Het Handelsreizigersprobleem

Het Handelsreizigersprobleem opgelost met behulp van reinforcement learning

Wouter Michel Mulders

Begeleiders:
Prof. Dr. F.M. Spieksma, Dr. T.M. Moerland

Een scriptie voor het afstudeerproject van de opleidingen:
Wiskunde & Informatica

MI, LIACS
Universiteit Leiden
Leiden - Nederland

Samenvatting

Het Handelsreizigersprobleem (HRP) is een probleem dat voorkomt in zowel de wiskunde als in de informatica. Hiertoe moet bij het HRP een route gevonden worden langs alle steden, waarbij je start en eindigt in dezelfde stad, maar waar verder geen enkele stad meer dan één keer bezocht wordt.

Figuur 1: Kortste route die de 15 grootste steden van Duitsland verbindt.

We hebben dynamisch programmeren geimplementerend voor het oplossen van het HRP, hiertoe bekijken we twee methodes van dynamisch programmeren, de oneindige horizon en de eindige horizon. We constateren dat indien het aantal steden groot is het HRP moeilijk op te lossen is. We hebben verder onderzoek gedaan naar het oplossen van het HRP met behulp van reinforcement learning. Hiertoe hebben we verschillende algoritmes bekeken: we hebben Monte Carlo, Sarsa en Q-learning getest met verschillende combinatie van strategieën, namelijk ϵ-greedy, Gradient Temperature, Lower Confidence Bound en Optimistic Initialisation. We hebben hierbij gevarieerd in de grootte van de graaf, Q-learning versus SARSA, verschillende verdelingen bekeken en tot slot hebben we de toestandsruimte groter gemaakt, waarbij we ook gekeken hebben of het uitmaakt om de toestandsruimte tijdsafhankelijk te maken. We zijn tot de conclusie gekomen dat reinforcement learning een bijdrage kan leveren tot het vinden van een sub-optimale oplossing voor het HRP. De combinatie Q-learning met Optimistic Initialisation werkt het beste voor het vinden van een sub-optimale oplossing, maar ook de oplossing voor de tijdsafhankelijke toestandsruimte werkt goed. Er is verder onderzoek nodig naar de combinatie van Optimistic Initialisation met een tijdsafhankelijk toestandsruimte en er zou nog vooruitgang geboekt kunnen worden naar hoe de beginwaarden van de toestandsruimte geïnitialiseerd wordt voor de Optimistic Initialisation.

Inhoudsopgave

1 Definities

1.1 Het Handelsreizigersprobleem

Een ongerichte graaf $G = (V, E)$ is een paar (V, E) met V een eindige verzameling knooppunten, en $E \subset V \times V$ een ongeordende verzameling paren knooppunten, ofwel de takken. We noemen een graaf volledig, indien voor alle $i, j \in V$ met $i \neq j$ geldt $(i, j) \in E$. Een volledige ongerichte graaf met $|V| = n$ knopen noteren we met K_n. We beschouwen in deze scriptie alleen volledige grafen. Elke tak heeft een afstand gegeven door de afstandfunctie w:

$$w : E \to \mathbb{R}_{>0}. \tag{1.1.1}$$

We gebruiken de notatie $w_{ij} = w((i, j)) = w(e)$ voor $e = (i, j) \in E$.

Definitie 1.1 (Hamiltonkring)**.** Zij $G = (V, E)$ een ongerichte graaf. Een enkelvoudige keten die alle knooppunten bevat, heet een Hamiltonketen. Indien een Hamiltonketen ook een kring is, noemen we de (Hamilton)-keten een Hamiltonkring.

Definitie 1.2 (Handelsreizigersprobleem)**.** Zij H de verzameling van alle Hamiltonkringen van $G = (V, E)$ en $h \in H$ een Hamiltonkring. Dan definieert h een graaf $G = (V, E)$ welke we voor het gemak h noemen. De takken van $h \in H$ noteren we met E_h. Bepaal h^* met

$$h^* = \operatorname{argmin}\{\sum_{e \in E_h} w(e) \mid h \in H\}. \tag{1.1.2}$$

De graaf h^* noemen we de oplossing van het Handelsreizigersprobleem van G.

We zijn dus op zoek naar een Hamiltonkring h, met $\sum_{h \in E_h} w(e)$ minimaal.

Voorbeeld 1.3 (gewichtenmatrix voor een K_6)**.**

$$\begin{pmatrix} - & 35 & 60 & 23 & 33 & 19 \\ 35 & - & 47 & 42 & 21 & 26 \\ 60 & 47 & - & 36 & 31 & 30 \\ 23 & 42 & 36 & - & 70 & 17 \\ 33 & 21 & 31 & 70 & - & 45 \\ 19 & 26 & 30 & 17 & 45 & - \end{pmatrix}$$

Dan heeft h^* de volgende takkenverzameling:
$E_{h^*} = \{(1,4),(4,3),(3,5),(5,2),(2,6),(6,1)\}$,
met totale afstand 156 (zie: (1)).

We kunnen het HRP ook formuleren als een lineair programmeringsprobleem (LP), dit is gegeven door:

$$\min \left\{ \sum_{i=1}^{n} \sum_{i\neq j, j=1}^{n} w_{ij} x_{ij} \;\middle|\; \begin{array}{ll} \sum_{i\neq j, j=1}^{n} x_{ij} = 1, & 1 \leq i \leq n \\ \sum_{i\neq j, i=1}^{n} x_{ij} = 1, & 1 \leq j \leq n \\ u_i - u_j + (n-1)\cdot x_{ij} \leq (n-2), & 2 \leq i, j \leq n \\ u_i \in \mathbb{Z}, & 2 \leq i \leq n \\ x_{ij} \in \{0,1\}, & 1 \leq i, j \leq n \text{ en } i \neq j \end{array} \right\}.$$

Bovenstaand LP is de zogenaamde Miller–Tucker–Zemlin formulering van het HRP (2). Het bewijs dat dit LP een oplossing geeft van het HRP is daarin te vinden, maar kort uitgelegd. De eerste en tweede voorwaarde zorgen ervoor dat elke toelaatbare oplossing een vereniging is van disjuncte kringen, waarbij alle knooppunten gebruikt worden. Toevoeging van de derde voorwaarde reduceert het tot een Hamiltonkring. Minimaliseren over de geselecteerde takken geeft dat bovenstaand LP een oplossing geeft van het HRP.

Om meer uitleg te geven over reinforcement learning en de methode om een HRP op te lossen, worden hier enige korte definities geven.

Definitie 1.4 (Markov beslissingsketen (MBK))**.** We beschouwen een MBK met tijd $T = 0, 1, ...n$. Op ieder tijdstip T wordt het systeem waargenomen. Het systeem bevindt zich dan in toestand $s_T \in \mathcal{S}_T$ met $\mathcal{S}_T$ de toestandsruimte op tijdstip T. Op tijdstip $0 \leq T \leq n-1$ wordt vervolgens een actie gekozen uit de actieverzameling $\mathcal{A}_{s_T}$. Als actie $a_{s_T} \in \mathcal{A}_{s_T}$ wordt gekozen, dan worden er kosten $c(s_T, a_{s_T})$ gemaakt en bevindt het systeem zich met kans $t(s_{T+1}|a_{s_T}, s_T)$ in toestand $s_{T+1} \in \mathcal{S}_{T+1}$ op tijdstip $T+1$. Op $T = n$ stopt het systeem en kunnen er nog eindkosten zijn.

We gaan ons HRP modelleren als een niet-stationaire MBK. De toestanden in $\mathcal{S}_T$ zijn enkelvoudige ketens van lengte T. Voor $\mathcal{A}_{s_T}$ nemen we alle mogelijke takken die we kunnen toevoegen aan toestand s_T, zodanig dat de tak grenst aan de laatste knoop van de keten s_T, en zodat er geen kring ontstaat na het toevoegen van een tak tenzij er een Hamiltonkring ontstaat.

De transitiefunctie t is deterministisch. In het bijzonder geldt dat $t(s_{T+1}|a_{s_T}, s_{T+1}) = 1$ d.e.s.d.a s_{T+1} de keten s_T is waaraan a_{s_T} is toegevoegd. Voor de gemaakte kosten in toestand s_T en actie $a_{s_T} = (i, j)$ nemen we $c(s_T, a_{s_T}) = w(i, j)$.

We zijn bij het HRP geïnteresseerd in de Hamiltonkring h met totaal gewicht $\sum_{e \in E_h} w(e)$ minimaal. Daarmee zijn we in iedere toestand s_T geïnteresseerd in de verwachte toekomstige kosten. Deze zijn alleen afhankelijk van de acties die we nog gaan selecteren en de toestand waar we ons in dat moment in bevinden. De vraag is welke acties we nog gaan kiezen, daarvoor hebben we het begrip strategie nodig waarvoor we het symbool π gebruiken.

Een strategie π is een rij beslisregels met $\pi(T))$ de beslisregel op tijdstip $0 \leq T \leq n-1$. In het bijzonder is

$$\pi(T) : \ \mathcal{S}_T \times \mathcal{A}_{s_T} \to [0, 1], \tag{1.1.3}$$

een kansverdeling op het domein $\mathcal{A}_{s_T}$ gegeven $s_T \in \mathcal{S}_T$, voor ieder tijdstip T. We definiëren een rij beslisregels $(\pi(T), ...\pi(n))$ vanaf tijdstip T als volgt:

$$\pi_T := (\pi(T), ..., \pi(n)). \tag{1.1.4}$$

Nu kunnen we alle $a_{s_T} \in \mathcal{A}_{s_T}$ invullen in $\pi(T)$ gegeven toestand s_T, daarna wordt er met bepaalde kans een actie a_{s_T} geselecteerd.

Zij nu gegeven een tijdstip $T \in \{0, ..., n\}$. Dan bevinden we ons in toestand s_T. We zijn geïnteresseerd in de verwachte toekomstige kosten gegeven een strategie π_{T+1} en de actie a_{s_T}. We meten dat met de volgende formules:

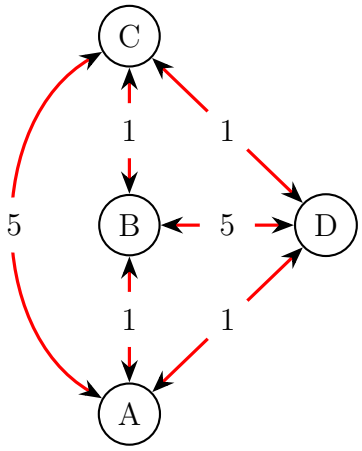
$$Q^{\pi_{T+1},T}(s_T, a_{s_T}) = c(s_T, a_{s_T}) + \mathbb{E}_{\pi_{T+1}}(\sum_{i=T+1}^{n} c(s_i, a_{s_i}) \mid s_T, a_{s_T}). \tag{1.1.5}$$

Per definitie geldt dat $Q^{\pi,T}(s_n, a_{s_n}) = 0$, dat is dus alleen als de toestand een Hamiltonkring is.

Omdat we geïnteresseerd zijn in een optimale strategie, zijn we dus geïnteresseerd in de strategie die $Q^{\pi_{T+1},T}(s_T, a_{s_T})$ minimaliseert.

$$\pi^{*,T}_{T+1} := \underset{\pi_{T+1}}{\operatorname{argmin}} Q^{\pi_{T+1},T}(s_T, a_{s_T}). \tag{1.1.6}$$

$\pi^{*,T}_{T+1}$ is een optimale strategie over tijdsinterval $\{T+1, ..., n\}$. Merk op dat de optimale strategie niet uniek hoeft te zijn, neem als voorbeeld onderstaande graaf van 4×4.



Het is duidelijk dat een optimale oplossing voor dit HRP de kring $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$ is met totale kosten 4. De volgende twee strategieën, π^1, π^2 voldoen hieraan:

$$\boldsymbol{\pi^1} = \begin{array}{c} \\ \text{naar} \end{array} \begin{array}{c} \text{van} \\ \begin{array}{cccc} \text{A} & \text{B} & \text{C} & \text{D} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{l} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \end{array}$$

$$\boldsymbol{\pi^2} = \begin{array}{c} \\ \text{naar} \end{array} \begin{array}{c} \text{van} \\ \begin{array}{cccc} \text{A} & \text{B} & \text{C} & \text{D} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} \text{A} \\ \text{B} \\ \text{C} \\ \text{D} \end{array}$$

De Bellman vergelijking speelt in deze context een belangerijke rol. Dit is een formule om $Q^{\pi}(s_T, a_{s_T})$ recursief te bepalen, in navolging van Formule 1.1.5

$$Q^{\pi}(s_T, a_{s_T}) = c(s_T, a_{s_T}) + \sum_{a_{s_{T+1}} \in \mathcal{A}_{s_{T+1}}} [\pi(s_{T+1}, a_{s_{T+1}}) \cdot Q^{\pi}(s_{T+1}, a_{s_{T+1}})] \quad \text{waarbij } t(s_{T+1} \mid s_T, a_{s_T}) = 1. \tag{1.1.7}$$

De optimale strategie $\pi^*(T+1)$ krijgen we recursief als volgt:

$$Q^{\pi^*}(s_T, a_{s_T}) = c(s_T, a_{s_T}) + \min\{ Q^{\pi^*}(s_{T+1}, a_{s_{T+1}}) \} \quad \text{waarbij } t(s_{T+1} \mid s_T, a_{s_T}) = 1. \tag{1.1.8}$$

1.2 Reinforcement learning voor het HRP

Het HRP is een NP compleet probleem. We zullen reinforcement learning gaan gebruiken om de complexiteit te verkleinen.

Reinforcement learning zal hier kort geïntroduceerd worden. Reinforcement learning is begonnen als methode om de strategie te leren, dat wil zeggen: welke actie moet ik selecteren in een bepaalde toestand om de toekomstige kosten te minimaliseren, in een partieel of volledig observeerbare MBK . Onderstaand zal ook een uitleg gegeven worden hoe we reinforcement learning gebruiken om het HRP zo goed mogelijk op te lossen.

We nemen eerst een stationare strategie, dit is een strategie die tijdens de looptijd van het programma dezelfde beslisregels heeft. We initialiseren vervolgens de $Q^\pi(s_T, a_{s_T})$ (Merk op dat we s_T en a_{s_T} invullen, merk op dat de Q tabel niet alle informatie hoeft te onthouden over de tijd T. Dit kan wel of niet gebeuren en om duidelijke onderscheid te maken in de gebruikte notatie vullen we altijd een s_T en a_{s_T} in) tabel voor alle toestanden en acties op een bepaalde waarde (afhankelijk van de gekozen strategie). Kies een willekeurige starttoestand s_0 (s_0 is één knoop). Volg de van te voren vastgestelde strategie (die vaak afhankelijk is van $Q^\pi(s_T, a_{s_T})$), uit de strategie volgt een actie a_{s_0} waarna we ons bevinden in een nieuwe toestand s_1. Bepaal dan in s_1 alle toelaatbare acties en kies vervolgens met de strategie de actie om in s_2 te komen. Herhaal dit tot dat we ons in toestandand s_n bevinden, dan hebben we een Hamiltonkring. We updaten vervolgens de $Q^\pi(s_T, a_{s_T})$ waardes met een vooraf bepaalde updateregel (meer info over de updateregel volgt in hoofdstuk 4) en met behulp van de kosten van de Hamiltonkring. We blijven het maken van de kringen en de waardes van de $Q^\pi(s_T, a_{s_T})$ een x aantal keer herhalen. We willen op de lange termijn acties selecteren waarvoor de som van de kosten zo laag mogelijk zijn, zodat we een Hamiltonkring krijgen met lage kosten.

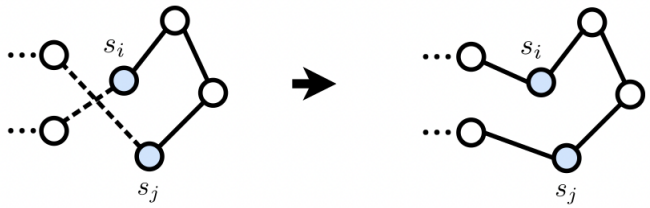
In bovenstaand verhaal over reinforcement learning zijn er nog twee aspecten die goed zijn om te benoemen. Ten eerste de strategie: wij zullen in deze scriptie verschillende strategieën bekijken. Daarin is er altijd een balans tussen exploitatie en de exploratie. Exploitatie betekent dat we takken kiezen die we al vaker zijn tegen gekomen en die ook een lage waarde $Q(s_T, a_{s_T})$ hadden, deze zijn dus goed om te kiezen. Exploratie betekent dat we takken

kiezen die we nog niet vaak zijn tegen gekomen of een hoge $Q(s_T, a_{s_T})$ waarde hadden. Deze takken zijn niet 'goed' geweest in een bepaalde Hamiltonkring maar wellicht zijn ze beter in een andere Hamiltonkring.

