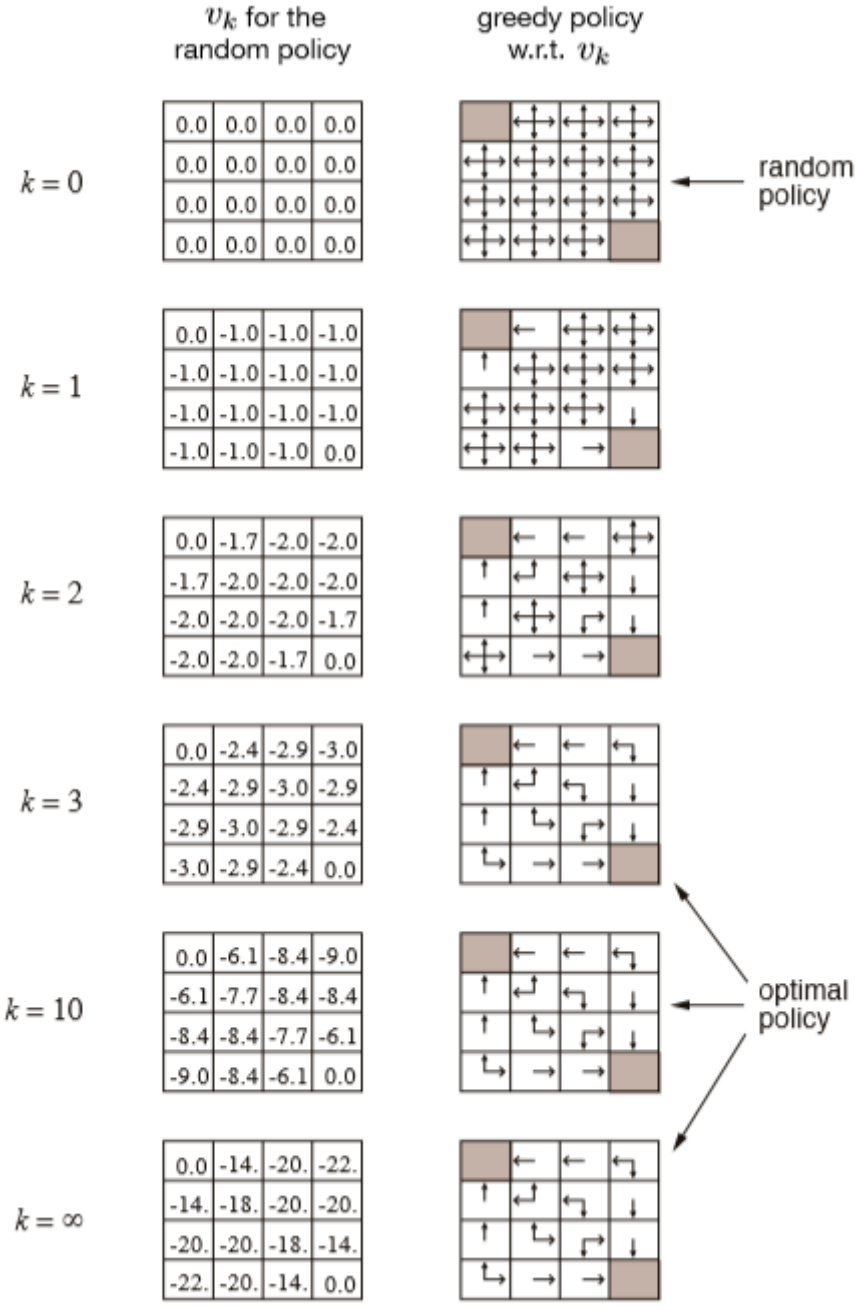


Deze figuur is fundamenteel, omdat de andere algoritmes die we in de volgende hoofdstukken gaan bespreken, eveneens dit principe toepassen, zij het dan wel op een meer gesofisticeerde manier. In de eerste stap van policy evaluation kunnen we dus om het even welk algoritme toepassen dat een gegeven policy evalueert door het bepalen van de value function, en in de policy improvement stap kunnen we ook om het even welk algoritme gaan toepassen die een policy gaat verbeteren a.d.h.v. een gegeven value function.

Voorbeeld: 4 x 4 gridworld

Om uit te leggen wat "greedy policy" betekent, nemen we terug ons voorbeeld van de 4 x 4 gridworld. In dit voorbeeld hebben we de policy niet geüpdatet, maar we kunnen wel na elke iteratie

de policy bepalen die overeenkomt met de tussentijdse waarden van de state-value function *v*. Dit doen we door de random walk, waarin er 25% kans is dat de agent naar boven, beneden, links of rechts gaat, te vervangen door een policy waarbij de agent naar het vakje gaat met de grootste waarde. Is er één vakje waarvan de waarde groter is dan de drie andere aangrenzende vakjes, dan gaat de agent met 100% zekerheid naar dit vakje. Zijn er twee vakjes met de grootste waarde, dan kiest de agent met 50% kans voor één van die twee vakjes. Hieronder zien we in de linker kolom de tussentijdse oplossingen die we in het vorige punt 3.2 hebben uitgewerkt. In de rechter kolom wordt de overeenkomstige policy na elke iteratie weergegeven door pijltjes.



In het begin (*k* = 0) is er de gegeven random policy, na 3 iteraties krijgen we al de optimale policy. Let wel dat de stap van policy improvement normaal gezien pas op het einde van de policy evaluation stap wordt gedaan. De iteraties voor het evalueren van de policy zijn dus zogenaamde inwendige iteraties (Engels: inner iterations). De uitwendige iteraties (Engels: outer iterations) bestaan uit de combinatie van de stappen evaluation en improvement, zoals we hierboven hebben besproken. In dit

geval bereiken we dus de optimale policy al na één uitwendige iteratie, wat in de meeste andere gevallen niet het geval is.

