KU LEUVEN kulak

Groep Wetenschap & Technologie Kulak
Informatica
Gegevensstructuren en Algoritmen

Project Gegevensstructuren en Algoritmen: verslag

Academiejaar 2022 – 2023

Arthur Cremelie

Inhoudsopgave

Inleiding

In dit verslag bespreken we twee oplossingsmethodes voor het Traveling Salesman Problem (TSP). Eerst vermelden we in een korte samenvatting wat het probleem inhoudt en welke twee oplossingsmethodes gekozen werden en waarom. Als tweede zal de lezer te weten komen welke ontwerpbeslissingen er genomen zijn voor de twee oplossingsmethodes met hun motivatie. Als derde zullen we het hebben over de concretere implementatiedetails die een belangrijke impact kunnen hebben op de efficiëntie. Op het einde van dit verslag zullen we antwoorden zoeken op een aantal onderzoeksvragen door een aantal experimenten uit te voeren. Ook zullen we verschillende metaheuristieken met elkaar vergelijken.

1 Projectsamenvatting

TSP is een gekend probleem in het domein van de theoretische computerwetenschappen en heeft volgend doel. Gegeven een lijst met n steden en hun onderlinge afstanden, wat is het korst mogelijke pad dat iedere stad juist één keer bezoekt en eindigt in de stad waar het is vertrokken? Dit probleem is NP-hard, wat als gevolg heeft dat de exacte algoritmes relatief traag zijn. Daarom implementeren en vergelijken we twee 'niet-exacte' oplossingen die toch een relatief goede oplossing geven in snellere tijd. De 'niet-exacte' oplossingsmethodes die we gebruiken in dit project zijn Tabu Search (TS) en Ant Colony Optimization (ACO). Beide methodes staan ook bekend als metaheuristieken.

2 Ontwerpbeslissingen

2.1 Tabu Search

2.1.1 Algoritme

Het doel van TS is om aan de hand van local search een tour te vinden met een zo laag mogelijke kost C. Local search gaat hiervoor vertrekken vanuit een bepaalde initiële oplossing en zal daarna de buurt bekijken van die oplossing. Indien er een buur is waarbij de kost lager is dan de kost van de tot nu toe beste tour, zullen we de beste tour vervangen door de gevonden buur. De buren zien we hier als een 'move' tussen alle verschillende knopen. We bespreken de move die we in dit project gebruiken in 2.1.2. Daarna passen we opnieuw local search toe op de nieuwe gevonden oplossing. Deze stap herhalen we tot een bepaald stop criterium is bereikt. Dat stop criterium is in dit project een maximum aantal iteraties. Maar andere stopcriteria zijn ook mogelijk.

TS gebruikt een datastructuur, die we tabu list noemen, om te verhinderen dat local search vast raakt in een lokaal optimum. De tabu list bestaat uit 'moves' die hebben geleid tot een betere oplossing. Deze lijst vullen we telkens aan wanneer we een nieuwe beste oplossing vinden. TS mag deze 'move' dus niet meer maken in de toekomst. Deze tabu list heeft een vaste lengte, en hanteert het *first-in first-out* principe. Wanneer we een 'move' willen toevoegen, maar de tabu list vol is, wordt het eerste element niet meer als *tabu* aanzien en voegen we de nieuwe 'move' toe.

2.1.2 Move

In dit project kozen we voor 2-opt als 'move'. Dit omdat 2-opt in de literatuur het vaakst werd gekozen en een degelijke 'move' blijkt te zijn [1]. Het principe van 2-opt is dat twee knopen i en j van plaats wisselen en dat we alle knopen ertussen in omgekeerde volgorde doorlopen. Zo kunnen we onder andere twee kruisende bogen niet meer laten kruisen [2]. Een 2-opt 'move' zullen we in de tabu list bijhouden als een paar van twee knopen (i, j).

2.1.3 Initiële oplossing

TS heeft een initiële oplossing nodig om een verbetering uit te kunnen voeren. Een goede initiële oplossing vinden we bij het greedy algoritme nearest neighbour [1]. Dit algoritme zal starten met een random knoop i en zal nadien een nieuwe knoop j toevoegen die nog niet tot de tour behoort, zodat de kost c_{ij} van de nieuw toegevoegde boog (i, j) zo laag mogelijk is. Deze stap wordt herhaald totdat we een volledige tour hebben.