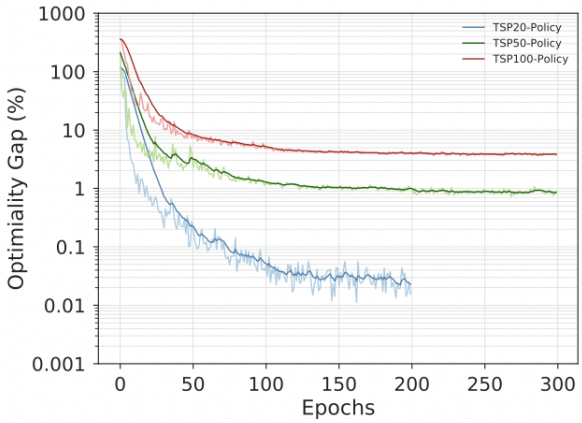
Als laatste de toestandsruimte, er zijn verschillende mogelijkheden om de toestandsruimte te modelleren. We zouden de volledige informatie bij kunnen houden. Dat houdt in dat elk pad wat mogelijk is als toestand, opgeslagen kan worden in de $Q^\pi(s_T, a_{s_T})$ tabel en dus te modelleren is. Wij zullen de toestandsruimte ook kleiner gaan modelleren, dit doen we door alleen de laatst bezochte knoop te onthouden in de $Q^\pi(s_T, a_{s_T})$ tabel. Er zijn nog meer varianten die uitgelegd worden in paragraaf 4.5, door bijvoorbeeld de laatst bezochte k knopen te onthouden of om voor elke tijdstip T de laatste bezochte knoop te onthouden.

2 Literatuuronderzoek

Door P. R. d O Costa, J. Rhuggenaath, Y. Zhang, en A. Akcay wordt in (3) beschreven hoe reinforcement learning gebruikt kan worden om het Handelsreizigersprobleem op te lossen. Hierbij wordt gebruik gemaakt van de algemene 2-opt-heuristiek. De 2-opt-heuristiek werkt als volgt: maak een Hamiltonkring en verwissel vervolgens twee takken als de nieuwe oplossing lagere kosten heeft. De volgende afbeelding van over de 2-opt-heuristiek verduidelijkt wat er gebeurt.



De reinforcement methode die ze gebruiken is het policy gradient algoritme (in het Nederlands: strategie afgeleide) met behulp van deep learning. Meer informatie hierover is te vinden in het boek van Sutton en Barto (4). De onderzoekers van het artikel hebben drie klassen van grafen onderzocht, namelijk de volledige grafen van 20, 50 en 100 knopen (respectievelijk G_{20}, G_{50} en G_{100}) waarbij ze ook hebben aangenomen dat de gewichten de euclidische afstanden tussen de knopen zijn. In Figuur 2 zie je dat het verschil met het optimum klein is en hiervoor relatief weinig iteraties nodig zijn.



Figuur 2: Bron: zie (3, figuur 3). Optimaliteits verschillen voor 256 validaties over 200 iteraties. In bovenstaande figuur wordt eveneens voor de gewichten de euclidische afstand tussen de steden gebruikt, waarbij het aantal steden $n = 20, 50, 100$ is. We zien in de figuur dat zeker voor een lager aantal steden de waarde goed convergeert.

In een volgend artikel (5) wordt de reinforcement learning methode voor het handelsreizigersprobleem meer benaderd zoals wij dat ook willen aanpakken in deze scriptie. In het artikel wordt eerst besproken waarom je dit met behulp van reinforcement learning wil aanpakken. Ten eerste kan reinforcement learning ervoor zorgen dat actuele veranderingen meteen meegenomen kunnen worden in de volgende oplossing, terwijl bij hedendaagse heuristieken vanuit gegaan wordt dat de gewichten van een graaf statisch zijn (ze veranderen niet door de tijd heen). In het artikel wordt de euclidische afstand gebruikt tussen de steden net als in het artikel hiervoor.

De onderzoekers bestudeerden alleen de Q-learning techniek. Voor meer informatie over Q-learning kan het boek van Sutton en Barto geraadpleegd worden (6), Q-learning wordt in deze scriptie ook besproken in Paragraaf 4.3. Voor nu is het belangrijk om te laten zien dat de methode werkt. Dit is te zien in de volgende grafiek.

Figuur 3: Bron: zie (5, figuur). Q-learning model met een graaf van $n = 50$ en $\epsilon = 0.999$ tot 400 iteraties. Na 400 iteraties wordt ϵ lager gezet om de strategie juiste keuzes te laten maken op de lange termijn, we zien dat het model goed leert op de lange termijn.

In het volgende artikel (7) wordt uitgebreid gesproken over de omzetting van het HRP naar een ander bekend probleem uit de NPC klasse. Ook worden praktische problemen uit het hedendaagse leven als HRP gemodelleerd. De auteurs bespreken ook een paar heuristieken die hier benoemt zullen worden. De eerste is de kortste buur heuristiek, waarvan de pseudocode hieronder staat.

Algoritme 1 Kortste buur heuristiek

Require: $n \geq 3$, met K_n
  $kosten \leftarrow 0$
  $W \leftarrow \{1, 2, ..., n\}$ ▷ alle knopen die we nog kunnen kiezen
  $B = L = 1$ ▷ De startknoop / knoop waar we ons in bevinden
  $W = W \setminus L$ ▷ haal startknoop weg
  $P = 1$ ▷ hier slaan we het gevonden pad op
  while $W \neq \emptyset$ **do**
    selecteer $j \in W$ zodat $w_{Lj} = \min\{w_{Li} \mid \forall i \in W\}$ ▷ selecteer de tak met het minimale gewicht
    $kosten \leftarrow kosten + w_{Lj}$ ▷ update de kosten
    $P = P \cup L$ ▷ update het pad
    $L \leftarrow j$
    $W = W \setminus L$ ▷ Verwijder de geselecteerde knoop uit de knopen die we nog kunnen selecteren;
  end while
  $kosten \leftarrow kosten + w_{LB}$;
  $P = P \cup B$;

Merk op dat Algoritme 1 niet optimaal hoeft te zijn. Als gevolg op de kortste buur heuristiek toegepast op het HRP in Voorbeeld 1.3 geeft de volgende oplossing $1 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 1$ met kosten 168. De optimale oplossing heeft kosten 156.

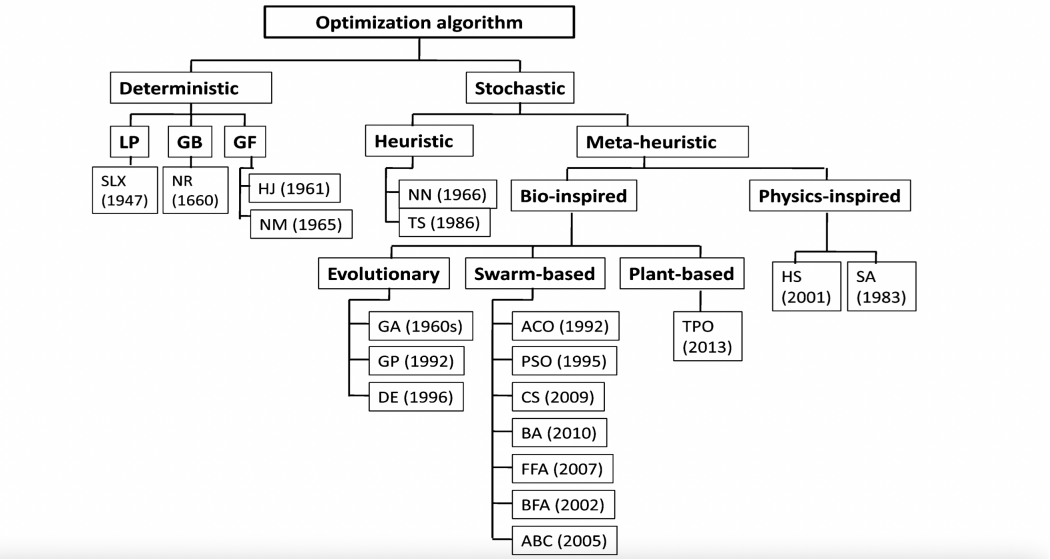
Ook de volgende heuristiek komt in voorgaand artikel (7) voorbij, waarbij er sprake is van een startkring van k knopen. Elke iteratie wordt er een knoop toegevoegd aan de kring. Het toevoegen van de knoop aan de kring gaat in drie stappen. Stap 1, selecteer een knoop die je wil toevoegen. Stap 2, verwijder een tak uit de kring zodat hier de nieuwe knoop kan komen. Stap 3, verbind de nieuwe knoop met de knopen waar net de tak weggelaten is, zodat er weer een kring ontstaat. Het algoritme is als volgt:

Algoritme 2 Takken toevoegen

Require: $n \geq 4$, met K_n
  $R \leftarrow$ kring van minstens k knopen met $k \geq 3$ ▷ de startkring
  $V \leftarrow \{1, 2, ..., n\}$ ▷ alle knopen die we nog kunnen kiezen
  $V \leftarrow V \setminus R$ ▷ verwijder de knopen die al in de kring zitten
  while $V \neq \emptyset$ **do**
    selecteer $j \in V$ met bepaalde methode ▷ selecteer een knoop
    voeg j toe aan kring R, zodat het een goede kring blijft
    $V \leftarrow V \setminus j$ ▷ verwijder de geselecteerde knoop uit V
  end while
  $kosten \leftarrow$ kosten van kring R

De keuze voor de startkring, de knoop die je selecteert, en hoe je die toevoegt aan de kring, bepalen in bovenstaand algoritme de methode. Er zijn twee manieren die "vaak" gebruikt worden. Bij de eerste manier wordt kring R altijd zo klein mogelijk gemaakt (zodat de som van de gewichten van de takken minimaal is) na het toevoegen van een nieuwe knoop. Bij de tweede methode selecteer je een knoop die nog niet in een kring zit, waarbij de afstand tot één van knopen die al wel in de kring zit zo groot mogelijk is. Probeer daarna deze knoop die dan nog niet in de kring zit zo goed mogelijk toe te voegen (zodat de nieuwe kring een zo laag mogelijk totaal gewicht krijgt).

In het volgende artikel (8), wordt over oplossingsmethoden gesproken die gebruikt kunnen worden om NPC problemen aan te pakken, er wordt daarbij onderscheid gemaakt tussen stochastisch en deterministisch. Daarna wordt er een verdere onderverdeling gemaakt. Zoals in het volgende figuur te zien is zijn er veel oplosmogelijkheden voor het HRP probleem (er zijn zelfs nog meer oplossingenstechnieken mogelijk, maar deze zullen we hier nu niet verder bespreken).



Figuur 4: Bron : zie (8). In bovenstaande afbeelding zijn verschillende oplossingsmethodes om (lokale) optima te vinden. Hierbij wordt een onderscheid gemaakt tussen stochastische en deterministische algoritmes. Bovenstaande methodes/algoritmes kunnen gebruikt worden om het HRP op te lossen.

In Figuur 4 zijn heuristieke algoritmes te zien die naar een lokaal of globaal optimum convergeren. Met reinforcement learning zullen wij proberen om het probleem anders aan te pakken dan te zien is in Figuur 4.

3 Dynamisch programmeren

3.1 Introductie

In deze paragraaf zullen we dynamisch programmeren implementeren. Daartoe bekijken we twee verschillende varianten. De eerste is dynamisch programmeren met eindige horizon en de tweede is dynamisch programmeren met oneindige horizon. We zullen beide methodes uitleggen en gebruiken, uiteindelijk zal er ook te zien zijn dat de methodes niet werken voor het HRP met grote grafen.

3.2 Eindige horizon dynamisch programmeren

Zij $s_T \in \mathcal{S}_T$. Dan is s_T een keten van lengte T, voor $T = 0, ..., n-1$ (merk op dat de toestand afhangt van de tijd T). We starten in deze variant met toestand s_0 (pad van lengte 0), voor een vaste knoop $v \in V$ (zonder verlies van algemeenheid starten we in knoop 1). Als we ons dan in toestand s_{n-1} bevinden, moeten we een de tak kiezen naar knoop 1 (om zo de Hamiltonkring te sluiten). Laat $\mathcal{A}_{s_T}$ de toelaatbare actieruimte zijn, in een gegeven toestand s_T.

Op tijd T bekijken we de toestanden s_T en s_{T-1}. We zijn bij elke tijdstip T geïnteresseerd in (waarbij we na actie $a_{s_{T-1}}$ in toestand s_T komen):

$$Q^{\pi^*}(s_{T-1}, a_{s_{T-1}}) = c(s_{T-1}, a_{T-1}) + \min\{Q^{\pi^*}(s_T, a_{s_T}) \mid a_{s_T} \in \mathcal{A}_{s_T}\}. \quad (3.2.1)$$

In bovenstaande is π^* een optimale strategie. Met andere woorden: $Q^{\pi^*}(s_T, a_{s_T})$ is minimaal voor alle s_T en a_{s_T}. We zijn dus nu geïnteresseerd in $Q^{\pi^*}(s_T, a_{s_T})$ voor alle $s_T \in \mathcal{S}$ en $a_{s_T} \in \mathcal{A}_{s_T}$.

Per definitie geldt dat $Q^{\pi^*}(s_n, a_{s_n}) = 0$ voor $a_{s_n} \in \mathcal{A}_{s_n}$. Dan kunnen we bovenstaande formule bepalen door achterwaarts door de toestandsruimte heen te lopen met de bijbehorende acties. Dat wil zeggen dat deze eerst bepaald wordt voor tijdstip T en daarna voor $T-1, T-2, ..., 2, 1$. Dat zorgt ervoor dat alle waardes van $Q^{\pi}(s_T, a_{s_T})$ bepaald worden, waarna de optimale a_{s_T} bepaald kan worden in elke s_T. Het algoritme is als volgt:

Algoritme 3 oplossen HRP met dynamisch programmeren, eindige horizon

Require: $n \geq 3$, met K_n;
$s_0 = 1$
declareer $Q^{\pi^*}(s_T, a_{s_T})$ tabel. $s_T \in \mathcal{S}_T, \forall a_{s_n} \in \mathcal{A}_{s_n}$ voor $T = n, n-1, ..., 1, 0$
$\forall s_n \in \mathcal{S}_n, \forall a_{s_n} \in \mathcal{A}_{s_n} : Q^{\pi^*}(s_n, a_{s_n}) \leftarrow 0$
for each $T = n, n-1, ..., 1$ **do**
  for each $s_{T-1} \in \mathcal{S}_{T-1}$ **do**
    for each $a_{s_{T-1}} \in \mathcal{A}_{s_{T-1}}$ **do**
      $s_T \leftarrow$ **where** $t(s_T | s_{T-1}, a_{s_{T-1}}) = 1$
      $CO \leftarrow \min\{Q^{\pi^*}(s_T, a_{s_T}) \mid a_{s_T} \in \mathcal{A}_{s_T}\}$
      $Q^{\pi^*}(s_{T-1}, a_{s_{T-1}}) \leftarrow c(s_{T-1}, a_{s_{T-1}}) + CO$
    end for;
  end for;
end for;
$\pi^*_{a_{s_T}}(s_T) = \text{argmin}_{a_{s_T}} Q^{\pi^*}(s_T, a_{s_T}), \quad \forall s_T \in \mathcal{S}_T$ voor $T = n-1, ..., 1, 0$

3.3 Dynamisch programmeren, oneindige horizon

Er bestaat daarnaast ook een andere manier van dynamisch programmeren om het HRP op te lossen en dat is met behulp van een oneindige horizon. Bij de oneindige horizon wordt de toestandsruimte willekeurig doorlopen in plaats van achterwaarts door de tijd zoals bij de eindige horizon. Selecteer een willekeurige toestand, update $Q^\pi(s_{T-1}, a_{s_{T-1}})$ met bijhorende formule en herhaal dit.