Policy improvement

Laten we nu kijken of de improvement stap inderdaad tot een betere policy leidt en uiteindelijk tot de optimale policy. Stel dat we starten met een deterministische policy π:

$$a = \pi(s)$$

Deze policy kunnen we als volgt verbeteren tot π':

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_\pi(s, a)$$

Dit is wat bedoeld wordt met "gulzig" handelen:

$$\pi' = \operatorname{greedy}(v_\pi)$$

Dit verbetert ook effectief de waarde van de action-value function *q* wanneer de agent één stap verder gaat vanuit toestand *s*:

$$q_\pi\big(s, \pi'(s)\big) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi\big(s, \pi(s)\big) = v_\pi(s)$$

Onder een deterministische policy π geldt inderdaad dat $q_\pi\big(s, \pi(s)\big) = v_\pi(s)$. We gaan dit nu recursief toepassen voor de volgende stappen.

$$\begin{aligned} v_\pi(s) &\leq q_\pi\big(s, \pi'(s)\big) = \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma q_\pi\big(S_{t+1}, \pi'(S_{t+1})\big) \mid S_t = s\big] \\ &\leq \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi\big(S_{t+2}, \pi'(S_{t+2})\big) \mid S_t = s\big] \\ &\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \cdots \mid S_t = s] = v_{\pi'}(s) \end{aligned}$$

Daaruit volgt dus dat het ook de state-value function *v* van toestand *s* verbetert. Wanneer na een aantal iteraties geen verbetering meer optreedt, dan geldt:

$$q_\pi\big(s, \pi'(s)\big) = \max_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi\big(s, \pi(s)\big) = v_\pi(s)$$

waardoor aan de Bellman optimality equation is voldaan:

$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a)$$

en er dus geldt dat $v_\pi(s) = v_*(s)$ en dat voor alle toestanden $s \in \mathcal{S}$. Daaruit volgt dat de uiteindelijk bekomen π de optimale policy π_* is. Intuïtief kunnen we aanvoelen dat we effectief de optimale policy bereiken door de policy stapsgewijs te verbeteren, doordat er een partiële ordening is gedefinieerd op de verzameling van policies. Merk ook op dat we onze redenering zijn gestart met een deterministische policy, maar dat die redenering geldig blijft wanneer we met een stochastische policy starten, aangezien we na de eerste iteratiestap sowieso een deterministische policy krijgen.

Modified policy iteration

Bij policy iteration wordt in de evaluation stap de value function berekend, waarna in de improvement stap de policy wordt afgeleid uit de berekende value function. Maar in het voorbeeld van de 4 x 4 gridworld was de optimale policy al bereikt na slechts 3 inwendige iteraties van de evaluation stap. De daaropvolgende iteraties waren dus eigenlijk nutteloos voor het bepalen van de optimale policy. Om die reden werden enkele uitbreidingen op policy iteration bedacht die de value

function tijdens de evaluation stap enkel gaan benaderen om zo het aantal iteraties te beperken. Dit noemen we modified policy iteration.

In een eerste variant wordt een stopcriterium ε gebruikt. Dit betekent dat het iteratieve proces tijdens de evaluation stap wordt stopgezet als de maximale wijziging in de berekende value function tussen twee opeenvolgende inwendige iteraties kleiner is dan ε. Dit zal het aantal iteraties effectief beperken, maar waarschijnlijk zullen er nog te veel iteraties worden doorlopen in functie van het bepalen van de optimale policy.

In een tweede variant wordt het aantal inwendige iteraties tijdens de evaluation stap beperkt tot k. In het voorbeeld van de gridworld zouden we $k = 3$ kunnen nemen. Ondanks het beperken van het aantal inwendige iteraties zal het algoritme nog steeds convergeren naar de optimale policy. In het extreme geval zouden we zelfs $k = 1$ kunnen nemen. Dit is het value iteration algoritme dat we in het volgende hoofdstuk zullen bespreken.

3.4. Value Iteration

Principle of optimality

Elke optimale policy kan worden opgedeeld in twee componenten:

1) een optimale eerste actie a_*
2) gevolgd door een optimale policy vanuit de volgende toestand s'

Dit kunnen we formeel uitdrukken a.d.h.v het volgende **theorema**: een policy π bereikt de optimale waarde in toestand s, d.i. $v_\pi(s) = v_*(s)$, als en slechts als die policy π de optimale waarde bereikt in elke toestand s' die vanuit toestand s bereikbaar is, d.i. $v_\pi(s') = v_*(s')$.

We gaan dit principe toepassen om het value iteration algoritme op te stellen. Stel dat we de optimale waarden kennen in de toestanden s', m.a.w. we kennen de oplossing voor de subproblemen $v_*(s')$, dan kunnen we de optimale waarde $v_*(s)$ in toestand s bepalen door een stap terug te gaan:

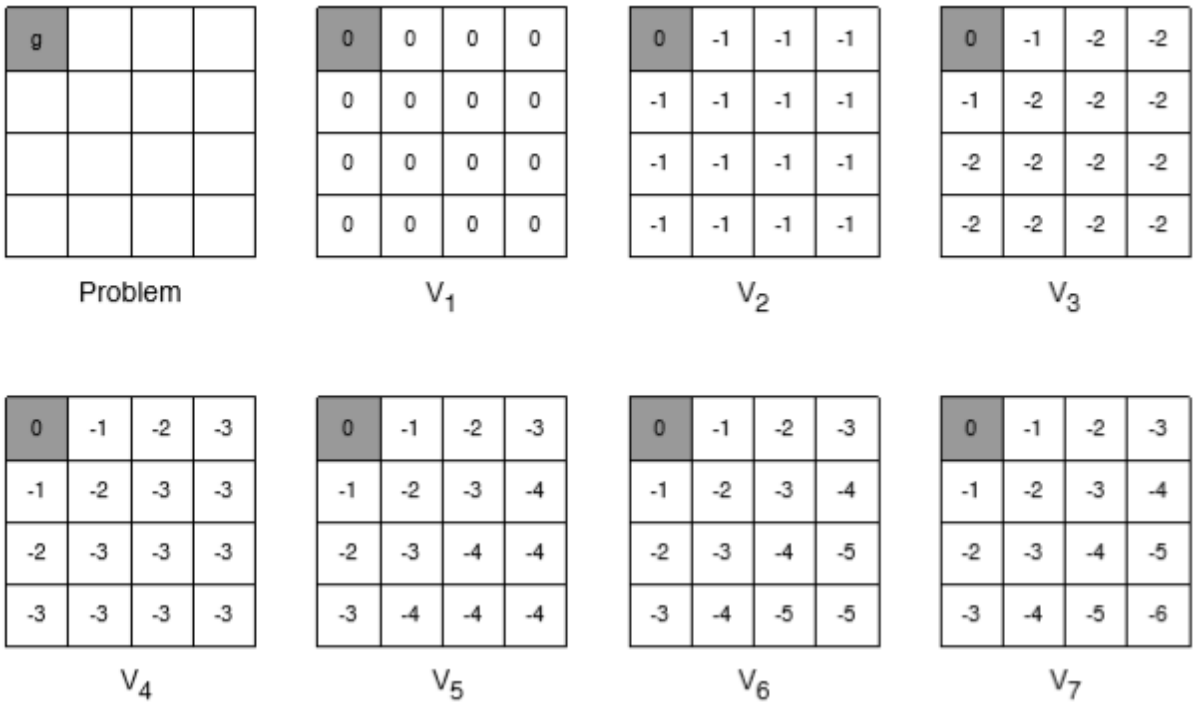
$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$

Dit updaten van $v_*(s)$ gaan we bij value iteration iteratief doen. Intuïtief komt dit neer op het starten bij de finale reward in de eindtoestand om dan vanuit die eindtoestand stap voor stap terug te gaan om de waarden in de andere toestanden te bepalen. Dit is een deterministische manier van redeneren, om inzicht te krijgen in de werking van het algoritme. Maar in de praktijk hoeft men niet noodzakelijk bij een eindtoestand te starten, omdat men alle toestanden doorloopt. Daardoor werkt het algoritme ook bij stochastische MDPs met lussen, en er mogen ook meerdere of zelfs geen eindtoestanden zijn.