2.1.4 Tabu list

Een eerste belangrijke datastructuur die TS gebruikt is de tabu list. Wanneer een element aan de lijst wordt toegevoegd, en de lijst vol is, wordt het eerst toegevoegde element verwijderd om plaats te maken. Dit staat ook wel bekend als het FIFO-principe. Omdat een queue gebruikt maakt van het FIFO-principe, maken we daar gebruik van voor onze tabu list. Java heeft een klasse `ArrayBlockingQueue` die we gebruiken in de implementatie. Een probleem dat we bekomen bij het gebruik van een queue, is dat het zoeken of een element in die lijst zit lineair is. Dit verslechterd de efficiëntie van het algoritme, omdat we deze check telkens moeten doen bij het bekijken van alle buren. Daarom hebben we ervoor gekozen om parallel een binaire matrixstructuur bij te houden met volgende waarden:

$$\begin{cases} 1 & \text{als boog } (i, j) \text{ tabu is,} \\ 0 & \text{anders.} \end{cases}$$

Een belangrijke implicatie bij deze keuze is dat we steeds twee datastructuren moeten bijhouden, maar het opvragen of een boog (i, j) tabu is, is nu wel constant. Om de koppels van bogen die tabu zijn in de queue bij te houden, maken we gebruik van een zelfgemaakte klasse `Pair`. Het enige wat deze klasse doet is twee integers bijhouden.

2.1.5 Tour

Een tweede belangrijke datastructuur is de tour zelf. Voor TS wordt de tour bijgehouden in een Doubly Linked List. Deze keuze is bewust gemaakt omdat een 2-opt 'move' dan zeer efficiënt, namelijk in constante tijd, kan worden uitgevoerd [3]. Zo'n Doubly Linked List bestaat uit Nodes, die elk een getal en twee Nodes (pointers) als kinderen bevatten. Een belangrijke eigenschap van deze Doubly Linked List is dat de Nodes in de lijst geen onderscheid maken tussen welke Node de vorige is en welke Node de volgende is. We houden gewoon twee Nodes bij. De lijst kunnen we dan in de correcte volgorde doorlopen als we weten wat de vorige Node is. Dankzij deze Doubly Linked List moeten er altijd slechts vier pointers gewijzigd worden voor een 2-opt 'move', wat constant kan. Tevens hebben we ook de keuze gemaakt om de Doubly Linked List circulair te maken, i.e. de head verwijst naar de tail en omgekeerd. Dit maakt het programmeren eenvoudiger. In Java hebben we zelf de klasse `DoublyLinkedList` ontworpen met een inner class `Node`.

Bij het doorlopen van de buren is het belangrijk om de pointers van die buren steeds bij te houden. Als we telkens de knoop moeten opzoeken in de Doubly Linked List, zal dit ook de tijdscomplexiteit negatief beïnvloeden. Om dit correct te implementeren, moeten we 4 pointers bijhouden, zodat we de potentiële 2-opt swap correct kunnen uitvoeren.

2.2 Ant Colony Optimization

2.2.1 Algoritme

We kozen voor Ant Colony Optimization omdat het domein van bio-inspired computing intrigerend was. Daarnaast bestaan er veel verschillende versies van Ant Colony Optimization. In dit project kozen we voor Ant Colony System (ACS) besproken in [4].

Het principe gaat als volgt. Wanneer mieren in de echte wereld op zoek gaan naar voedsel, scheiden ze feromonen af. De feromonenwaarden worden bepaald volgens twee criteria: de kwaliteit en de hoeveelheid van het voedsel. Hoe beter de twee criteria zijn, hoe sterker de feromonenwaarden en hoe meer mieren dat pad zullen volgen. De feromonenwaarden zullen naar verloop van tijd ook verdampen. We vertalen dit principe naar TSP. Een aantal, zeg m, verschillende 'ants' maken een bepaalde tour, startend vanuit een random knoop i. Ze kiezen een volgende knoop j uit een lijst N, dat alle knopen die nog niet tot de tour behoren bevat, volgens formule 1.

$$j = \begin{cases} argmax_{l \in N}\{\tau_{il}[\eta_{il}]^{\beta}\} & \text{als } q \leq q_0, \\ argmax_{l \in N}\{\frac{[\tau_{il}]^{\alpha}[\eta_{il}]^{\beta}}{\sum_{r \in N}[\tau_{ir}]^{\alpha}[\eta_{ir}]^{\beta}}\} & \text{anders.} \end{cases} \tag{1}$$

Met $\eta_{ij} = \frac{1}{c_{ij}}$, α en β constanten. q is een uniform verdeeld willekeurig getal in $[0, 1]$ en $0 \leq q_0 \leq 1$ een parameter.