$$Q^\pi(s_{T-1}, a_{s_{T-1}}) = c(s_{T-1}, a_{s_{T-1}}) + \min\{Q^\pi(s_T, a_{s_T}) \mid a_{s_T} \in \mathcal{A}_{s_T}\}. \quad (3.3.1)$$

Het algoritme is als volgt:

Algoritme 4 oplossen HRP met dynamisch programmeren, oneindige horizon

Require: $n \geq 3$, met K_n
initialisatie van $Q^\pi(s_T, a_{s_T})$ tabel. $\forall s_T \in \mathcal{S}_\mathcal{T}, \forall a_{s_T} \in \mathcal{A}_{s_T}$
$\forall a \in \mathcal{A}, s_T \in \mathcal{S}_\mathcal{T} : Q^\pi(s_T, a_{s_T}) \leftarrow 0$
initialisatie van drempel, met 0 <= drempel $< \infty$
deltaError $\leftarrow$ drempel + 1
while deltaError > drempel **do**
  deltaError $\leftarrow 0$
  for each $s_{T-1} \in \mathcal{S}_{T-1}$ **do**
    for each $a_{s_{T-1}} \in \mathcal{A}_{T-1}$ **do**
      waardeAS $\leftarrow Q^{\pi^*}(s_{T-1}, a_{s_{T-1}})$
      $s_T \leftarrow$ **waar** $t(s_T | s_{T-1}, a_{s_{T-1}}) = 1$
      $CO \leftarrow \min\{Q^\pi(s_T, a_{s_T}) \mid a_{s_T} \in \mathcal{A}_{s_T}\}$
      $Q^\pi(s_{T-1}, a_{s_{T-1}}) \leftarrow c(s_{T-1}, a_{s_{T-1}}) + CO$
      deltaError $\leftarrow \max\{|Q^\pi(s_{T-1}, a_{s_{T-1}}) - \text{waardeAS}|, \text{deltaError}\}$
    end for
  end for
end while
$\pi_{a_{s_T}}(s_T) = \text{argmin}_{a_{s_T}} Q(s_T, a_{s_T}), \quad \forall s_T \in \mathcal{S}_T$ voor $T = n-1, ..., 1, 0$

3.4 Oneindige horizon versus eindige horizon

Bovenstaande pseudocodes, voor het oneindige horizon probleem en het eindige horizon probleem, staan in `MDP.py`. Als we de code runnen voor verschillende K_n kunnen we de grootte van de toestandsruimte printen. Dit geeft de volgende resultaten:

K_n	**Toestandsruimte eindige\oneindige horizon**	
	met $\mathcal{A}$ /actieruimte	**zonder $\mathcal{A}$ /actieruimte**
3	15	5
4	64	16
5	325	65
6	1.956	326
7	13.699	1.957
8	109.600	13.700
9	986.409	109.601
10	9.864.100	986.410
11	108.505.111	9.864.101
12	–	–

Tabel 1: De grootte van de toestandsruimte voor de oneindige horizon en eindige horizon van de code *MDP.py*, waarbij de actieruimte wel of niet mee wordt genomen. In deze tabel wordt de actieruimte groter genomen dan normaal. Zo worden in elke toestand alle acties mee gemodelleerd omdat dit gemakkelijker was. In werkelijkheid kan het programma dus kleiner gemodelleerd worden. Voor $n = 12$ werkte het programma op onze apparatuur niet meer. De toestandsruimte wordt te groot. Bij Python is 1 float variable 8 bytes groot. Stel we nemen aan dat de stap van $n = 11$ naar $n = 12$ minstens een factor 10 groter wordt (de stap van $n = 10$ naar $n = 11$ was ook minstens een factor 10 groter). Inclusief de actieruimte krijgen we een totale geheugenruimte van minstens: $108.505.111 \cdot 10 \cdot 8 \geq 4.8GB$.

De belangrijkste verschillen tussen het oneindige horizon probleem en het eindige horizon probleem staan hieronder opgesomd:

1 Het doorlopen van de toestanden is onafhankelijk van de tijd T voor oneindige horizon, de toestanden worden doorlopen zoals ze geïnitialiseerd zijn.

2 Bij oneindige horizon is het stop criterium afhankelijk van een vooraf bepaalde waarde, die bepaalt hoe groot het verschil is van de waardes van $Q^{\pi}(s_T, a_{s_T})$ ten opzichte van de optimale waardes van $Q^{\pi^*}(s_T, a_{s_T})$.

Merk op dat in dit programma de actieruimte groter wordt gemodelleerd dan echt nodig is. In elke toestand worden er namelijk n acties opgeslagen. Neem als voorbeeld een graaf van K_{10} waarbij we in toestand $1-2-3-4-5-6-7-8-9$ zitten. Dan kunnen we alleen toestand 10 selecteren. Echter de acties $1, 2, 3, 4, 5, 6, 7, 8, 9$ zijn ook aanwezig. Dit verhoogt het benodigde geheugen dat we nodig hebben voor het programma. We hebben dit zo gemaakt omdat dit veel makkelijker te modelleren is.

Voorbeeld eindige horizon versus oneindige horizon K_3

Bekijk Voorbeeld 1.3 waarbij we de graaf aanpassen door de knopen 4, 5 en 6 (kolom 4, 5 en 6, en rij 4, 5 en 6) weg te halen. Dan krijgen we een volledige graaf met 3 knopen. We voeren de algoritmes voor de eindige horizon (links) en de oneindige horizon (rechts) uit.

Initialisatie

actie	**1**	**2**	**3**
s = {1}	–	0	0
s={1, 2}	–	–	0
s={1, 3}	–	0	–
s={1, 2, 3}	0	–	–
s={1, 3, 2}	0	–	–

actie	**1**	**2**	**3**
s = {1}	–	0	0
s= {1, 2}	–	–	0
s= {1, 3}	–	0	–
s= {1, 2, 3}	0	–	–
s= {1, 3, 2}	0	–	–

		eindige horizon	**Actie**				**oneindige horizon**	**Actie**		
			1	**2**	**3**			**1**	**2**	**3**
Iteratie	**Stap**	**Toestand**					**Toestand**			
1	**1**	s = {1, 3, 2}	35	–	–		s = {1}	–	35	60
	2	s = {1, 2, 3}	60	–	–		s = {1, 2}	–	–	47
	3	s = {1, 3}	–	107	–		s = {1, 3}	–	47	–
	4	s = {1, 2}	–	–	82		s = {1, 2, 3}	60	–	–
	5	s = {1}	–	142	142		s = {1, 3, 2}	35	–	–
2	**6**						s = {1}	–	82	107
	7						s = {1, 2}	–	–	107
	8						s = {1, 3}	–	82	–
	9						s = {1, 2, 3}	60	–	–
	10						s = {1, 3, 2}	35	–	–
3	**11**						s = {1}	–	142	142
	12						s = {1, 2}	–	–	107
	13						s = {1, 3}	–	82	–
	14						s = {1, 2, 3}	60	–	–
	15						s = {1, 3, 2}	35	–	–
4	**16**						s = {1}	–	142	142
	17						s = {1, 2}	–	–	107
	18						s = {1, 3}	–	82	–
	19						s = {1, 2, 3}	60	–	–
	20						s = {1, 3, 2}	35	–	–

Tabel 2: Het HRP oplossen met eindige horizon en oneindige horizon, gegeven de graaf K_3. Dit is een triviaal voorbeeld, want er is immers maar één ronde mogelijk die als gevolg meteen de oplossing is van het HRP. We zien in dit voorbeeld dat er 4 keer zoveel iteraties nodig zijn voor de oneindige horizon als voor de eindige horizon.

Laten we nu gaan kijken naar de ruimtecomplexiteit in de algemene formule. We beginnen met de ruimtecomplexiteit zonder de actieruimte van een volledig gerichte graaf G_n van het HRP. Dat wil zeggen, we tellen het aantal enkelvoudige ketens/paden van lengte 0 tot en met $n-1$, waarbij we aanemen dat we een vaste beginknoop hebben.

Stelling 3.1. *De ruimtecomplexiteit van de gerichte volledige graaf G_n (≥ 3) van het HRP zonder actieruimte en eindtoestanden word gegeven door:*

$$\frac{1}{n} \cdot \sum_{i=0}^{n-1} \prod_{j=0}^{i} (n-j). \tag{3.4.1}$$

Bewijs. Zij G_n een volledige gerichte graaf. Voor de lengte van de toestand(en), nemen we de lengte van een enkelvoudig(e) keten/pad aan. De volledige gerichte graaf G_n bevat n toestanden van lengte 0. Voor lengte 1 zijn er voor elke toestand van lengte 0 weer $n-1$ toestanden extra. Omdat we n toestanden hebben van lengte 0 hebben we dus voor de lengte van 1: $n \cdot (n-1)$ toestanden. Voor de lengte van 2 zijn er van lengte 1: $n \cdot (n-1)$ toestanden. Dan kunnen we $(n-2)$ toestanden aan toevoegen. Dat geeft voor de lengte van 2: $n \cdot (n-1) \cdot (n-2)$ toestanden. Dit gaat zo door voor de lengtes $3, 4, ..., n-1$. Omdat we in dit model maar 1 begintoestand hebben, moeten we aan het einde alles delen door n. Dat geeft de volgende formule:

$$\frac{1}{n} \cdot (\sum_{i=0}^{n-1} \prod_{j=0}^{i} (n-j). \tag{3.4.2}$$

□

Uit bovenstaande Formule kunnen we makkelijk de toestandsruimte afleiden van een graaf K_n van het HRP.

Stelling 3.2. *De ruimtecomplexiteit van de graaf K_n (≥ 3) van het HRP zonder actieruimte en eindtoestanden wordt gegeven door:*

$$1 + \frac{1}{2n} \cdot (\sum_{i=1}^{n-1} \prod_{j=0}^{i} (n-j). \tag{3.4.3}$$

Bewijs. Laten we eerst kijken naar de ruimtecomplexiteit van de volledige gerichte graaf G_n van het HRP. Waar zijn daar toestanden die er niet zijn bij graaf K_n van het HRP (of welke weggelaten kunnen worden)? Dat is bij alle toestanden behalve

bij de toestanden van lengte 1. Dat geeft dat we alles moeten delen door 2, behalve bij toestanden van lengte 1. Dat is in bovenstaande formule alleen bij $i = 0$, dat geeft dan het volgende:

$$1 + \frac{1}{n} \cdot \frac{1}{2} \cdot \sum_{i=1}^{n-1} \prod_{j=0}^{i} (n - j). \tag{3.4.4}$$

□

Dan nu de vraag wat de toestandsruimte is met de actieruimte erbij. Voor de toestanden van lengte 0 betekent dat er $n-1$ acties mogelijk zijn. Voor toestandsruimte van lengte 1 zijn dat er $n-2$. Dit gaat zo door totdat we in een toestand zitten waar we n knopen bezocht hebben. Dan is er nog maar 1 actie mogelijk. Dat geeft de volgende formules voor de gerichte volledige graaf G_n en respectievelijk graaf K_n van het HRP in toestandsruimte voor de $Q^\pi(s_T, a_{s_T})$:

$$\frac{1}{n} \cdot \sum_{i=0}^{n-1} \prod_{j=0}^{i} (n - j) \cdot \boldsymbol{f_n(i)}. \tag{3.4.5}$$

$$1 \cdot \boldsymbol{f_n(0)} + \frac{1}{n} \cdot \frac{1}{2} \cdot \sum_{i=1}^{n-1} \prod_{j=0}^{i} (n - j) \cdot \boldsymbol{f_n(i)}. \tag{3.4.6}$$

Hierbij word $f_n(i)$ gegeven door:

$$\boldsymbol{f_n(i)} = \begin{cases} (n - i - 1) & \text{Als } n - i > 1 \\ 1 & \text{Anders} \end{cases}. \tag{3.4.7}$$

Nu de grootte van de toestandsruimte bekend is gaan we verder naar de grootte van het aantal operaties dat het programma nodig heeft. Hierbij zullen we aannemen dat elke operatie gelijke zwaarte/gewicht heeft (wat in de praktijk niet zo is). We kijken daarbij naar het algoritme voor de oneindige horizon en dus niet naar de tijdsduur van het programma *MDP.py*.

Voor dynamisch programmeren met oneindige horizon maakt het veel uit in welke volgorde de toestanden geïnitialiseerd worden. De snelheid van convergeren kan dan veel uitmaken. Om deze redenen zullen het aantal operates twee keer uitgerekend worden. Eén keer waarbij het programma zo slecht (worst case scenario) mogelijk doorlopen kan worden (dus waarbij de toestanden op de slechts mogelijke volgorde geïnitialiseerd worden) en één keer waarbij er zo goed mogelijk door het programma gelopen kan worden. We zullen operaties die in constante tijd

gebeuren (en dus onafhankelijk van n zijn) niet meenemen in het aantal operaties. Daarnaast kijken we alleen naar de graaf K_n voor het HRP.

De initialisatie voor de $Q^\pi(s_T, a_{s_T})$ tabel met op alle waardes 0 wordt gegeven door de formule hierboven. Dat geeft dus al $1 + \frac{1}{n} \cdot \frac{1}{2} \sum_{i=1}^{n-1} \prod_{j=0}^{i} (n-j)$ operaties. Laten we dan kijken naar het gedeelte waarbij we alle toestanden en daarna alle acties doorlopen. Alleen de volgende regel is dan afhankelijk van n in dat gedeelte:

$$Q^\pi(s_{T-1}, a_{s_{T-1}}) \leftarrow c(s_{T-1}, a_{s_{T-1}}) + min\{Q^\pi(s_T, a_{s_T}) | a_{s_T} \in \mathcal{A}_{s_T})\}. \tag{3.4.8}$$

Dan is de vraag hoeveel $a_{s_T} \in \mathcal{A}_{s_T}$ zijn er waarvoor $Q^\pi(s_T, a_{s_T})$ bestaan. Dat zijn er $f_n(i) - 1$ (in de vorige actie waren er namelijk $f_n(i)$ acties mogelijk), na een actie gedaan te hebben zijn dat er nog $f_n(i) - 1$). Waarbij er altijd minstens 1 actie is. Dat geeft dat als je door alle toestanden en acties loopt en daarna het minimum bepaalt van de toekomstige acties je de volgende formule krijgt:

$$\gamma + \beta \cdot \phi = \tag{3.4.9}$$

$$\gamma + \beta \cdot (1 \cdot f_n(0) \cdot \boldsymbol{g_n(0)} + \frac{1}{n} \cdot \frac{1}{2} \cdot \sum_{i=1}^{n-1} \prod_{j=0}^{i} (n-j) \cdot f_n(i) \cdot \boldsymbol{g_n(i)}). \tag{3.4.10}$$

In bovenstaande is:
$\gamma :=$ aantal operaties nodig voor initalisatie algoritme.
$\beta :=$ aantal runs tot dan geconvergeerd.
$\phi :=$ aantal operaties doorlopen van toestanden/acties en minimum bepalen.