Voorbeeld: shortest path

Om de intuïtieve betekenis van value iteration te illustreren, nemen we als voorbeeld opnieuw een gridworld van 4 op 4 met één eindtoestand (grijs vakje). Deze eindtoestand heeft een waarde gelijk aan nul. De andere vakjes zijn de overige toestanden waarvoor we de optimale state-value function v_* gaan bepalen met het value iteration algoritme. De agent kan ook hier in vier richtingen bewegen, wat de vier mogelijk acties zijn: 1 vakje naar boven, naar beneden, naar links of naar rechts. Bij elke actie krijgt hij een reward van -1. De optimale value function zal op die manier het kortste pad (Engels: shortest path) bepalen om vanuit een vakje naar de eindtoestand te gaan.

De oplossing is zeer eenvoudig en daarom op het zicht af te leiden. In de figuur hieronder worden de stappen visueel weergegeven.



We starten het algoritme door alle toestanden een waarde nul te geven (V_1). In de eerste iteratiestap worden alle waarden -1, met uitzondering van de waarde van de eindtoestand die 0 blijft (V_2). Dat is omdat de agent altijd een reward van -1 krijgt als hij uit om het even welk vakje een actie kiest. Daarna krijgen alle vakjes een waarde van -2, met uitzondering van de vakjes die aan de eindtoestand grenzen (V_3). Dat is omdat de agent vanuit een gegeven vakje naar het aangrenzende vakje met de hoogste waarde gaat, en daarbij krijgt hij een reward van -1 plus de waarde van het vakje waar hij naartoe gaat. Die procedure herhalen we nog 4 keer om tot de uiteindelijke optimale waarden te komen (V_7).

We kunnen dit uiteraard ook programmeren, bijvoorbeeld in R. Omdat we weten dat het algoritme na 6 stappen de optimale waarden heeft gevonden, nemen we het aantal iteraties `niter` gelijk aan 7, omdat we ook de initiële stap meerekenen waarin alle waarden V_1 = `v[,,1]` gelijk aan nul worden gezet. De tussentijdse oplossingen V_2 = `v[,,2]` t.e.m. V_7 = `v[,,7]` worden na elke stap uitgeprint. In elke stap worden alle vakjes doorlopen, met uitzondering van de eindtoestand, d.i. vakje (1,1).

In elke iteratiestap wordt voor elk vakje de q-waarde van de vier mogelijke acties als volgt berekend:

$$q(s,a) = \mathcal{R}_s^a + \gamma v(s')$$

In deze formule is $\mathcal{R}_s^a$ de reward gelijk aan -1 en γ de discount factor gelijk aan 1. We krijgen dus:

$$q(s,a) = -1 + v(s')$$

Toestand *s* is het huidige vakje en toestand *s'* het aangrenzende vakje waarin de agent terechtkomt wanneer hij actie *a* kiest. We overlopen alle mogelijke acties, nl. één vakje naar boven (north), naar onder (south), naar links (east) of naar rechts (west) opschuiven. Wanneer de agent tegen een rand botst, dan blijft hij in het huidige vakje staan. Elke q-waarde is dus gelijk aan de reward plus de v-waarde van het vakje waarin de agent terechtkomt. Omdat er per vakje 4 acties mogelijk zijn, krijgen

we 4 *q*-waarden die in vector `q` worden opgeslagen. De nieuwe *v*-waarde van het vakje is tenslotte gelijk aan het maximum van die 4 *q*-waarden.

```
# gridworld dimensions
n = 4

# reward
r = -1

# number of iterations
niter = 7

# initializing v
v = array(0, dim=c(n, n, niter))

# value iteration algorithm
for (iter in 1:(niter-1)) {
  for (irow in 1:n) {
    for (icol in 1:n) {
      # not the terminal state (1, 1)
      if (!(irow==1 & icol==1)) {
        q = rep(0, n)
        # north
        if (irow > 1) q[1] = r + v[irow-1, icol, iter]
        else q[1] = r + v[irow, icol, iter]
        # south
        if (irow < n) q[2] = r + v[irow+1, icol, iter]
        else q[2] = r + v[irow, icol, iter]
        # east
        if (icol < n) q[3] = r + v[irow, icol+1, iter]
        else q[3] = r + v[irow, icol, iter]
        # west
        if (icol > 1) q[4] = r + v[irow, icol-1, iter]
        else q[4] = r + v[irow, icol, iter]
        # maximum
        v[irow,icol, iter+1] = max(q)
      }
    }
  }
  print(v[, , iter+1])
}
```

In de console zien we de tussentijdse oplossingen en de eindoplossing. Hier tonen we matrix `v` voor de laatste drie stappen, en die komen overeen met de oplossing die we visueel hebben afgeleid:

```
     [,1] [,2] [,3] [,4]
[1,]    0   -1   -2   -3
[2,]   -1   -2   -3   -4
[3,]   -2   -3   -4   -4
[4,]   -3   -4   -4   -4
     [,1] [,2] [,3] [,4]
[1,]    0   -1   -2   -3
[2,]   -1   -2   -3   -4
[3,]   -2   -3   -4   -5
[4,]   -3   -4   -5   -5
     [,1] [,2] [,3] [,4]
[1,]    0   -1   -2   -3
[2,]   -1   -2   -3   -4
[3,]   -2   -3   -4   -5
[4,]   -3   -4   -5   -6
```

Algoritme

Het value iteration algoritme in het voorbeeld dat we net hebben uitgewerkt, is zeer eenvoudig, en had als doel om inzicht te krijgen in de werking van het algoritme. In deze paragraaf gaan we het algoritme algemeen formuleren.