Daarna kiezen ze opnieuw een volgende knoop $j + 1$ op basis van formule 1. Dit herhalen we totdat alle knopen tot de tour behoren. Bij iedere boog die gekozen wordt zal de feromonenwaarde van die boog (i, j) verdampen volgens formule 2.

$$\tau_{ij} = (1 - \xi)\tau_{ij} + \xi\tau_0, \tag{2}$$

Met $0 < \xi < 1$ een parameter en τ_0 de initiële feromonenwaarde. τ_0 wordt verder in deze paragraaf besproken.

Uiteindelijk zullen enkel de feromonenwaarden van de bogen van de tour T^{bs} van de 'ant' met de tour die de laagste kost heeft gecreëerd, versterken. Die feromonenwaarden wijzigen we als volgt:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs}, \forall(i, j) \in T^{bs}, \tag{3}$$

met ρ de verdampingsgraad, $\Delta\tau_{ij}^{bs} = \frac{1}{C^{bs}}$ en C^{bs} de kost van T^{bs}.

Om de performantie van ACS te verbeteren, stellen we de initiële feromonenwaarden τ_0 van de bogen in op $1/(n\dot{C}_{nn})$, met C_{nn} de kost van de nearest neighbour oplossing [4].

2.2.2 Gegevensstructuren

De gegevensstructuren voor ACS zijn vrij eenvoudig. Als voorstelling van de tour maken we gebruik van een ArrayList, omdat er geen specifieke compilaties zijn qua snelheid. De feromonenwaarden worden bijgehouden in een matrix met doubles.

3 Analyse implementatie

3.1 Tabu Search

Het is geen goed idee om bij het onderzoeken van de buren telkens een nieuwe tour aan te maken, de nieuwe kost te berekenen en te vergelijken met de beste tour tot nu toe. Omdat we telkens bij het aanmaken van een nieuwe tour een lijst moeten kopiëren, wat lineair is. Ook het berekenen van de kost zou dan lineair zijn.

We bekijken een paar belangrijke aspecten van de implementatie die ervoor zorgen dat de efficiëntie van de gekozen datastructuren nut hebben. Een eerste aspect is het bijhouden van pointers van de te overlopen knopen i en j in de buurt. Indien we geen pointers bijhouden, moeten we telkens zoeken naar de positie van de desbetreffende knoop in de lijst, wat lineair zou zijn. Met pointers kunnen we de kinderen van de knopen in constante tijd aanpassen. Er moeten telkens vier pointers van vier knopen worden bijgehouden: knoop i, de knoop voor i, knoop j en de knoop na j. Deze zijn nodig om een swap uit te voeren en dus ook om een kostenreductie te berekenen. De pointers worden in de for-lussen aangepast. Een tweede aspect is het berekenen van de potentiële kostenreductie in de plaats van de 'move' daadwerkelijk uit te voeren en daarna te berekenen of die tour heeft geleid tot een lagere kost. Een kostenreductie berekenen aan de hand van de vier pointers kan in constante tijd. Wanneer we een betere kostenreductie vinden, houden we de pointers bij. De 'move' moet dan enkel uitgevoerd worden aan de hand van de bijgehouden pointers gevonden bij de beste kostenreductie. Als laatste aspect kan het voordelig zijn om de 2-opt swap constant te maken door slechts de vier pointers aan te hoeven passen. Dit kan dankzij het gebruik van de Doubly Linked List.