Waarbij $g_n(i)$ gegeven wordt door:

$$\boldsymbol{g_n(i)} = \begin{cases} f_n(i) - 1 & \text{Als } f_n(i) > 1 \\ 1 & \text{Anders} \end{cases}. \tag{3.4.11}$$

.
Dan de vraag hoeveel "tot dan geconvergeerd" voorkomt. Laten we positief zijn en stellen dat tabel zo snel mogelijk convergeert. Wat is het snelste? We zien dat toestanden van lengte 0, de toestanden van lengte 1 nodig hebben en dat de toestanden van lengte $n-2$ de waardes van lengten $n-1$ nodig hebben. Elke toestand heeft dus toestanden nodig waarvan de toestandslengte precies 1 langer is dan zijn eigen toestandslengte. Dat betekent dat als we toestanden doorlopen in lengte van $n-1, n-2, ...2, 1, 0$ we meteen de eindoplossing hebben. Echter is de fouterror dan niet 0 voor die ronde. Daarom moeten we nog 1 keer extra door alle toestanden gaan, wat in totaal 2x geeft voor "tot dan geconvergeerd". Het slechte geval is hierbij ook makkelijk af te leiden. Door te starten in omgekeerde volgorde, dus:

$0, 1, ..., n-2, n-1$. Dat geeft dat we n keer moeten lopen voordat hij op de juiste waardes convergeert en daarna nog 1x zodat de fouterror daarna ook op 0 staat. Dat geeft dus $n+1$ gevallen. Dat geeft de volgende formules, waarbij het beste geval en het slechste geval van het het aantal operaties voor het oplossen van de HRP:

$$\frac{1}{n}\cdot(n+\frac{1}{2}\cdot(\sum_{i=1}^{n-1}\prod_{j=0}^{i}(n-j)))+\frac{1}{n}\cdot\mathbf{2}\cdot(n\cdot f_n(0)\cdot g_n(0)+\frac{1}{2}\cdot\sum_{i=1}^{n-1}\prod_{j=0}^{i}(n-j)\cdot f_n(i)\cdot g_n(i)). \quad (3.4.12)$$

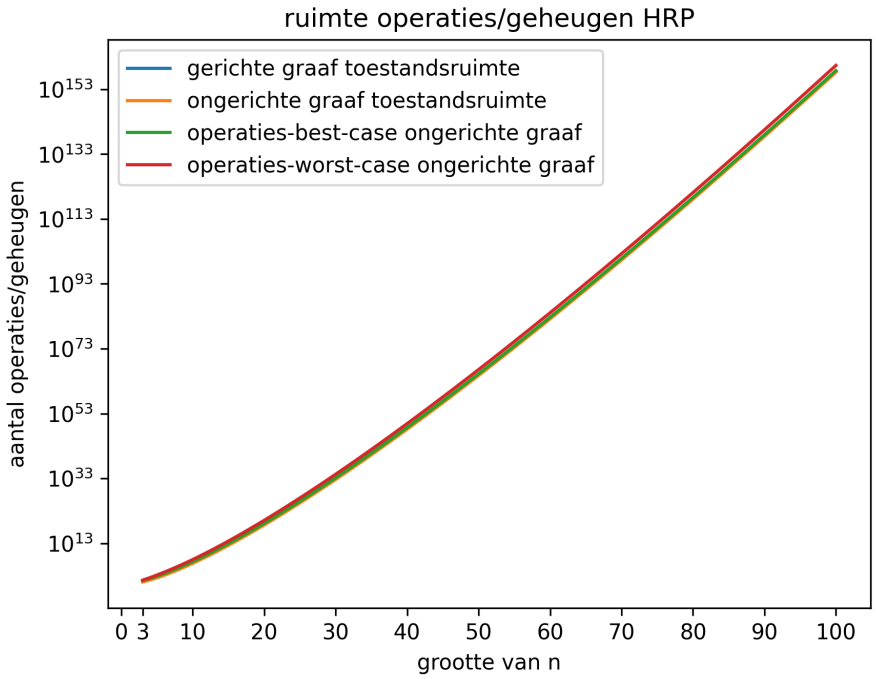
$$\frac{1}{n}\cdot(n+\frac{1}{2}\cdot(\sum_{i=1}^{n-1}\prod_{j=0}^{i}(n-j)))+\frac{1}{n}\cdot\mathbf{(n+1)}\cdot(n\cdot f_n(0)\cdot g_n(0)+\frac{1}{2}\cdot\sum_{i=1}^{n-1}\prod_{j=0}^{i}(n-j)\cdot f_n(i)\cdot g_n(i)). \quad (3.4.13)$$

Opmerking: er bestaat een theoretisch geval waarbij "tot dan geconvergeerd" in theorie in één keer doorlopen kan worden. Dat is als de "drempel" variabele groot genoeg gekozen wordt bij het oneindige horizon algoritme, omdat dit een flauw voorbeeld is, is dat niet meegenomen.

Uit bovenstaande is ook meteen de tijdscomplexiteit af te lezen voor de eindige horizon gegeven door:

$$\frac{1}{n}\cdot(n+\frac{1}{2}\cdot(\sum_{i=1}^{n-1}\prod_{j=0}^{i}(n-j)))+\frac{1}{n}\cdot\mathbf{1}\cdot(n\cdot f_n(0)\cdot g_n(0)+\frac{1}{2}\cdot\sum_{i=1}^{n-1}\prod_{j=0}^{i}(n-j)\cdot f_n(i)\cdot g_n(i)). \quad (3.4.14)$$

Ter verduidelijking voor de lezer bijgaand een figuur om te zien hoeveel operaties er nodig zijn voor verschillende n.



Figuur 5: Aantal operaties/geheugen dat nodig is voor het HRP. In deze grafiek gebruiken we de oplossingsmethode MDP voor verschillende grootte van de instanties n. We zien dat voor grotere n de waardes te hoog worden om met de computer te kunnen berekenen.

3.5 Dynamisch programmeren in de praktijk

In 2020 hebben Europese landen een opdracht gegeven om een supercomputer te bouwen die 516 peta flops rekenkracht heeft (bron: tweakers, zie (9)). Hoe snel zou je het HRP van grootte $n = 100$ kunnen oplossen? Er wordt aangenomen dat: een jaar $365,25$ dagen heeft en het aantal operaties voor $n = 100$ 10^{153} is. Daarnaast is 516 peta flops gelijk aan $552 * 10^{15}$ float berekeningen per seconde. Verder heeft een jaar $365,25 * 24 * 60 * 60 = 31557600$ seconden. Dan volgt dat een supercomputer ongeveer

$$\frac{10^{153}}{31557600 * 552 * 10^{15}} \approx 5.7 * 10^{127}$$

jaar nodig heeft om het HRP van grootte $n = 100$ door te rekenen. Je zou in theorie ook op elke atoom in het universum een supercomputer kunnen neerzetten. Er zijn ongeveer 10^{80} atomen in het universum. Dan kost het de supercomputer $5.7 * 10^{127}/10^{80} = 5.7 * 10^{47}$ jaar.

Voor de opslag zijn er soortgelijke problemen. Omdat we ons dit (qua geheugen en tijd) niet kunnen veroorloven zullen we keuzes moeten maken om dit probleem anders aan te pakken voor grotere n. We kunnen niet alles doorrekenen en we kunnen dus niet het hele probleem kunnen observeren. Met andere woorden, we willen $Q^{\pi}(s_T, a_{s_T})$ benaderen zonder de hele geschiedenis van $Q^{\pi}(s_T, a_{s_T})$ te kennen, zodat we niet teveel geheugen en rekenkracht nodig hebben. De vraag is welke modellen daarvoor geschikt zijn.

4 Reinforcement learning

4.1 Introductie, de methode

We hebben in de inleiding al een introductie gegeven over reinforcement learning. We willen een beperking leggen op de toestandsruimte voor de $Q^\pi(s_T, a_{s_T})$ tabel. Daarnaast hebben we een strategie nodig, waarbij er bepaalt gaat worden welke actie we gaan kiezen. Ook is ons probleem nog afhankelijk van de grootte van de graaf en de kostenfunctie w. Als laatste hebben we ook nog de update regel, namelijk: op welke manier updaten we de Q tabel? We kunnen daarbij 1-staps of meerstaps updates uitvoeren. Een van deze gevolgen is dat we al updates kunnen doen tijdens de episode (1-staps) of pas aan het eind (Monte Carlo), ook alles daar tussenin is mogelijk, waar we bijvoorbeeld na 2, 3 , ... staps updates uitvoeren. Er zijn allemaal variablen die we kunnen aanpassen en met elkaar kunnen vergelijken met als doel een Hamiltonkring vinden waarbij de som van de gewichten zo laag mogelijks is. Hierbij is dan weer het doel om de optimale oplossing te vinden, of eventueel te kunnen benaderen.

4.2 Variatie in grootte van de grafen

Bij dit onderdeel onderzoeken we verschillende strategieën in releatie met verschillende groottes van de grafen. We bespreken eerst de toestandsruimte, de update regel en daarna het algoritme. De methode die we gaan bekijken heet 'Monte Carlo reinforcement learning' en dit is dus alleen de update regel.

Voor de toestandsruimte in deze sectie nemen we aan dat we alleen de laatst bezochte knoop onthouden in de Q tabel. Zo is toestand $1 \to 2$ hetzelfde als $1 \to 4 \to 3 \to 2$, voor de waarde van de Q tabel. Daarnaast introduceren we ook een tweede tabel, de N tabel. Deze tabel houdt bij hoe vaak een bepaalde toestand tegenover een bepaalde actie is gekozen in de Q tabel. Net als in de Q tabel bekijken we daar alleen de laatst bezochte knoop voor een toestand. Dan de update regel, eerst maken we een volledige Hamiltonkring, vervolgens krijgen we kosten c terug voor de tijd $T = 0, ...n-1$. Hierna updaten we de Q tabel als volgt (hierbij is α een hyperparameter die we kunnen bepalen), voor alle tijdstippen $T = 0, ..., n-1$:

$$Q^\pi(s_T, a_{s_T}) = Q^\pi(s_T, a_{s_T}) + \alpha \cdot \left[\left(\sum_{i=T}^{n} c(s_T, a_{s_T}) \right) - Q^\pi(s_T, a_{s_T}) \right] \quad 0 \leq T \leq n-1 \tag{4.2.1}$$

Nadat alle $Q^\pi(s_T, a_{s_T})$ geüpdate zijn starten we in een willekeurige knoop, gegeven de nieuwe waardes van de Q tabel, hierbij gebruiken we een strategie die acties

moet gaan selecteren, de strategie kan explorerend werken of juist exploiterend. Bij exploreren zal de strategie (nieuwe) acties gaan selecteren die nog niet vaak gekozen zijn in eerdere iteraties of een slechte uitkomst hadden. Bij exploitatie selecteren we acties die we al zijn tegengekomen in vorige iteraties en goede uitkomsten hadden. De strategie bepaald of die explorerend of exploiterend moet zijn aan de hand van een formule en met behulp van de Q tabel, de verschillende strategieen introduceren we op de volgende pagina. Met behulp van de strategie en de Q tabel maken we vervolgens weer een Hamiltonkring. Het aantal Hamiltonkringen dat we uiteindelijk maken noemen we het aantal iteraties en korten we af met de letters IT .
Voor meer info over Monte Carlo, zie (10). Het algoritme is dan als volgt:

Algoritme 5 Monte Carlo, HRP

Require: $n \geq 3$, met K_n
Require: strategie
initialisatie van $Q^\pi(s_T, a_{s_T})$ tabel. $s_T \in \mathcal{S}_T, \forall a_{s_T} \in \mathcal{A}_{s_T}$, $T = 0, ..., n-1$
$s_T \in \mathcal{S}_T, \forall a_{s_T} \in \mathcal{A}_T : Q^\pi(s_T, a_{s_T}) \leftarrow$ gebasseerd op strategie, $T = 0, ..., n-1$
$s_T \in \mathcal{S}_T, \forall a_{s_T} \in \mathcal{A}_T : N(s_T, a_{s_T}) \leftarrow 0$ afhankelijk van strategie, $T = 0, ..., n-1$
$aantalIteraties \leftarrow$ gebasseerd op experiment
$IT \leftarrow 0$
while aantalIteraties >= IT **do**
$s_0 \leftarrow$ willekeurige knoop
Genereer Hamiltonkring gebaseerd op strategie π
$\pi : a_{s_0}, c(s_0, a_{s_0}), s_1, a_{s_1}, c(s_1, a_{s_1}), ..., s_n, a_{s_n}, c(s_n, a_{s_n})$
$IT \leftarrow IT + 1$
$G \leftarrow 0$
for each i = n - 1, n - 2, ... 1, 0 **do**
$G \leftarrow G + c(s_i, a_{s_i})$
$Q^\pi(s_i, a_{s_i}) \leftarrow Q^\pi(s_i, a_{s_i}) + \alpha \cdot (G - Q^\pi(s_i, a_{s_i}))$
$N(s_i, a_{s_i}) \leftarrow N(s_i, a_{s_i}) + 1$ (afhankelijk van strategie)
end for
end while

In bovenstaand algoritme zijn er twee regels die we kunnen aanpassen. Ten eerste: de initialisatie van de $Q^\pi(s_T, a_{s_T})$, ten tweede: de kansverdeling voor het nemen van een actie gegeven een toestand s_T. We beschouwen deze twee regels afhankelijk

van de strategie, deze strategieën kunnen we variëren. We definiëren hieronder vier strategieën met bijbehorende initialisaties:

epsilon-greedy (ϵ-greedy), *voor meer info, zie (11)*

Initialisatie:

$Q^{\pi}(s_T, a_{s_T}) = 0$

$N(s_T, a_{s_T})$ tabel niet nodig

strategie:

$$\pi_{\epsilon\text{-greedy}}(s_T, a_{s_T}) = \begin{cases} 1-\epsilon, & \text{als } a_{s_T} = \text{argmin}_{b_{s_T} \in \mathcal{A}_{s_T}}[Q^{\pi}(s_T, b_{s_T})] \\ \frac{\epsilon}{|A_{s_T}|-1}, & \text{anders} \end{cases}.$$

hyperparameters:

$\alpha, \epsilon \in (0, 1]$

Lower Confidence Bound (LCB), *voor meer info, zie (12)*

Initialisatie:

$Q^{\pi}(s_T, a_{s_T}) = 0$

$N(s_T, a_{s_T}) = 0$

strategie:

$$\pi_{\text{LCB}}(s_T, a_{s_T}) = \begin{cases} 1, & \text{als } a_{s_T} = \text{argmin}_{b_{s_T} \in \mathcal{A}_{s_T}}[Q^{\pi}(s_T, b_{s_T}) - c \cdot \sqrt{\frac{ln(IT)}{N(s_T, b_{s_T})}}] \\ 0, & \text{anders} \end{cases}.$$

hyperparameters:

$\alpha, c \in (0, 1]$

Laat a_{s_T} gegeven zijn door de tak (i', j'). Dan is i' en j' op zich zelf een knoop. Deze notatie hebben we nodig voor het volgende:

Optimistic Initialization (OI), *voor meer info, zie (13)*

Initialisatie:

$$Q^{\pi}(s_T, a_{s_T}) = \frac{c}{n(n-2)} \cdot \begin{bmatrix} (n-2)w((i', j')) + \sum_{i \neq i'} \sum_{j \neq j'} w((i, j)) \\ +0.5 \cdot (\sum_{j \neq j'} w((i', j)) + \sum_{i \neq i'} w((i, j')) \end{bmatrix}.$$

$N(s_T, a_{s_T}) =$ tabel niet nodig

strategie:

$$\pi_{\text{OI}}(s_T, a_{s_T}) = \begin{cases} 1, & \text{als } a_T = \text{argmin}_{b_{s_T} \in \mathcal{A}_{s_T}}[Q^{\pi}(s_T, b_{s_T})] \\ 0, & \text{anders} \end{cases}.$$

hyperparameters:

$\alpha, c \in (0, 1]$

Opmerking 4.1. Bovenstaande initialisatie van de Q tabel is gekozen voor de OI strategie. Er volgt nu een korte uitleg over hoe de formule tot stand is gekomen. Gegeven de definitie zou de Q tabel de verwachting van de toekomstige kosten moeten geven voor een bepaalde knoop en actie. De $(n-2)\cdot w((i',j'))$ in de formule zijn de actie en de knoop die samen geselecteerd worden, deze combinatie knoop en actie moet een hoge factor hebben (de kosten komen sowieso terug in de update regel) daarom vermenigvuldiging we met $(n-2)$. De $0.5\cdot(\sum_{j\neq j'} w((i',j)) + \sum_{i\neq i'} w((i,j'))$ zijn de aangrenzende takken/kosten van de knoop en actie die geselecteerd zijn, deze worden later nog met een kleine kans geselecteerd, vandaar de vermenigvuldiging met 0.5 (dan wordt er nog maar één tak geselecteerd van al deze aangrenzende takken). De overige combinaties van takken en knopen hebben twee keer zoveel kans om geselecteerd te worden, dat is het deel van de formule met $\sum_{i\neq i'}\sum_{j\neq j'} w((i,j))$.

Gradient temperature (GT) *voor meer info, zie (14)*

Initialisatie:

$Q^{\pi}(s_T, a_{s_T}) = 0$

$N(s_T, a_{s_T}) =$ tabel niet nodig

strategie:

$$\pi_{\mathrm{GT}}(s_T, a_{s_T}) = \frac{e^{c\cdot 0.001\cdot Q^{\pi}(s_T,a_{s_T})}}{\sum_{b_{s_T}\in\mathcal{A}_{s_T}} e^{c\cdot 0.001\cdot Q^{\pi}(s_T,b_{s_T})}}.$$

hyperparameters:

$\alpha, c \in (0,1]$

Opmerking 4.2. We hebben hierbij de factor 0.0001 toegevoegd in zowel de teller als de noemer. Dit hebben we gedaan om de exploratie parameter in dezelfde range te houden als de andere strategieën .