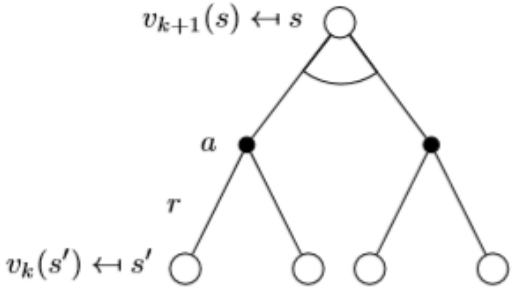
Het probleem is opnieuw dat we een optimale policy in een gegeven MDP moeten vinden. In tegenstelling tot policy iteration, starten we niet met een gegeven policy die we stapsgewijs gaan verbeteren, maar we gaan de **Bellman optimality equation** voor de state-value function iteratief toepassen om tot een oplossing te komen. Deze oplossing is de optimale state-value function waaruit we de optimale policy kunnen afleiden.

$$\forall s: \; v_0(s) \rightarrow v_1(s) \rightarrow v_2(s) \rightarrow \cdots \rightarrow v_*(s) \rightarrow \pi_*$$

We starten dus met initiële waarden v_0 voor de state-value function. Vaak neemt men die initiële waarden gelijk aan nul. Daarna gaan we stap voor stap alle toestanden *s* doorlopen, waarbij we in iteratiestap *k* de waarden v_k updaten a.d.h.v de Bellman optimality equation, waarin we de waarden v_{k-1} uit de vorige iteratiestap invullen:

$$v_k(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \, v_{k-1}(s') \right)$$

Schematisch stellen we dat als volgt voor, waarbij *r* de reward is en de boog onder toestand *s* voor het maximum staat:



Wanneer we bovenstaande vergelijking voor alle toestanden s opstellen, kunnen we dit ook in matrixvorm noteren:

$$\boldsymbol{v}^{k+1} = \max_{a \in \mathcal{A}} \left(\boldsymbol{\mathcal{R}}^{\boldsymbol{a}} + \gamma \boldsymbol{\mathcal{P}}^{\boldsymbol{a}} \boldsymbol{v}^k \right)$$

Omdat we alle toestanden overlopen in elke iteratiestap maken we hier dus gebruik van **synchronous backups**. Zoals we nog zullen zien is dat niet noodzakelijk en kan het proces efficiënter worden uitgevoerd.

Deze procedure convergeert altijd naar de optimale value function v_*, wat ook wiskundig kan worden bewezen. De tussentijdse value function v_k correspondeert niet noodzakelijk met een policy en dus worden geen tussentijdse policies bepaald in dit algoritme. Pas op het einde wordt de optimale policy π_* afgeleid uit de bekomen optimale value function v_*.

Voorbeeld: the gambler's problem

In dit voorbeeld gaan we het algoritme toepassen om het al iets ingewikkelder probleem van de gokker op te lossen, die over een afgerond bedrag tussen 0 en 100 euro beschikt. Hij kan een bedrag inzetten bij het tossen van een munt. Bij kop wint de gokker het ingezette bedrag, bij munt verliest hij dat bedrag. De kans op munt is 40%. De gokker moet bij elke toss een afgerond bedrag inzetten

dat maximaal gelijk is aan het bedrag waarover hij beschikt. Het spel stopt wanneer de gokker in totaal 100 euro heeft gewonnen, of wanneer zijn geld op is.

Dit probleem kunnen we formeel als een MDP definiëren. De agent is de gokker. De toestanden komen overeen met het bedrag waarover de gokker op een bepaald moment in het spel beschikt. Aangezien de bedragen afgerond zijn, kunnen we 101 toestanden definiëren: 0, 1, 2, 3, …, 99, 100.

We schrijven nu een R script om het probleem op te lossen m.b.v. het value iteration algoritme. We beginnen met de toestanden te definiëren:

```
# states
n = 100
S = 0:n
```

De eerste en de laatste toestanden zijn eindtoestanden omdat het spel stopt wanneer de gokker 0 of 100 euro heeft. De kans op munt kennen we toe aan variabele `p`. De rewards `R` zijn in alle toestanden 0, tenzij in toestand 100, want dan wint de gokker, en krijgt hij een reward van 1.

```
# probability
p = 0.4

# rewards
R = c(rep(0, n), 1)
```

De mogelijke acties die de gokker kan uitvoeren komen overeen met de bedragen die hij kan inzetten. Stel dat de gokker 4 euro heeft, dan kan hij 1, 2, 3 of 4 euro inzetten. Vanuit toestand 4 zijn er dus 4 acties mogelijk. Maar dat geldt ook als hij 96 euro heeft, want dan zal hij eveneens 1, 2, 3 of 4 euro inzetten. Meer dan 4 euro inzetten heeft geen zin, aangezien het doel is om 100 euro te verzamelen. De gokker zal ook altijd minimum 1 euro inzetten. Dus de mogelijke acties `A` vanuit state `s` zijn `1:min(s,n-s)`. Stel dat de gokker 96 euro heeft en 3 euro inzet. Dan zal hij ofwel winnen en 99 euro hebben, ofwel verliezen en 93 euro hebben. Die 99 en 93 zijn de mogelijke volgende toestanden vanuit toestand 96 wanneer de actie 3 is. De kans dat het 99 wordt, is `p`; de kans dat het 93 wordt, is 1-`p`. Als mogelijke acties `A` zijn zoals hierboven gedefinieerd, dan zijn alle mogelijke volgende toestanden vanuit toestand `s` bij winst gelijk aan `s+A`, en bij verlies gelijk aan `s-A`.

Nu we het probleem volledig gedefinieerd hebben, kunnen we het value iteration algoritme toepassen om de optimale value function van elke toestand te bepalen. We gaan niet met een stopcriterium werken, maar met een maximum aantal iteraties van 1000. Dit is meer dan voldoende om de optimale waarden te bereiken. We gaan ook alle tussentijdse iteraties bijhouden in matrix `v`. Elke kolom in die matrix bevat dus de tussentijdse value function en de laatste kolom de uiteindelijke optimale value function, of strikt genomen een benadering ervan. Matrix `v` bevat dus 101 rijen, die overeenkomen met de toestanden, en 1001 kolommen, die overeenkomen met het aantal iteraties, waarbij de eerste kolom de initiële waarden bevat die we gelijk stellen aan nul.

```
# number of iterations
niter = 1000

# initial values
V = matrix(0, nrow=n+1, ncol=niter+1)
```

Omdat in R de indices van vectoren en matrices starten bij 1, is de index van een toestand `s` gelijk aan `s+1`. De mogelijke rewards vanuit toestand `s` bij winst is dus gelijk aan `R[s+A+1]`, bij verlies gelijk aan `R[s-A+1]`. Op dezelfde manier worden de waarden van de mogelijke volgende

toestanden geselecteerd. Bij het toepassen van het value iteration algoritme doorlopen we de toestanden 1 t.e.m. 99, en dus niet de eindtoestanden:

```
# value iteration
for (i in 1:niter) {

  # loop through all non-terminal states
  for (s in s[2:n]) {

    # possible actions
    A = 1:min(s, n-s)

    # possible next states
    iwin = s + A + 1
    ilose = s - A + 1

    # apply Bellman optimality equation
    V[s+1, i+1] = max(p * (R[iwin] + V[iwin, i]) +
                      (1 - p) * (R[ilose] + V[ilose, i]))
  }
}

# optimal value of state 100 is 1
V[n+1, ] = 1
```

In de laatste regel zetten we de waarde van toestand 100 gelijk aan de reward van 1.