We voeren het experiment uit voor 3000 iteraties, met verschillende aantallen van de steden. Voor elk aantal steden en elke strategie nemen we één parameter c of ϵ en één parameter α. We draaien het programma voor elke strategie en het aantal steden. Om vervolgens de beste parameters te kiezen voor elke strategie en het aantal steden. We selecteren daarbij de parameters die over de laatste 10 iteraties de laagste waarde teruggeeft. We selecteren daarbij uit de volgende parameters: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. Daarnaast moeten we de gewichten van de graaf nog initialiseren, dat doen we met behulp van de uniformele verdeling ($\mathcal{U}$), we nemen voor iedere tak een gewicht welke we willekeurig trekken uit de uniforme verdeling.
Hierbij is de uniformele verdeling gegeven door:

$$f(x) = \frac{1}{b-a} \quad 0 < a \leq x \leq b < \infty \tag{4.2.2}$$

We nemen vervolgens voor $a = 76$ en $b = 124$ (deze a en b zijn a-select gekozen).

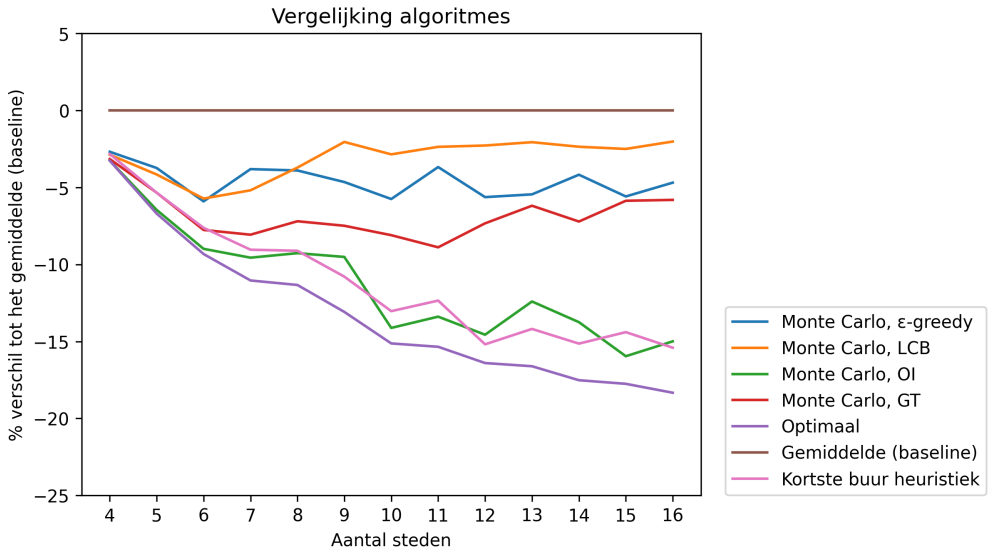
$$w((i,j)) \sim \mathcal{U}_{(76,124)} \quad \forall i, j \in V \text{ met } i \neq j \tag{4.2.3}$$

In de resultaten, zie ook figuur 6, zijn er 3 resultaten die we kort zullen toelichten. De optimale lijn, de gemiddelde (baseline) lijn en de kortste buur heuristiek lijn. De optimale lijn geeft de oplossing voor het HRP. Deze wordt bepaald door een Python-framework, zie (15). Dit Python framework gebruikt dynamisch programmeren als oplossingstechniek. De baseline is wat er uitkomt voor een gemiddelde Hamiltonkring, in formule vorm:

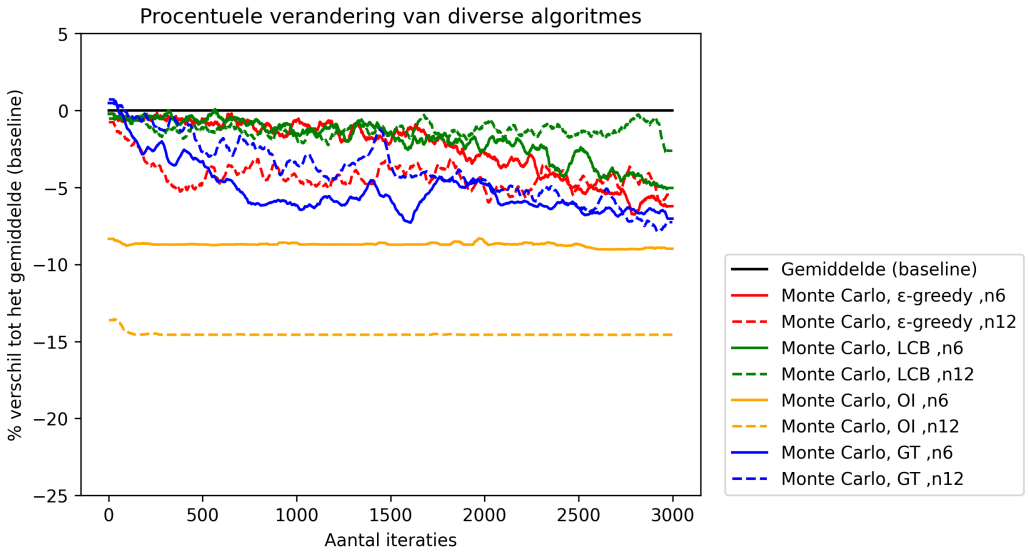
$$\text{baseline} := \frac{\sum_{h \in H} \sum_{e \in e_h} w(e)}{\sum_{h \in H} 1} \tag{4.2.4}$$

De kortste buur heuristiek hebben we behandeld in het Literatuur onderzoek, zie pseudocode 1.

We zijn vooral geïnteresseerd in dat sommige strategieën goed werken en of dat sommige methodes nog doorleren na deze 3000 iteraties. Ook zijn we geïnteresseerd in hoe goed de strategieën werken voor grafen die een groter aantal knopen hebben. De resultaten zijn als volgt:



Figuur 6: **Monte Carlo** algoritme met de strategieen; `ϵ-greedy`, `LCB`, `GT`, `OI`, daarnaast de kortste buur heuristiek en de optimale oplossing. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). Omdat de Monte Carlo algoritmes gedurende het aantal iteraties leren, wordt daar het gemiddelde genomen over de laatste 10 iteraties. De parameters die voor Monte Carlo worden gekozen zijn als volgt, voor elke strategie en elke grootte van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$, $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke grootte van de graaf 10 grafen, waarbij de takken van de graaf gewichten hebben van de uniformele verdeling. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de laatste 10 iteraties gretig worden doorlopen startend in toestand 1 (gegeven de parameters).



Figuur 7: **Monte Carlo** algoritme met de strategieen; ϵ-`greedy`, `LCB`, `GT`, `OI` voor grootte van de grafen $n = 6$ en $n = 12$. De parameters die voor Monte Carlo worden gekozen zijn als volgt, voor elke strategie en elke grootte van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$, $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke grootte van de graaf 10 grafen voor 3000 iteraties, waarbij de takken van de graaf gewichten hebben van de uniformele verdeling. De smoothing window = 51. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de zoveelste iteratie gretig wordt doorlopen, startend in toestand 1 (gegeven de parameters).

Resultaten

Laten we eerst kijken naar de Figuur van 6. We zien bij de grafieken dat de strategieën ϵ-greedy , LCB, GT voor grotere grafen niet naar lagere waardes gaan terwijl de optimale oplossing en de kortste buur heurisitek wel steeds een grotere procentuele verandering hebben tot het gemiddelde. We zien dat de LCB strategie voor kleine grafen nog wel een sub-optimale eind oplossing heeft (grafen van grootte $n = 4, 5, 6$). Maar voor grafen met meer knopen helaas niet meer. De redenen hiervoor is dat LCB te weinig exploreert, we zien in Figuur 7 dat de LCB al snel is uitgeleerd. Het kiezen van een hogere c parameter had ervoor gezorgd dat de resultaten voor LCB waarschijnlijk beter zouden zijn voor grotere grafen. De ϵ-greedy methode leert ook niet optimaal, de reden hiervoor is dat deze methode te veel forceert op exploitatie of exploratie bij een vaste epsilon. Te veel exploratie

heeft het risco dat er kringen ontstaan waarvan de takken niet bijdragen aan de optimalisatie. Te veel exploitatie zorgt ervoor dat er te veel naar dezelfde takken worden gekeken en daarmee te veel dezelfde kringen ontstaan. Er worden te weinig nieuwe varianten (Hamiltonkringen) geïntroduceerd waardoor het programma onvoldoende leert. Een methode om het op te lossen is om de epsilon gedurende het aantal iteraties af te bouwen (lager te zetten), waardoor er in het begin van het leren veel geëxploreerd wordt en naarmate het programma langer leert er steeds meer geëxploiteerd wordt.

We zien voor de GT methode dat deze naar betere waardes convergeert in vergelijking met de LCB en ϵ-greedy methode, dit komt mede doordat GT sneller en langer doorleert en op de lange termijn dus naar betere waardes gaat. Dit is te zien in Figuur 7. Dat de GT beter en langer doorleert komt door de exploratie versus exploitatie methode, de kansen worden gelijker verdeeld waardoor de exploitatie meer plaatsvindt op een verzameling takken in plaats van op één tak. Dit, in tegenstelling tot de LCB en ϵ-greedy methodes die erg gretig zijn om telkens de beste tak te kiezen. Dan hebben we nog de OI strategie, deze presteert bijna even goed als de kortste buur heuristiek. We zien in leergrafiek 7, dat deze nauwelijks leert of zelfs negatief leert. De toevoeging van de OI lijkt op zichzelf dus weer een kortste buur heuristiek met ongeveer dezelfde oplossing. Wat ook opvalt is dat de kortste buur heuristiek en ook de OI strategie niet veel slechter zijn dan de optimale strategie. Dit geldt zelfs als grafen groter zijn.

4.3 Backup, diepte 1 tot n-stap-bootstrapping, on versus off strategie

We gaan een nieuwe techniek toepassen, in plaats van dat we de waardes van de Q tabel pas aanpassen nadat we de Hamiltonkring hebben doorlopen, updaten we de waardes al tussentijds. Deze techniek heet bootstrapping, bij bootstrapping heb je parameter k. Deze parameter k bepaalt na hoeveel stappen je de $Q^{\pi}(s_T, a_{s_T})$ tabel gaat updaten, om vervolgens verder de Hamiltonkring af te maken. Wanneer geldt $k = n$ (met n de grootte van de graaf) dan zien we het Monte Carlo algoritme terug. Wie een gegeneraliseerd algoritme wil hebben kan opnieuw het boek van Sutton en Barto raadplegen (16).

Opmerking 4.3. Merk op dat we n al gebruiken voor het aantal knopen/steden in de graaf, daarom zullen we hier k gebruiken voor het aantal stappen waarin we de kosten sommeren voordat we bootstrappen. In plaats van dat we spreken over n-stap-bootstrapping wat gebruikelijk is in de vakliteratuur spreken we dus over k-stap-bootstrapping voor $1 \leq k \leq n$.

Het algoritme voor het HRP is onderstaand te vinden.

Algoritme 6 k-bootstrapping, strategie, HRP

Require: $n \geq 3$, met K_n
Require: strategie
initialization of $Q^\pi(s_T, a_{s_T})$ tabel. $s_T \in \mathcal{S}_T, \forall a_{s_T} \in \mathcal{A}_{s_T}$, $T = 0, ..., n-1$
$s_T \in \mathcal{S}_T, \forall a_{s_T} \in \mathcal{A}_T : Q^\pi(s_T, a_{s_T}) \leftarrow$ gebasseerd op strategie, $T = 0, ..., n-1$
$s_T \in \mathcal{S}_T, \forall a_{s_T} \in \mathcal{A}_T : N(s_T, a_{s_T}) \leftarrow 0$ afhankelijk van strategie, $T = 0, ..., n-1$
$aantalIteraties \leftarrow$ gebasseerd op experiment
$IT \leftarrow 0$
while aantalIteraties >= IT **do**
$s_0 \leftarrow$ willekeurige knoop
Genereer een keten van lengte k (geen kring) gebaseerd op strategie π
$\pi : a_{s_0}, c(s_0, a_{s_0}), ..., s_k, a_{s_k}, c(s_k, a_{s_k})$
$IT \leftarrow IT + 1$
$u \leftarrow 0$
while $(u + k) < n$ **do**
$u \leftarrow u + 1$
$G \leftarrow \sum_{i=u}^{u+k-1} c(s_i, a_{s_i})$
$G \leftarrow G + Q^\pi(s_{u+k}, a_{s_{u+k}})$
$Q^\pi(s_u, a_{s_u}) \leftarrow Q^\pi(s_u, a_{s_u}) + \alpha \cdot (G - Q^\pi(s_u, a_{s_u}))$
$N(s_u, a_{s_u}) \leftarrow N(s_u, a_{s_u}) + 1$ (afhankelijk van de strategie)
Genereer de keten/hamiltonkring verder gebaseerd op strategie π
π: ..... s_{u+k}, $a_{s_{u+k}}$, $c(s_{u+k}, a_{s_{u+k}})$.
end while
if $k > 1$ **then**
Genereer de keten verder tot een Hamiltonkring
gebaseerd op strategie π
π: ..... s_{u+k+1}, $a_{s_{u+k+1}}$, $c(s_{u+k+1}, a_{s_{u+k+1}}), ..., s_n$, a_{s_n}, $c(s_n, a_{s_n})$
end if
$G \leftarrow 0$
for each i = n - 1, ..., u + k + 1 **do**
$G \leftarrow G + c(s_i, a_{s_i})$
$Q^\pi(s_i, a_{s_i}) \leftarrow Q^\pi(s_i, a_{s_i}) + \alpha \cdot (G + \gamma \cdot Q^\pi(s_{i+1}, a_{s_{i+1}}) - Q^\pi(s_i, a_{s_i}))$
$N(s_i, a_{s_i}) \leftarrow N(s_i, a_{s_i}) + 1$ (afhankelijk van de strategie)
end for
end while

Bovenstaand algoritme is een on-strategie algoritme, er bestaat ook een off-strategie algoritme. Het enige verschil is de formule waar de Q waardes geüpdate worden, bij de on-strategie is de update als volgt:

$$Q^{\pi}(s_i, a_{s_i}) \leftarrow Q^{\pi}(s_i, a_{s_i}) + \alpha \cdot \\ ((\sum_{i=T}^{min\{T+k-1,n\}} c(s_i, a_{s_i})) + \gamma \cdot Q^{\pi}(s_{i+1}, a_{s_{i+1}}) - Q^{\pi}(s_i, a_{s_i})). \tag{4.3.1}$$

Bij een off-strategie is de formule:

$$Q^{\pi}(s_i, a_{s_i}) \leftarrow Q^{\pi}(s_i, a_{s_i}) + \alpha \cdot \\ ((\sum_{i=T}^{min\{T+k-1,n\}} c(s_i, a_{s_i})) + \gamma \cdot \min_{b_{s_{i+1}} \in \mathcal{A}_{s_{i+1}}} [Q^{\pi}(s_{i+1}, b_{s_{i+1}})] - Q^{\pi}(s_i, a_{s_i})). \tag{4.3.2}$$

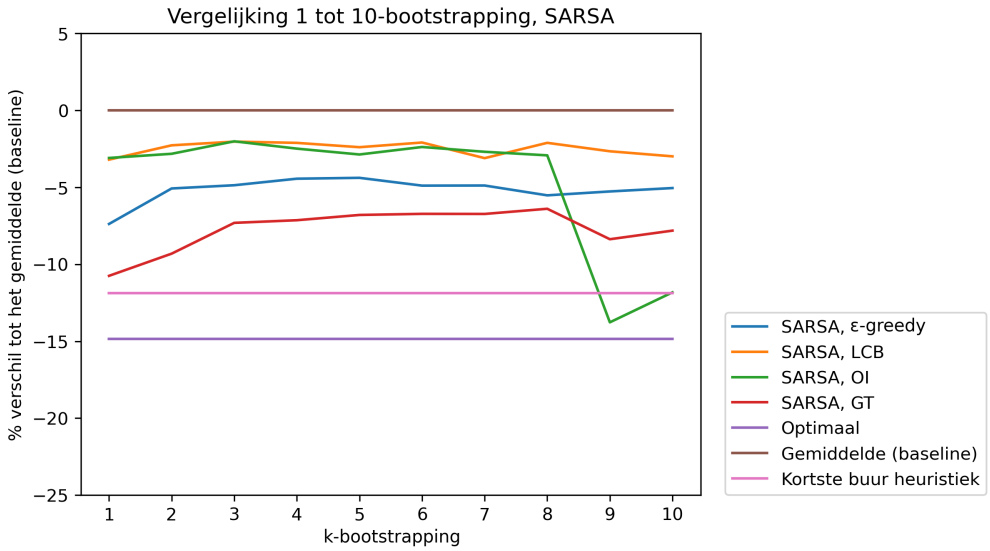
We voeren twee experimenten uit; één met behulp van on-strategie en één met behulp van off-strategie, op een graaf met 10 knopen. We voeren voor beide de experimenten uit voor 3000 iteraties. Voor elke k-bootstrapping en elke strategie nemen we één parameter c of ϵ en één parameter α. Om de parameters te kiezen draaien we het programma voor een strategie en de k-stap-bootstrapping voor alle parameters. We selecteren vervolgens de parameters die over de laatste 10 iteraties de laagste waarden terug geven. We selecteren daarbij uit de volgende parameters: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$, $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. Daarnaast moeten we de gewichten van de graaf nog initialiseren, dat doen we met behulp van een uniformele verdeling ($\mathcal{U}_{[76,124]}$), we nemen voor iedere tak een gewicht welke we willekeurig trekken uit de uniforme verdeling.

Opmerking 4.4. We hebben in Vergelijkingen 4.3.1 en 4.3.2 een discount factor toegevoegd, met het symbol γ op het domein $\gamma \in [0, 1]$. Deze parameter bepaald hoe belangrijk het is om toekomstige kosten mee te laten wegen, in ons geval is dat hoe belangrijk het is om nog naar de toekomstige pad kosten te kijken. We hebben experimenten uitgevoerd met $\gamma \in \{0.1, 0.2, 0.4, 0.6, 0.8\}$ en hebben vervolgens $\gamma = 0.2$ gekozen voor Vergelijking 4.3.2. Voor Vergelijking 4.3.1 hebben we $\gamma = 1.0$ gekozen om meer verschillen te onderzoeken . Deze parameter kan in de toekomst altijd bepaald worden per algoritme. Merk voor de rest op dat als we $\gamma = 0$ kiezen we de Q tabel laten leren met de kosten van elke tak en vervolgens op een oplossing komen van kortste buur heuristiek (indien we 1-stap-bootstrapping doen). De reden waarom dit zo is, is dat bij $\gamma = 0$ we de volgende vergelijking krijgen (voor 1 stap bootstrapping).

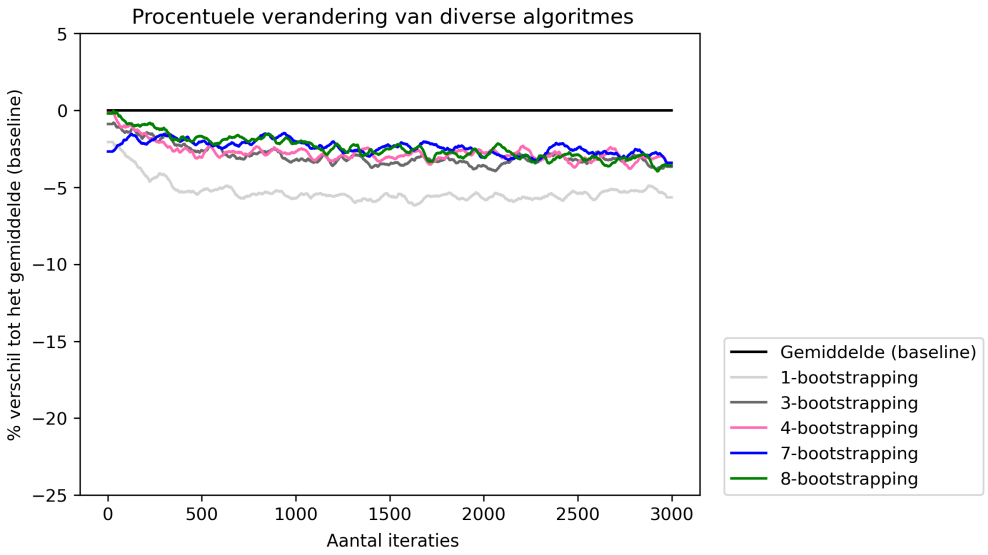
$$Q^{\pi}(s_i, a_{s_i}) \leftarrow Q^{\pi}(s_i, a_{s_i}) + \alpha \cdot (c(s_i, a_{s_i})) - Q^{\pi}(s_i, a_{s_i})). \tag{4.3.3}$$

We weten dat $Q^{\pi}(s_i, a_{s_i}) = 0$ geïnitialiseerd wordt en $c(s_i, a_{s_i}) > 0$. Definieer $B := c(s_i, a_{s_i}) - Q^{\pi}(s_i, a_{s_i})$. Dan geldt dat $B > 0$ zolang $Q^{\pi}(s_i, a_{s_i}) < c(s_i, a_{s_i}$ en dan blijft $Q^{\pi}(s_i, a_{s_i})$ strikt stijgend tot dat $Q^{\pi}(s_i, a_{s_i}) \geq c(s_i, a_{s_i}$. Dan is $Q^{\pi}(s_i, a_{s_i})$ weer dalend. Dit zorgt ervoor dat er geldt $Q^{\pi}(s_i, a_{s_i}) \approx c(s_i, a_{s_i})$ en daarmee de Q tabel de gewichten van de takken krijgt.

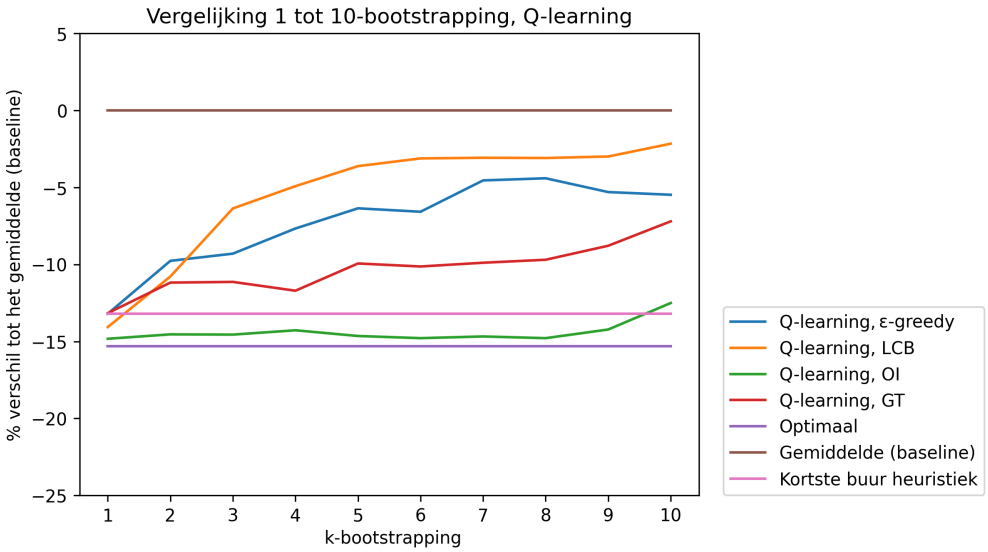
We zijn geïnteresseerd of er bepaalde waarden van k zijn, waarvoor k-stap-bootstrapping beter werkt. Ook zijn we geïnteresseerd of er bepaalde waarden van k zijn, waarvoor $k-$stap-bootstrapping nog niet uitgeleerd is (het loont dan om nog door te leren). Als laatste willen we het verschil tussen on-strategie versus off-strategie bekijken.



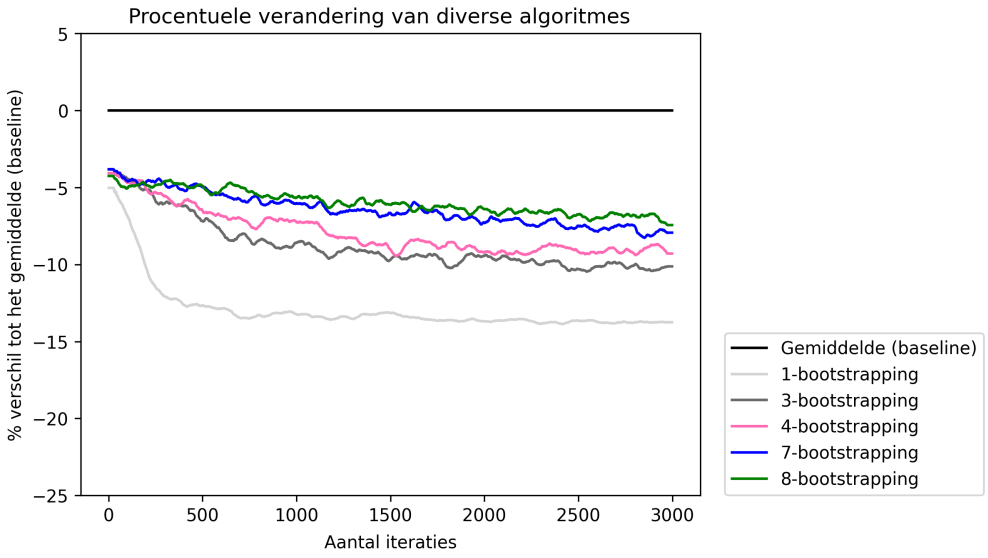
Figuur 8: ***k*-stap-bootstrapping met on-strategie (SARSA))** algoritme met de strategieen: ϵ`-greedy, LCB, GT, OI`, daarnaast de korstebuur heuristiek en de optimale oplossing voor een graaf van 10 knopen. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). Omdat de algoritmes over tijd leren, wordt daar het gemiddelde genomen over de laatste 10 iteraties. De parameters die voor de on-strategie worden gekozen zijn als volgt: voor elke strategie en elke k-stap-bootstrapping van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. De grafen hebben gewichten van een uniformele verdeling. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de laatste 10 iteraties gretig worden doorlopen, startend in toestand 1 (gegeven de beste parameters).



Figuur 9: ***k*-stap-bootstrapping met off-strategie (SARSA))** algoritme met de gemiddeldes over de strategieen: `ϵ-greedy, LCB, GT, OI`, voor een graaf van 10 knopen. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). We zien een leertraject over 3000 iteraties. De parameters die voor de off-strategie worden gekozen zijn als volgt, voor elke strategie en elke $1, 3, 4, 7, 8$-bootstrapping van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke $1, 3, 4, 7, 8$-stap-bootstrapping van de graaf 10 grafen, waarbij de grafen gewichten hebben van een uniformele verdeling. De smoothing window $= 51$. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de zoveelste iteratie gretig wordt doorlopen, startend in toestand 1 (gegeven de parameters).



Figuur 10: **k-stap-bootstrapping met on-strategie (Q-learning))** algoritme met de strategieen: ϵ-`greedy, LCB, GT, OI`, daarnaast de korstebuur heuristiek en de optimale oplossing voor een graaf van 10 knopen. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). Omdat de on-strategie algoritmes over tijd leren, wordt daar het gemiddelde genomen over de laatste 10 iteraties. De parameters die voor de on-strategie worden gekozen zijn als volgt: voor elke strategie en elke k-bootstrapping van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke k-stap-bootstrapping van de graaf 10 grafen, waarbij de grafen gewichten hebben van een uniformele verdeling. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de laatste 10 iteraties gretig worden doorlopen, startend in toestand 1 (gegeven de beste parameters).



Figuur 11: **k-stap-bootstrapping met on-strategie (Q-learning))** algoritme met de gemiddeldes over de strategieen: ϵ-`greedy, LCB, GT, OI`, voor een graaf van 10 knopen. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). We zien een leertraject over 3000 iteraties. De parameters die voor de off-strategie worden gekozen zijn als volgt: voor elke strategie en elke $1, 3, 4, 7, 8$-stap-bootstrapping van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke $1, 3, 4, 7, 8$-stap-bootstrapping van de graaf 10 grafen, waarbij de graaf gewichten hebben van een uniformele verdeling. De smoothing window $= 51$. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de zoveelste iteratie gretig wordt doorlopen, startend in toestand 1 (gegeven de parameters).

Resultaten
Laten we eerst kijken naar de on-strategie versus off-strategie. We zien dat de off-strategie beter werkt dan de on-strategie. De off-strategie geeft bij lage k-stap-bootstrapping zeer goede eindoplossingen aan het einde van het programma. Bij 1-stap-bootstrapping voor de off-strategie liggen ze zelfs net onder het niveau van de kortste buur heuristiek. De reden waarom hij net onder de kortste buur heuristiek ligt is waarschijnlijk het volgende: bij een discount factor van 0.0 ($\gamma = 0$) zijn we de getallen aan het initialiseren van de kortste buur heuristiek. Als we de γ op 0.2 zetten, nemen we voor de toekomstige kosten 20 procent mee. Merk op dat de $\gamma = 0.2$ voor reinforcement learning erg laag is. We leggen veel nadruk op de kosten die we direct krijgen, een kleine γ is voor reinforcement learning vrij uitzonderlijk. We zijn in feite bijna een kortste buur heuristiek aan het uitvoeren en dat lijkt een goede oplossing. Het loont dus om de kortste buur heuristiek te nemen met daarbij een klein beetje toekomstige kosten. Als we vervolgens de k groter maken voor k-stap-bootstrapping wordt er verhoudingsgewijs meer geüpdate met de kosten van een keten waardoor er minder kortste buur heuristiek wordt gebruikt, dit leidt tot slechtere resultaten. We zien dat over het algemeen geldt dat hoe hoger de k is voor k-stap-bootstrapping hoe lager de eindoplossing is en dat k-stap-bootstrapping minder snel leert.

Voor de rest maakt het niet veel uit welke k je gebruikt voor k-stap-bootstrapping bij de on-strategie (behalve de OI strategie). De strategieën geven voor verschillende k bijna allemaal dezelfde waardes, echter geeft een lage k wel vaker een lagere uitkomst en leert de lage k sneller. De reden hiervoor is dat bij de off-strategie altijd met de waardes van de strategie geüpdate wordt. Variëren in de k waarde geeft niet veel verandering in hoe er geüpdate wordt, terwijl dat bij de off-strategie wel uitmaakt. De redenen waarom de OI strategie voor grote k bij k-stap-bootstrapping naar lagere (betere) eindoplossingen gaat bij on-strategie moet nog verder onderzocht worden. Mogelijk ontstaat dit doordat er verhoudingsgewijs meer geüpdate wordt met de eindoplossing die de waarde 0 heeft.

4.4 Variatie in de gewichten van de grafen

We gaan in dit onderdeel een graaf bekijken die bestaat uit 50 steden met de Q-learning techniek (met 1-stap-bootstrapping). We zijn geïnteresseerd of dat het werkt wanneer het aantal aantal steden groot is. Tevens bekijken we of de structuur van de gewichten van de graaf invloed hebben op de eindoplossing. Daarvoor gaan we onze functie w variëren, we hebben in de vorige experimenten voor w de uniforme verdeling genomen. We gaan nu nog naar de volgende verdelingen kijken:

- Uniforme verdeling:

$$f(x) = \frac{1}{b-a} \quad 0 < a \leq x \leq b < \infty \tag{4.4.1}$$

  Hierbij zijn b en a parameters die de grenzen van de verdeling aangeven.

- Lognormaal verdeling, met als kansdichtheid:

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\cdot(\frac{ln(x)-\mu}{2})^2} \quad 0 < \sigma, x < \infty, -\infty < \sigma < \infty \tag{4.4.2}$$

  σ en μ zijn parameters om de verwachting en variantie te bepalen.

- Gamma verdeling, met als kansdichtheid:

$$f(x) = \frac{1}{\theta^k \cdot \Gamma(k)} x^{k-1} \cdot e^{-x/\theta} \quad 0 < k, \theta, x < \infty \tag{4.4.3}$$

  θ en k zijn parameters om de verwachting en variantie te bepalen.