Merk op dat de rewards in dit voorbeeld ook afhankelijk zijn van de volgende toestand. In plaats van $\mathcal{R}_s^a$ hebben we hier dus met $\mathcal{R}_{ss'}^a$ te maken. De Bellman vergelijking is in dit geval:

$$v_k(s) = \max_{a \in \mathcal{A}} \left(\sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma v_{k-1}(s')] \right)$$

In feite is dit een meer algemene vergelijking, die kan omgezet worden tot de vorm die hierboven werd gegeven:

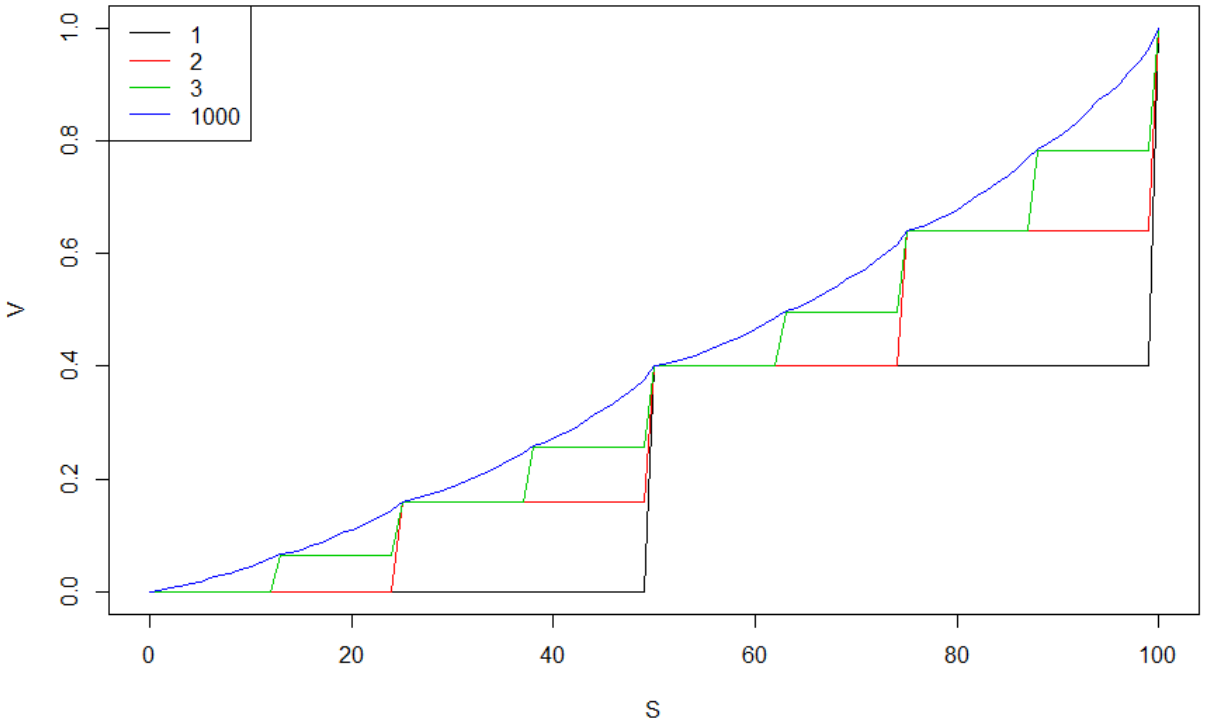
$$\begin{aligned} v_k(s) &= \max_{a \in \mathcal{A}} \left(\sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a [\mathcal{R}_{ss'}^a + \gamma v_{k-1}(s')] \right) \\ &= \max_{a \in \mathcal{A}} \left(\sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \mathcal{R}_{ss'}^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{k-1}(s') \right) \\ &= \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{k-1}(s') \right) \end{aligned}$$

met $\mathcal{R}_s^a = \sum_{s'} \mathcal{P}_{ss'}^a \mathcal{R}_{ss'}^a$.

We maken nu een plot van de optimale value function *v* en ter illustratie tonen we ook de tussentijdse oplossingen die we na 1, 2 en 3 iteraties hebben bekomen:

```
# plot optimal state-value function
matplot(s, V[, c(2:4, niter+1)], type="l", lty=1, col=1:4,
        xlab="S", ylab="V")
legend("topleft", legend=c(2:4, niter+1), lty=1, col=1:4)
```

Dit geeft ons de volgende grafiek:

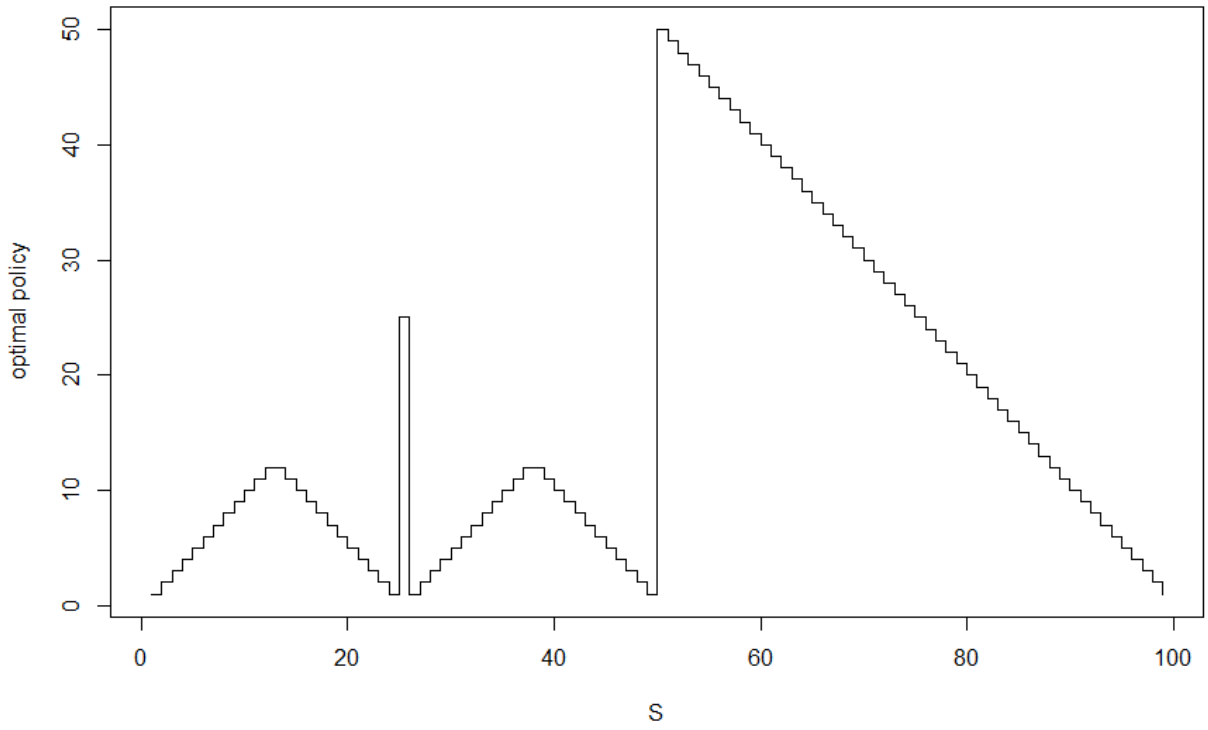


Tenslotte gaan uit de bekomen optimale waarden de optimale policy afleiden. Daarvoor gaan we opnieuw alle toestanden doorlopen en voor elke toestand de actie kiezen die de maximale waarde geeft. Ook al is dat niet nodig in R, toch gaan we de acties van elke toestand met een lus doorlopen en de grootste waarde en bijhorende actie bijhouden in de variabelen `best.q` en `best.a`. We gaan bovendien de beste actie enkel vervangen door de huidige actie als de waarde van de eerste meer dan `eps` kleiner is dan de waarde van de huidige actie. Deze `eps` stellen we gelijk aan 1E-10. Dat doen we omdat het resultaat gevoelig is voor afrondingsfouten. Bovendien zijn er ook meerdere optimale policies mogelijk, zoals blijkt uit mailverkeer met de auteurs Sutton & Barton van het standaardwerk “Reinforcement Learning: An Introduction”, waaruit dit voorbeeld komt: http://incompleteideas.net/book/first/gamblers.html. Onderaan deze link staat ook de oplossing van de auteurs geprogrammeerd in Lisp. Onze R-code voor het bepalen van de optimale policy ziet er als volgt uit:

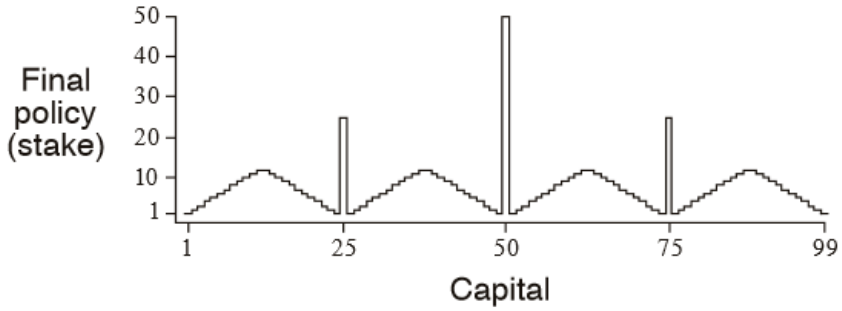
```
# get optimal policy
policy = c()
eps = 1e-10
for (s in S[2:n]) {
  A = 1:min(s, n-s)
  best.q = -1
  best.a = A[1]
  for (a in A) {
    iwin = s + a + 1
    ilose = s - a + 1
    q = p * (R[iwin] + V[iwin, niter+1]) +
        (1 - p) * (R[ilose] + V[ilose, niter+1])
    if (q > (best.q + eps)) {
      best.q = q
      best.a = a
    }
  }
  policy[s] = best.a
}
```

De bekomen optimale policy kunnen we ook plotten:

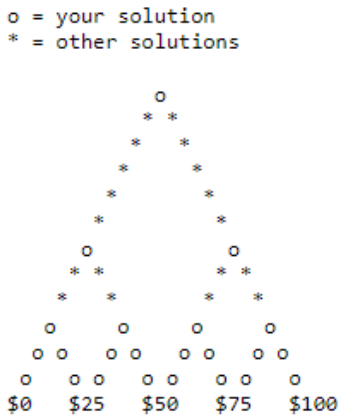
```
# plot optimal policy
plot(s[2:n], policy, type="s", xlab="s", ylab="optimal policy")
```



In hun boek (2de editie) geven Sutton & Barto deze oplossing voor de optimale policy (figure 4.3):



In de discussie met de auteurs, waar we hierboven naar verwezen, blijkt dat dit alle mogelijke oplossingen zijn, en dus is ook onze oplossing correct.



3.5. Uitbreidingen van Dynamic Programming

De algoritmes die we tot nu toe hebben besproken, doorlopen altijd alle toestanden in elke iteratiestap. Dit noemen we synchronous dynamic programming. Maar dit is niet altijd nodig. Daarom heeft men varianten op de synchrone algoritmes bedacht waarin men niet telkens alle toestanden doorloopt. Deze varianten worden verzameld onder de noemer **asynchronous dynamic programming**, en kunnen serieuze winst in rekentijd opleveren. Deze algoritmes convergeren ook altijd naar de optimale oplossing, zolang alle toestanden blijven geselecteerd worden.

In-Place Dynamic Programming

Bij in-place DP gaat men de geüpdatete waarden onmiddellijk gebruiken, in plaats van pas in de volgende iteratiestap. Je zou het kunnen zien als een programmeertruc waarbij er slechts één vector van waarden wordt bijgehouden in plaats van twee. In veel gevallen zal dit het rekenproces ook veel efficiënter maken.

Bij synchrone DP heeft men dus twee vectoren v_{old} en v_{new}, en één iteratiestap ziet er als volgt uit:

$$v_{new}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{old}(s') \right)$$

$$v_{old}(s) \leftarrow v_{new}(s)$$

Bij in-place DP is er slechts één vector v die onmiddellijk wordt geüpdatet:

$$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right)$$

Laten we dat eens illustreren met het zopas uitgewerkte voorbeeld van de gokker. In het R-script dat we hierboven hebben uitgewerkt, hebben we alle tussentijdse oplossingen voor v bijgehouden ter illustratie, maar normaal gezien houdt men enkel die van de vorige iteratie bij. Deze keer gaan we ook een stopcriterium definiëren om te zien welke van de twee algoritmes het snelst een goeie benadering van de optimale value function geeft. Hiervoor moeten we per iteratiestap het maximale absolute verschil Δ berekenen:

$$\Delta = \max_{s \in \mathcal{S}} |v_{new}(s) - v_{old}(s)|$$

Als dat verschil kleiner is dan het opgegeven stopcriterium ε, een zeer kleine positieve waarde, dan is de oplossing voldoende geconvergeerd. Anders gezegd, de oplossing v_{new} is een voldoende nauwkeurige benadering van de exacte oplossing v_*.