  Daarnaast is $\Gamma(k)$ gegeven door:

$$\Gamma(k) = \int_0^\infty t^{k-1} e^{-k} dt$$

- Euclidisch-uniform
  In deze vorm worden de knopen random in een vierkant vak geplaatst. Daarna worden de afstanden tussen de punten gebruikt waardoor de euclidische afstand geldt, de grootte van het vierkant wordt met behulp van een algoritme bepaald. We willen dat de gewichten gemiddeld gelijk zijn aan de gewichten van de Gamma, Uniforme, Lognormaal verdeling. Dit doen we als volgt. We beginnen met 50 keer een klein vierkant van bepaalde grootte en 50 keer een

groot vierkant, We definiëren eerst het volgende:

A := *lengte zijde kleine vierkant.*

B := *lengte zijde grote vierkant.*

X := *het gemiddelde van de gewichten van de takken van het kleine vierkant.*

Y := *het gemiddelde van de gewichten van de takken van het grote vierkant.*

Z := *de verwachting van de gewichten van de uniforme verdeling*

Vervolgens plaatsen we de knopen in de vierkanten om zo de verwachtingen van de gewichten van beide vierkanten te schatten. Indien geldt $\frac{X+Y}{2} > Z$. *Dan wordt* B *geupdate als volgt* $B = \frac{A+B}{2}$, *in het andere geval wordt* A *geupdate als volgt:* $A = \frac{A+B}{2}$. *Dit blijven we herhalen (*A, B *updaten en knopen in een vierkant plaatsen om de verwachtingen van* X *en* Y *te bepalen) totdat de verwachtingen van het kleine vierkant en het grote vierkant ongeveer gelijk zijn aan de verwachting van* Z. *Zie code* graaf.py *voor de implementatie.*

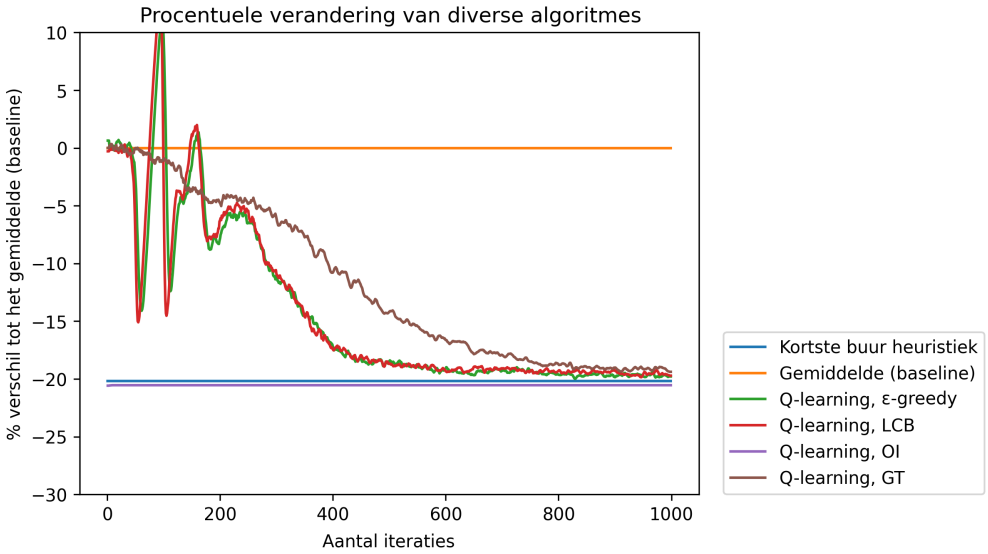
Om een zo eerlijk mogelijke vergelijking te maken tussen de verschillende verdelingen willen we dat de verwachtingen van de verdelingen gelijk zijn. Daarnaast willen we proberen om de variantie ook gelijk te hebben. Echter is dit voor de euclidische uniforme verdeling niet gelukt.

Om de parameters te bepalen hebben we de formules van de verwachtingen en de varianties van verschillende verdelingen op internet gevonden. Voor de referenties: Uniforme (17), Lognormaal (18) en Gamma : (19). We nemen een verwachting van 100 en een variantie van 192 (dit is een gevolg van de keuze van de gekozen parameters voor de uniforme verdeling $a = 76$ en $b = 124$). We hebben deze verwachting en variantie gelijk genomen voor alle verdelingen zodat hier geen verschil in kan ontstaan, we zijn bij dit onderdeel alleen geïnteresseerd of het HRP andere opgelossingen geeft voor verschillende verdelingen, de parameters worden dan als volgt:

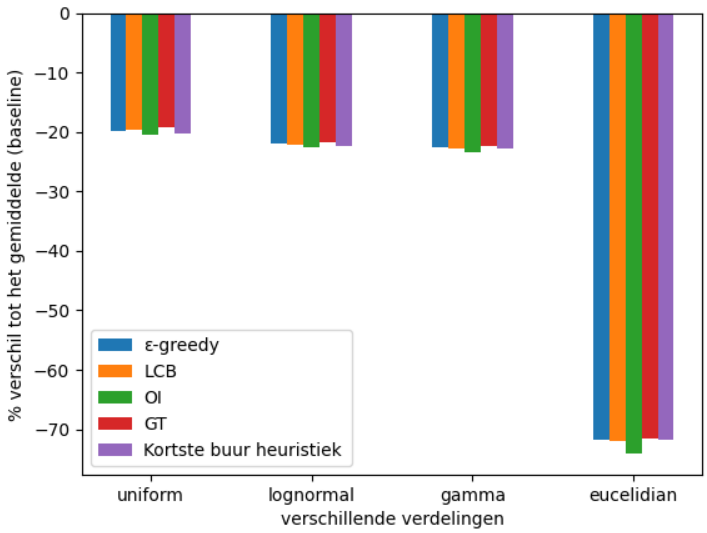
Verdeling	Parameters	Verwachting	Variantie
Uniform	$a = 76$ $b = 124$	$\frac{b-a}{2}$	$\frac{(b-a)^2}{12}$
Lognormaal	$\mu = 4.59566$ $\sigma = 0.137906$	$e^{\mu+\frac{\sigma^2}{2}}$	$(e^{\sigma^2} - 1) \cdot e^{2\cdot\mu+\sigma^2}$
Gamma	$k = \frac{625}{12}$ $\theta = \frac{48}{25}$	$k \cdot \theta$	$k \cdot \theta^2$

Tabel 3: De geïnitialiseerde parameters voor de Uniforme, Lognormaal en Gamma verdeling, waarvoor de verwachting gelijk is aan 100 en de variantie gelijk is aan 192.

We voeren het experiment uit met Q-learning over 1000 iteraties met 10 verschillende grafen, de graaf heeft 50 knopen. We voeren de experimenten weer uit op verschillende parameters van ϵ, c en α en selecteren de combinatie van de parameters die het beste eindresultaat geven. Omdat de optimale oplossing te veel computerkracht vereist voor een graaf van 50 knopen zullen we de optimale oplossing weglaten in deze resultaten. We weten uit de vorige experimenten dat de kortste buur heuristiek dicht bij de optimale oplossing zat. Dus dat is nog steeds een goede benadering.



Figuur 12: **Q-learning** 1-bootstrapping algoritme met de strategieën: ϵ`-greedy, LCB, GT, OI` voor grootte van de graaf $n = 50$. De parameters die voor Q-learning worden gekozen zijn als volgt: voor elke strategie worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke strategie van de graaf 10 grafen voor 1000 iteraties, waarbij de grafen gewichten hebben van de uniformele verdeling. De smoothing window = 5. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de zoveelste iteratie gretig wordt doorlopen, startend in toestand 1 (gegeven de parameters).



Figuur 13: **1-bootstrapping met off-strategie (Q-learning))** algoritme met de strategieën: ϵ`-greedy, LCB, GT, OI`, daarnaast de korstebuur heuristiek voor een graaf van 50 knopen, waarbij we verschillende initialisaties van de grafen zien voor de gewichten, namelijk: Uniform, Lognormaal, Gamma, euclidisch. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). Omdat de off-strategie algoritmes over tijd leren, wordt daar het gemiddelde genomen over de laatste 10 iteraties, waarbij voor 1000 iteraties geleerd wordt. De parameters die voor de off-strategie worden gekozen zijn als volgt: voor elke strategie en elke type verdeling van de gewichten en van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke verdeling van de graaf 10 grafen (we nemen het gemiddelde van de 10 grafen), waarbij de grafen gewichten hebben van de uniformele verdeling. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de laatste 10 iteraties gretig worden doorlopen, startend in toestand 1 (gegeven de beste parameters).

Resultaten

We zien in Figuur 13 dat er geen significante verschillen optreden bij de uniforme, lognormal en gamma verdeling, voor verschillende strategieën. De verschillen tegenover de baseline zijn dus allemaal redelijk gelijk aan het einde van het leer-

traject na 1000 iteraties. Het maakt dus niet veel uit hoe je de gewichten van de tak initialiseert gegeven dat de gewichten van de takken dezelfde verwachtingen en varianties hebben.

We zien bij de euclidische uniforme verdeling, dat de eindoplossing veel lager ligt in vergelijking met de andere verdelingen. De reden hiervoor is dat de variantie van de euclidische uniforme verdeling veel groter is. We konden deze variantie niet lager krijgen omdat we maar 1 parameter hadden om te bepalen bij de euclidische uniforme verdeling (namelijk de lengte van één zijde van het vierkant. We hebben besloten om de lengte van één zijde zo te bepalen zodat de verwachtingen van de verdelingen gelijk zijn). Om te laten zien dat variantie uitmaakt hebben we nog een extra figuur gemaakt met een uniforme verdeling waarbij we de variantie aanpassen, zie de appendix voor Figuur 16 waar dit bevestigd wordt. We zien inderdaad dat naarmate de variantie groter wordt de eindoplossingen grotere verschillen krijgen ten opzichte van de baseline.

Dan bekijken we nog Figuur 12. Deze figuur laat de Q-learning zien over het aantal iteraties voor de uniforme verdeling, voor een graaf met 50 knopen. We zien dat OI onder de korste buur heuristiek komt. Voor de rest zien we dat de strategieën ϵ-greedy, LCB en GT dicht bij de korstebuur heuristiek komen. Daarnaast leren ze voor de grootte van de graaf erg snel, zeker als je dat vergelijkt met de Monte Carlo techniek die we zagen in Figuur 7. Bij Q-learning leert deze sneller en heeft de eindoplossing een betere waarde dan met Monte Carlo.

In Figuur 12 valt ons iets bijzonders op. Je ziet voor de LCB en ϵ-greedy dat de leercurves in het begin twee keer hard naar beneden en naar boven schieten. Wat kan hier de oorzaak van zijn? Een mogelijke oorzaak is dat na ongeveer 50 iteraties de Q tabel genoeg geleerd heeft. Er zullen dan bijna geen waardes meer op 0 staan in de Q tabel waardoor de leercurve hard naar beneden gaat. Vervolgens zal de Q tabel verder geüpdate worden waardoor sommige combinaties van acties en toestanden in de Q tabel nieuwe waardes krijgen (deze zijn hoger dan de vorige waardes), waardoor deze niet meer in verhouding staan ten opzichte van andere waardes in de Q tabel (welke lager zijn). Hierdoor worden er vervolgens verkeerde Hamiltonkringen gekozen waardoor de leercurves terug omhoog gaan, na nog een keer alles goed leren (50x) zakt de grafiek vervolgens weer. Waarna er herhaling optreedt en nog een keer terugveert. Bovenstaande is een hypothese en zou verder onderzocht moeten worden.

4.5 Variatie in grootte van de toestandsruimte.

We gaan in dit onderdeel de grootte van de toestandsruimte variëren. Daartoe gaan we twee verschillende methodes toepassen. In de vorige paragraaf, paragraaf 4.4, hebben we een Q tabel gebruikt die grootte had van $n \times n$, hierbij werd telkens de laatst bezochte stad met de daarbij behorende actie onthouden. We gaan bij de eerste variant het aantal laatst bezochte steden groter maken zodat de waardes in de tabel hopelijk beter worden, hiertoe bekijken we in plaats van alleen de laatst bezochte stad, de laatst bezochte k steden. We hopen zo dat de strategie betere acties gaat kiezen omdat de Q tabel meer informatie bevat. Er zijn echter twee nadelen aan verbonden, één nadeel is dat de Q tabel groter wordt en we daardoor problemen kunnen krijgen met de ruimte complexiteit. Het andere nadeel is dat de toestandsruimte groter wordt, het duurt dan langer om alle waardes in de Q tabel naar een goede waarde te laten convergeren.

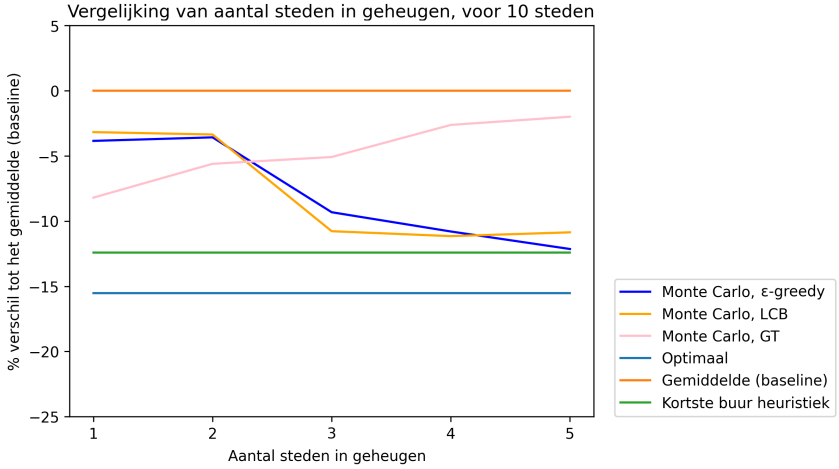
Bij de tweede variant gaan we weer het aantal steden in het geheugen variëren, echter gaan we nu ook een tijdsafhankelijke Q tabel maken. Dat betekent dat voor elk tijdstip T er een aparte Q tabel is. Het idee hierachter is dat we de informatie beter splitsen. Stel dat we namelijk maar 1 knoop bezocht hebben, dan zouden de toekomstige kosten nog gemiddeld 100 kunnen zijn, terwijl in stap 10 de toekomstige kosten gemiddeld 10 kunnen zijn (als we naar dezelfde waarde in de Q tabel kijken). Dan gaat de Q tabel gemiddeld naar een waarde van $(100 + 10)/2 = 55$. Maar bij een tijdsafhankelijke toestandsruimte, zou dit respectievelijk onthouden worden als 100 en 10. Het nadeel hiervan is wel dat we meer informatie moeten gaan onthouden en dit geheugen problemen kan geven, ook kan het langer duren voordat alle waardes in de Q tabel zijn geconvergeerd. Het kan dus langer duren om te leren.

We gebruiken in deze paragraaf de update regel van Monte Carlo (zie Algoritme: 5) en voeren het uit op de grootte van 10 steden met een uniforme verdeling met 3000 iteraties. We gebruiken de strategieën ϵ-greedy, Lower Confidence Bound en Gradient Temperature. De volgende parameters worden bekeken voor elke strategie: ϵ, c = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999] en α = [0.01, 0.1, 0.3, 0.5, 0.9], waarbij de parameters worden gebruikt die de laagste uitkomst geeft over de laatste 10 iteraties.

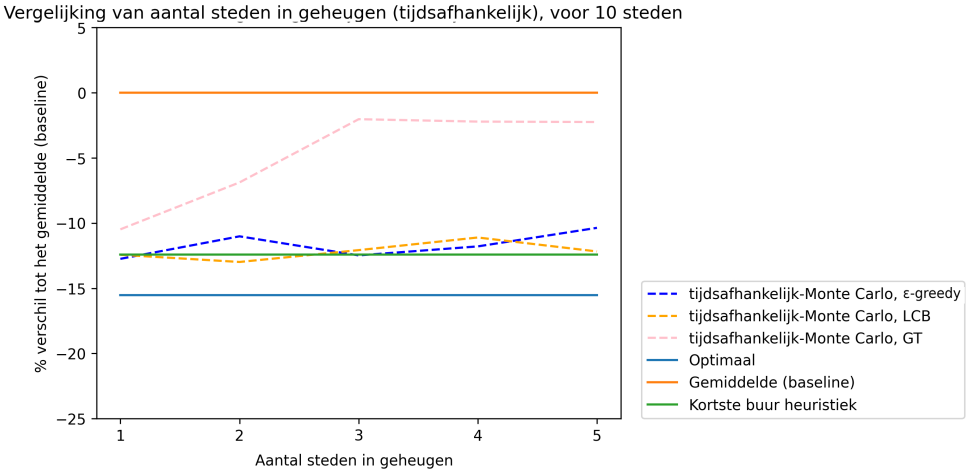
Opmerking 4.5. In bovenstaande is de strategie Optimistic Initialization weggelaten. De reden hiervoor is dat we de waarde in de Q tabel van te voren moeten inschatten. Als we elke keer de toestandsruimte groter maken moeten we de waardes in de Q tabel ook telkens anders schatten. Daardoor zijn we bang dat we twee variablen tegelijk aanpassen (namelijk hoe we de Q waarde initialiseren). Dit kan

een vertekend beeld geven in de uitkomsten en daarom hebben we de Optimistic Initialization weggelaten.

Hieronder zijn de experimenten met 3000 iteraties uitgevoerd waarbij "aantal steden in het geheugen" het aantal laatst bezochte steden van een keten in de Q tabel is. Eerst wordt de niet-tijdsafhankelijke figuur getoond en daarna de tijdsafhankelijke figuur:



Figuur 14: **Monte Carlo** algoritme met de strategieen: `ϵ-greedy`, `LCB`, `GT`, daarnaast de korstebuur heuristiek en de optimale oplossing. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). Omdat de Monte Carlo algoritmes gedurende de tijd leren, wordt het gemiddelde genomen over de laatste 10 iteraties. De parameters die voor Monte Carlo worden gekozen zijn als volgt: voor elke strategie en elke grootte van de toestandsruimte worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke grootte van de toestandsruimte 10 grafen. De grootte van de toestandsruimte wordt gevarieerd door het aantal laatst bezochte steden te variëren. Tot slot hebben de grafen gewichten van de uniformele verdeling. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de laatste 10 iteraties gretig worden doorlopen, startend in toestand 1 (gegeven de parameters).



Figuur 15: **Monte Carlo** algoritme met de strategieen: `ϵ-greedy`, `LCB`, `GT`, daarnaast de korstebuur heuristiek en de optimale oplossing. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). Omdat de Monte Carlo algoritmes gedurende de tijd leren, wordt het gemiddelde genomen over de laatste 10 iteraties. De parameters die voor Monte Carlo worden gekozen zijn als volgt, voor elke strategie en elke grootte van de toestandsruimte worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke grootte van de toestandsruimte 10 grafen. De grootte van de toestandsruimte wordt gevarieerd door het aantal laatst bezochte steden te variëren. De Q tabel wordt voor elke tijd T gemaakt, zodat de toestandsruimte nog groter is. Tot slot hebben de grafen gewichten van de uniformele verdeling. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de laatste 10 iteraties gretig worden doorlopen, startend in toestand 1 (gegeven de parameters).

Resultaten
In Figuur 14 zien we dat naarmate de toestandsruimte groter wordt de waardes dalen en richting de waarde van de heuristiek gaan, dat is voor de ϵ-greedy en de LCB in ieder geval zo. Voor de Gradient Temperature zien we dat de resultaten slechter worden naarmate de toestandsruimte groter wordt. We hebben hiervoor een verklaring, we zullen eerst de exploratie parameters noteren. Dit doen we omdat we die nodig hebben om te begrijpen waarom de verschillende strategieën andere uitkomsten krijgen.

	Aantal steden in het geheugen				
	1	**2**	**3**	**4**	**5**
epsilon-greedy	0.999	0.8	0.001	0.001	0.001
Lower Confidence Bound	0.6	0.15	0.15	0.3	0.001
Gradient Temperature	0.001	0.15	0.3	0.6	0.15

Tabel 4: De gekozen parameters voor c en ϵ voor verschillende strategieën. Voor de grafiek die niet tijdsafhankelijk is met grotere toestandsruimte, zie Figuur 14.

We zien dat naarmate het aantal steden in het geheugen toeneemt voor de LCB en ϵ-greedy er lage exploratie parameters zijn. Bij deze methodes wordt dan bijna alleen maar geëxploreerd. Dit is omdat de toestandsruimte heel groot is voor 5 steden in het geheugen. Daardoor worden er alleen acties geselecteerd die we nog niet eerder zijn tegen gekomen. (Er wordt dan namelijk telkens de 0 waarde geselecteerd, omdat zo de Q tabel geïnitialiseerd wordt). Anders doorlopen we niet de volledige toestandsruimte en blijven sommige combinaties van toestand en actie op waarde 0 staan (wat slecht is, want dan denken we dat deze combinatie van actie en toestand de optimale is). Er zijn voor 5 steden in het geheugen ongeveer $\frac{10!}{5!} = 30.240$ toestanden van lengte 5. De toestanden van enkelvoudige ketens van lengte 3,2, 1 en 0 niet eens meegenomen. Met 3000 iteraties en dat elke toestand ook nog (ongeveer) 9 acties heeft zorgt dit alles ervoor dat niet elke toestand en actie combinatie de juiste waarde bevat. Het volledig exploreren op de toestandsruimte (en dat gebeurt alleen als we volledig door de toestandsruimte gaan met lage exploratie parameters zodat de Q tabel met waardes 0 allemaal kunnen veranderen) is nodig om vervolgens goede acties te kiezen. De reden waarom Gradient Temperature niet werkt is meteen duidelijk, de kansverdeling is veel homogener voor lage exploratie parameters. Bij Gradient Temperature kunnen we minder goed forceren dat we met een grote kans (kans dicht bij 1) een actie selecteren

die we nog niet hebben geselecteerd terwijl dat bij LCB en ϵ-greedy wel kunnen forceren.

Dan gaan we kijken naar Figuur 15 (die tijdsafhankelijk is). We zien dat LCB en ϵ-greedy vanaf een laag aantal steden in het geheugen bijna dezelfde waarde hebben als de korste buur heuristiek. Het maakt vervolgens niet uit om het aantal steden in het geheugen verder te laten toenemen, blijkbaar zorgt het ervoor dat als je een tijdsafhankelijke toestandsruimte opbouwt er genoeg informatie is om goede beslissingen te nemen. Het feit dat de waarde niet lager wordt naarmate het aantal steden in het geheugen toeneemt kan naar onze mening twee oorzaken hebben. De eerste is net als bij de niet-tijdsafhankelijke toestandsruimte dat de toestandsruimte te groot wordt. Het groter maken van de toestandsruimte zorgt er voor dat de informatie in de Q tabel niet meer volledig geconvergeerd is (we hebben meer iteraties nodig om de Q tabel juist te vullen). Hierdoor worden er soms onjuiste beslissingen genomen als we gretig door de Q tabel lopen. Een tweede reden, maar minder waarschijnlijk naar onze mening, is dat door het tijdsafhankelijk maken van de toestandsruimte er veel extra informatie verkregen wordt, die indirect ook naar voren komt als we de toestandsruimte groter maken. Om de eerste reden verder te onderzoeken kan er in de toekomst gekeken worden naar de leercurves voor verschillende strategieën. Indien ze nog niet zijn uitgeleerd kan het aantal iteraties groter gemaakt worden om zo de Q tabel betere waardes te laten geven. We kunnen met dezelfde methode ook de tweede reden onderzoeken. Als de eindoplossingen niet verder gaan dalen bij toename van het aantal iteraties betekent dit dat er niet meer informatie wordt verkregen.

We zien dat het tijdsafhankelijk maken van de toestandsruimte meteen leidt tot een grote daling van de totale kosten voor een Hamiltonkring en dit geldt al voor een laag aantal steden in het geheugen. Het tijdsafhankelijk maken van de toestandsruimte heeft dus de voorkeur ten opzichte van het groter maken van het aantal steden in het geheugen.

5 Conclusie

We hebben gezien dat het Handelsreizigersprobleem op te lossen is met behulp van dynamisch programmeren. We hebben de eindige horizon en oneindige horizon geïmplementeerd en bekeken. We zien dat dynamisch programmeren een oplossing biedt voor niet al te grote grafen, echter leidt dit voor grote n tot geheugen en snelheids problemen. Op de korte termijn is niet te verwachten dat computers snel genoeg worden en genoeg geheugen krijgen om dit op te lossen.

Voor de reinforcement learning met de Monte Carlo update techniek zien we dat de strategieën naar redelijke oplossingen convergeren. De strategieën leren wel, maar zijn signifant slechter dan de korste buur heuristiek. De Optimistic Initialisation werkt bijna even goed als de kortste buur heuristiek en het verschil is altijd maar klein. We zien voor de rest dat Q-learning beter werkt dan SARSA en dat over het algemeen je beter 1-stap-bootstrapping kan gebruiken dan een k-stap-bootstrapping met een grotere k waarde, dit geldt in het bijzonder voor Q-learning. Q-learning leert sneller dan Monte Carlo en de strategieën convergeren naar goede oplossingen zelfs voor een grote waarde van n. De Optimistic Initialisation werkt zelfs beter dan de korste buur heuristiek bij Q-learning, ook al is het verschil klein. Voor de rest zien we dat het niet veel uitmaakt hoe de gewichten geïnitialiseerd zijn zolang ze maar dezelfde verwachting en variantie hebben.

We hebben als laatste de toestandsruimte groter gemaakt door het aantal steden in het geheugen groter te maken. We hebben daarbij gekeken naar twee verschillende modellen, de eerste waarbij de toestandsruimte ook tijdsafhankelijk is en de tweede waarbij de toestandsruimte niet tijdsafhankelijk is, voor beide modellen hebben we Monte Carlo gebruikt. We zien bij de niet-tijdsafhankelijke toestandsruimte dat het kan helpen om de toestandsruimte groter te maken, echter is dit wel afhankelijk van de strategie. Voor de tijdsafhankelijke toestandsruimte zien we dat de oplossing meteen beter wordt. Het vervolgens groter maken van de toestandsruimte voor de tijdsafhankelijke variant heeft verder geen effect. Beide modellen kunnen nog verder onderzocht worden om te kijken of het zin heeft om ze langer door te laten leren (ofwel om ze meer iteraties uit te laten voeren).

Voor verder onderzoek is het interessant om de tijdsafhankelijke toestandsruimte verder te onderzoeken, maar dan met behulp van Q-learning. Ook de Optimistic Initialisation kan dan gebruikt worden en kijken hoe deze werkt op de tijdsafhankelijke toestandsruimte. Daarnaast is het verder onderzoeken van de initialisatie van de Q tabel voor de Optimistic Initialisation iets wat de strategie beter kan maken. Een idee hierbij zou zijn om een neural netwerk te trainen zodat deze Q tabel beter geïnitialiseerd kan worden.

Reinforcement Learning heeft zeker nut om een goede oplossing te vinden voor het Handelsreizigersprobleem. Dit moet dan wel in combinatie met een goede strategie en updateregels gebeuren. Door nog verder onderzoek uit te voeren zoals beschreven in de alinea's hierboven zal reinforcement learning waarschijnlijk nog betere resultaten boeken. Het kan dan significant beter worden dan de bekende heuristieken en we hebben er vertrouwen in dat de optimale oplossing benaderd gaat worden.

Referenties

[1] M. v. d. B. L.C.M Kallenberg, F.M Spieksma, "Combinatoriek en optimalisering," p. 84, 2019.

[2] C. E. Miller, A. W. Tucker, and R. A. Zemlin, "Integer programming formulation of traveling salesman problems," *Journal of the ACM (JACM)*, vol. 7, no. 4, pp. 326–329, 1960.

[3] P. R. d O Costa, J. Rhuggenaath, Y. Zhang, and A. Akcay, "Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning," in *Asian Conference on Machine Learning*, pp. 465–480, PMLR, 2020.

[4] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," p. 321, 2018.

[5] T. A. D. Costa, "Solving the traveling salesman problem with reinforcement learning," 2021.

[6] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," p. 131, 2018.

[7] M. Jünger, G. Reinelt, and G. Rinaldi, "The traveling salesman problem," *Handbooks in operations research and management science*, vol. 7, pp. 225–330, 1995.

[8] A. H. Halim and I. Ismail, "Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem," *Archives of Computational Methods in Engineering*, vol. 26, no. 2, pp. 367–380, 2019.

[9] "Tweakers." https://tweakers.net/nieuws/173626/hpe-bouwt-supercomputer-in-finland-met-amd-hardware-en-552-petaflops-rekenkracht.html.

[10] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," p. 91, 2018.

[11] "Epsilon-greedy." https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/.

[12] S. Roberts, "The upper confidence bound (ucb) bandit algorithm," 2020.

[13] S. Lobel, O. Gottesman, C. Allen, A. Bagaria, and G. Konidaris, "Optimistic initialization for exploration in continuous control," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, 2022.

[14] H. Sun, L. Han, R. Yang, X. Ma, J. Guo, and B. Zhou, "Exploit reward shifting in value-based deep-rl: Optimistic curiosity-based exploration and conservative exploitation via linear reward shaping," *Advances in Neural Information Processing Systems*, vol. 35, 2022.

[15] "Tsp-solver." https://pypi.org/project/python-tsp/.

[16] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," p. 141, 2018.

[17] M. Taboga, "Uniform distribution." https://www.statlect.com/probability-distributions/uniform-distribution.

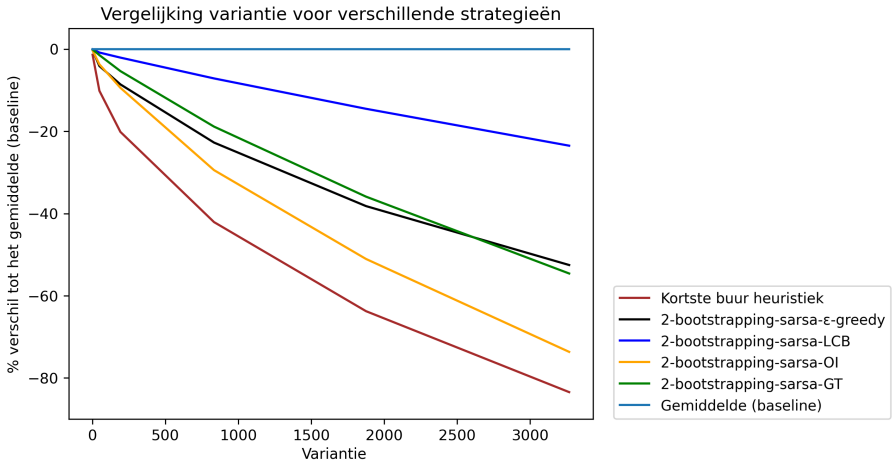
[18] M. Taboga, "log-normal distribution." https://www.statlect.com/probability-distributions/log-normal-distribution.

[19] M. Taboga, "Gamma distribution." https://www.statlect.com/probability-distributions/gamma-distribution.

6 Code

`https://github.com/woutermulders/HRP_reinforcement_learning`

7 Appendix



Figuur 16: **2-stap-bootstrapping met on-strategie (SARSA))** algoritme met de strategieën: ϵ-`greedy, LCB, GT, OI`, daarnaast de korstebuur heuristiek voor een graaf van 10 knopen, met verschillende initialisaties voor de gewichten van de takken om de grootte van de variantie te variëren. De algoritmes worden vergeleken met de totale som van de kosten van een gemiddelde Hamiltonkring (de baseline). Omdat de algoritmes over tijd leren, wordt daar het gemiddelde genomen over de laatste 10 iteraties. De parameters die voor on-strategie worden gekozen zijn als volgt, voor elke strategie en elke k-stap-bootstrapping van de graaf worden de volgende parameters bekeken: $c, \epsilon = [0.001, 0.15, 0.3, 0.6, 0.8, 0.999]$ en $\alpha = [0.01, 0.1, 0.3, 0.5, 0.9]$. De parameters worden zo geselecteerd dat deze de laagste uitkomst geven over de laatste 10 iteraties. We bekijken voor elke grootte van de variantie 2-bootstrapping van de graaf 10 grafen, waarbij de graaf gewichten hebben van een uniformele verdeling afhankelijk van de grootte van de variantie. De waardes die je ziet in de grafiek zijn uit de Q tabel, die op de laatste 10 iteraties gretig worden doorlopen, startend in toestand 1 (gegeven de beste parameters).