Ons nieuwe R-script beginnen we opnieuw met het definiëren van de toestanden `S`, de kans `p` op munt, en de rewards `R`:

```
# states
n = 100
S = 0:n

# probability
p = 0.4

# rewards
R = c(rep(0, n), 1)
```

Daarna gaan we `V` en `V.old` initialiseren als vectoren met allemaal nullen. Het stopcriterium `eps` stellen we gelijk aan 1E-6, en Δ is hier de variabele `dV` die we initieel oneindig groot maken. Het aantal iteraties houden we bij in variabele `i`:

```
# initial values
V = rep(0, n+1)
V.old = V
eps = 1e-6
dV = Inf
i = 0
```

Dan voeren we het synchrone algoritme uit:

```
# synchronous value iteration
while (dV > eps) {

  # loop through all non-terminal states
  for (s in S[2:n]) {

    # possible actions
    A = 1:min(s, n-s)

    # possible next states
    iwin = s + A + 1
    ilose = s - A + 1

    # apply Bellman optimality equation
    V[s+1] = max(p * (R[iwin] + V.old[iwin]) +
                 (1 - p) * (R[ilose] + V.old[ilose]))
  }

  # update i, dV and V.old
  i = i + 1
  dV = max(abs(V-V.old))
  V.old = V

}
```

We printen het aantal iteraties uit en plotten het resultaat:

```
# print number of iterations
print(i)

# optimal value of state 100 is 1
V[n+1] = 1

# plot
plot(S, V)
```

Bij in-place value iteration hebben we geen `V.old` nodig en hoeven we dus enkel `V` te initialiseren:

```
# initial values
V = rep(0, n+1)
eps = 1e-6
dV = Inf
i = 0
```

Bij het in-place algoritme moeten we ook na elke iteratiestap het maximale absolute verschil berekenen. We gaan dit maximale verschil `dV` echter updaten na elke toestand. Vandaar dat we `dV` op min oneindig zetten aan het begin van de lus die alle toestanden doorloopt, zodat we dan het

maximum kunnen nemen van `dV` en het nieuwe berekende absolute verschil na het updaten van de waarde van elke toestand. Om dit verschil te kunnen berekenen, bewaren we de oude waarde in de tijdelijke variabele `tmp`.

```
# in-place value iteration
while (dV > eps) {

  # loop through all non-terminal states
  dV = -Inf
  for (s in S[2:n]) {

    # possible actions
    A = 1:min(s, n-s)

    # possible next states
    iwin = s + A + 1
    ilose = s - A + 1

    # apply Bellman optimality equation
    # and calculate absolute difference
    tmp = V[s+1]
    V[s+1] = max(p * (R[iwin] + V[iwin]) +
                 (1 - p) * (R[ilose] + V[ilose]))
    dV = max(c(dV, abs(V[s+1]-tmp)))
  }

  # update i
  i = i + 1

}
```

Tenslotte gaan we ook het aantal iteraties uitprinten en de oplossing aan de grafiek toevoegen:

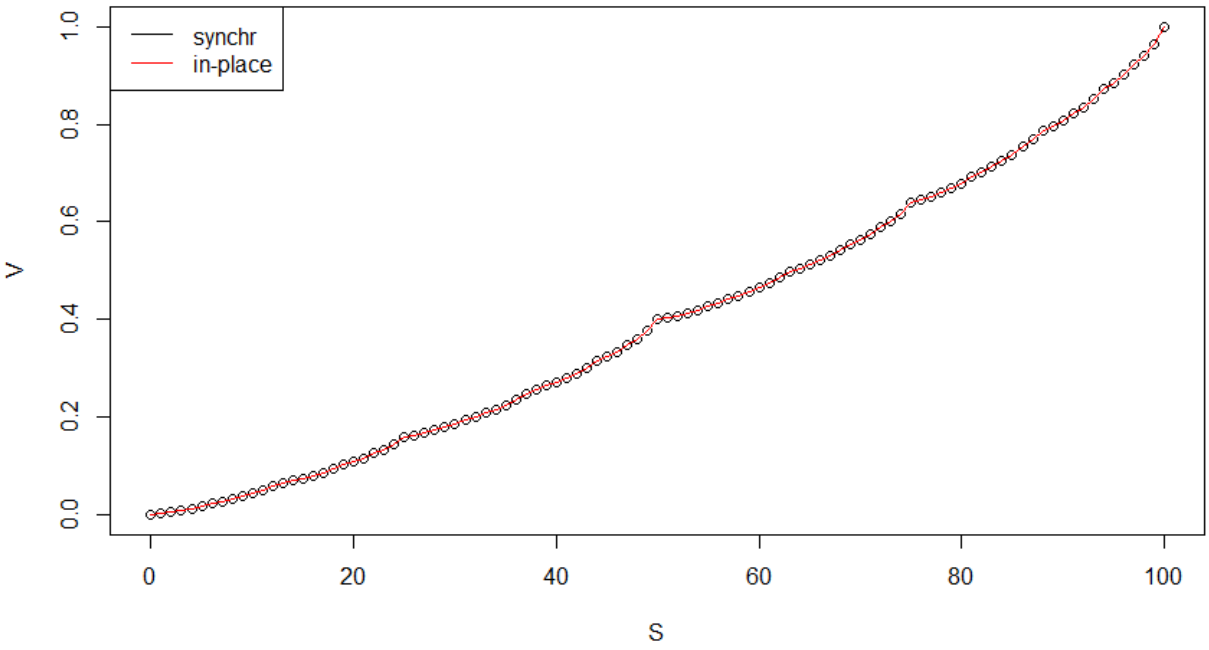
```
# print number of iterations
print(i)

# optimal value of state 100 is 1
V[n+1] = 1

# plot
lines(S, V, col="red")
legend("topleft", legend=c("synchr","in-place"),
       lty=1, col=c("black","red"))
```

Wanneer we dit nieuwe script laten runnen, dan zien we in de console het aantal iteraties bij het synchrone en het in-place algoritme:

```
[1] 20
[1] 12
```

Het in-place algoritme is dus inderdaad efficiënter, aangezien het maar 12 iteraties nodig heeft.

De grafiek toont dat beide oplossingen samenvallen. De zwarte bolletjes geven de oplossing weer berekend met het synchrone algoritme, de rode lijn de oplossing berekend met het in-place algoritme.



Prioritised Sweeping

Een andere mogelijkheid om het synchrone value iteration algoritme efficiënter te maken, is door gebruik te maken van de afwijking op de recursieve Bellman vergelijking, en op basis daarvan een prioritering van de toestanden bij te houden. Deze afwijking kan bijvoorbeeld als volgt worden gedefinieerd:

$$e(s) = \left| \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right) - v(s) \right|$$

De waarde Δ die we in vorige paragraaf hebben gedefinieerd om te vergelijken met het stopcriterium, is het maximum van alle *e(s)*. Bij prioritised sweeping gaan we bij elke iteratie alle errors *e* voor alle toestanden bijhouden, en die gebruiken om in de volgende iteratie de toestanden te ordenen van grote naar kleine *e*. Het is namelijk zo dat we de toestanden in willekeurige volgorde kunnen doorlopen, en dus kunnen we winst boeken door eerst de toestanden te updaten waarvan het verschil tussen twee iteraties het grootst is. Deze procedure kan efficiënt worden geïmplementeerd door gebruik te maken van een zogenaamde priority queue.

Real-Time Dynamic Programming

In real-time DP gaan we ons focussen op de toestanden die relevant zijn voor de agent. Het heeft geen zin om telkens opnieuw toestanden up te daten waar de agent nooit komt. Daarom gaan we een agent samples laten verzamelen. De ervaring van de agent gaan we dan gebruiken bij het selecteren van up te daten toestanden.

Full-width versus Sample Backups

Dit is ook wat de algoritmes doen die we in de volgende hoofdstukken gaan bespreken: ze maken gebruik van samples. Daar waar DP zogenaamde “full-width backups” gebruikt, d.w.z. alle mogelijke acties en daaruit volgende toestanden gaat bekijken, gaan deze methodes “sample backups” toepassen. In het geval van DP zijn full-width backups mogelijk doordat de MDP gekend is. Het nadeel van DP is dat het niet effectief is voor problemen met een zeer groot aantal toestanden. Deze problemen lijden aan wat Bellman de “curse of dimensionality” noemde. D.w.z. dat het aantal toestanden exponentieel toeneemt met het aantal variabelen.

Methoden die gebruik maken van “sample backups” hebben de MDP niet nodig en zijn dus modelvrij. Door gebruik te maken van samples (S, A, R, S’) hoeven ze niet alle acties en toestanden te doorlopen, waardoor ze de “curse of dimensionality” doorbreken en dus ook kunnen toegepast worden bij het oplossen van zeer grote problemen.

3.6. Samenvatting

In dit hoofdstuk hebben we oplossingsmethoden besproken voor planningsproblemen. Bij deze problemen is de MDP gekend. Als ook de policy gegeven is, dan kunnen we voorspellen, wat neerkomt op het evalueren van die policy. Dit kunnen we doen met de methode “Iterative Policy Evaluation”. Control gaat over het vinden van de optimale policy. Daar hebben we twee algoritmes voor uitgewerkt: Policy Iteration en Value Iteration. Onderstaande tabel vat dit samen:

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

De drie algoritmes zijn voorbeelden van dynamic programming, een algemene oplossingsmethode voor recursieve problemen, die deze problemen opdeelt in subproblemen, deze subproblemen oplost, en tenslotte de suboplossingen samenvoegt tot de algemene oplossing.

Bij Iterative Policy Evaluation wordt de value function iteratief geüpdatet via de Bellman expectation equation:

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

Uiteindelijk convergeert dit proces tot de value function v_π die bij de gegeven policy π hoort:

$$v_0 \to v_1 \to v_2 \to \cdots \to v_\pi$$

Bij Policy Iteration wordt Policy Evaluation (E) gecombineerd met Policy Improvement (I) om startend van een willekeurige policy π_0 de optimale policy π_* te vinden:

$$\pi_0 \overset{E}{\to} v_{\pi_0} \overset{I}{\to} \pi_1 \overset{E}{\to} v_{\pi_1} \overset{I}{\to} \pi_2 \overset{E}{\to} v_{\pi_2} \overset{I}{\to} \dots \overset{I}{\to} \pi_*$$

In de evaluation stap wordt ook de Bellman expectation equation toegepast om de bijhorende value-function te bepalen, waarna in de improvement stap de policy wordt verbeterd door “greedy” te bewegen naar de toestand met de grootste waarde. Bij het toepassen van Iteratieve Policy Evaluation in de evaluation stap blijkt echter dat het niet nodig is om telkens de exacte value-function te bepalen. Men kan dus het aantal inwendige iteraties beperken tot een klein aantal, wat men Modified Policy Iteration noemt.

In het extreme geval kan men zelfs de Iterative Policy Evaluation beperken tot 1 stap. Dit algoritme noemt men Value Iteration. Hierin wordt de Bellman optimality equation toegepast, waardoor het niet nodig is om telkens expliciet de policy te gaan updaten. Men hoeft zelfs niet te starten van een

policy, maar kan gewoon met een willekeurige value function v_0 starten. Het algoritme convergeert steeds naar de optimale value-function v_*, waaruit men dan de optimale policy π_* kan afleiden:

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_* \rightarrow \pi_*$$

Het iteratief updaten van de Bellman optimality equation gebeurt als volgt:

$$v_k(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{k-1}(s') \right)$$

Wanneer we in elke iteratiestap alle toestanden doorlopen en we v_k updaten aan de hand van de waarden v_{k-1} uit de vorige iteratiestap, dan spreken we van synchronous dynamic programming. Maar het kan ook efficiënter door asynchroon te werken en bijvoorbeeld een in-place algoritme toe te passen. In dit geval wordt *v* onmiddellijk geüpdatet. Dit kan toegepast worden bij zowel Iterative Policy Evaluation als bij Value Iteration.

De dynamic programming algoritmes die we in dit hoofdstuk hebben besproken, maakten allen gebruik van de state-value function *v*, terwijl het ook mogelijk is om de action-value function *q* toe te passen. Maar wanneer de MDP is gekend, dan is het efficiënter om met de state-value function te werken, omdat men bij de action-value function nog meer mogelijkheden moet overlopen. Wanneer de MDP niet gekend is, dan kan het wel praktischer zijn om de state-value function te gebruiken. Dergelijke modelvrije algoritmes zullen we in de volgende hoofdstukken bespreken.