3.2 Ant Colony System

De implementatie van Ant Colony System bestaat vooral uit veel berekeningen gebruikmakend van de formules besproken in 2.2.1. De implementatie bestaat uit twee klassen: `AntColonySystem` en `Ant`. De eerste klasse zal de basisstructuur van het algoritme bevatten, waarin een aantal 'ants' aangemaakt worden aan de hand van de klasse `Ant`. Deze 'ants' zullen dan elk een tour maken aan de hand van formule 1. Omdat elke tour elke knoop (buiten de vertrek- en eindknoop) slechts precies één keer moet bezoeken, houden we een lijst bij die de knopen bevat die nog niet bezocht zijn. Telkens we een knoop toevoegen aan de tour, moeten we deze knoop verwijderen uit de lijst. Daarna zullen we in de klasse `AntColonySystem` de bogen van de beste tour van de beste 'ant' telkens updaten aan de hand van formule 3.

4 Experimenten

We bekijken eerst een aantal experimenten die we uitvoerden op TS. Daarna bekijken we een aantal experimenten met ACS. Ten slotte zullen we ook TS met ACS en andere heuristieken vergelijken. We voeren onze experimenten eerst uit op een experimentenset die bestaat uit de problemen opgelijst in tabel 2. Indien er geen specifieke parameters staan vermeld, werden de parameters uit tabel 1 gebruikt voor ACS.

4.1 Tabu Search

De experimenten worden per iteratie 20 keer uitgevoerd, om de fluctuaties veroorzaakt door de initiële oplossing weg te werken.

variabele	waarde
ants	10
α	1
β	2
ρ	0.1
ξ	0.1

Tabel 1: Standaardparameters voor Ant Colony System

Probleem	Optimum	TS	ACS
berlin52	7542	7860	8056
tsp225	3919	4083	4465
pcb442	50778	52697	58158
gr666	294358	312767	347037
u724	41910	44439	50972

Tabel 2: Problemen met hun optima, TS resultaat na 100 iteraties en een tabu list van lengte $n/2$ en ACS resultaat na 50 iteraties met 10 ants

4.1.1 Tabu list lengte

Een eerste onderzoeksvraag die we ons kunnen stellen is de volgende. Wat is het effect van de grootte van de tabu list op het resultaat van de bekomen tour? Het gebruik van dit geheugen kunnen we zo aanpassen zodat TS zich richt op ofwel diversificatie, ofwel intensificatie. Een grotere tabu list zal als gevolg hebben dat er verder wordt gezocht in de oplossingsruimte (diversificatie). Een kleinere tabu list zal ervoor zorgen dat er meer gekeken wordt in een bepaalde regio van de oplossingsruimte (intensificatie).

De resulaten van de (uitgebreide) experimentenset staan in tabel 3. Daarin zien we dat de lengte van de tabu list in onze implementatie slechts een klein effect heeft op het uiteindelijke resultaat. We zien dat voor kleine problemen een grotere tabu list beter werkt dan voor grotere problemen. Voor problemen van grootteorde 100 tot 700 zien we dat een tabu list met lengte $n/2$ het best werkt. Na 700 is het best om de tabu list nog te verkleinen. Al merken we dat het effect van de lengte van de tabu list niet veel uitmaakt, daar de verschillen miniem zijn. In een kritieke omgeving waarbij optimalisatie heel belangrijk is, kan het wel nuttig zijn.

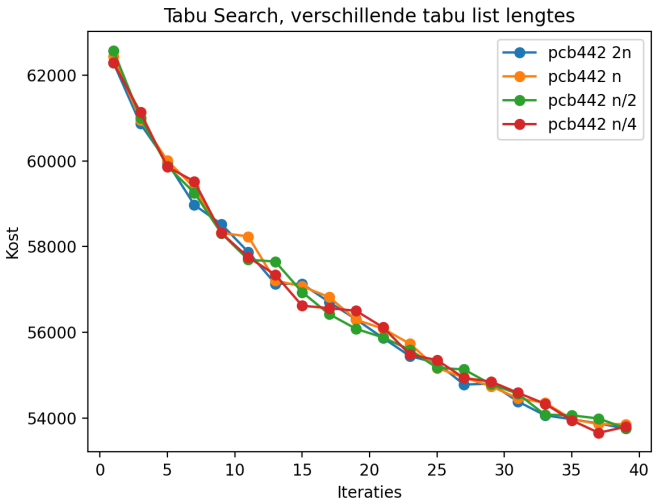
In figuur 1 plaatsen we deze test samen in een grafiek voor een probleem met 442 steden zoals `pcb442`. Hier is het duidelijk dat de verschillen van een heel kleine grootteorde zijn.

4.1.2 Dichtheid van punten

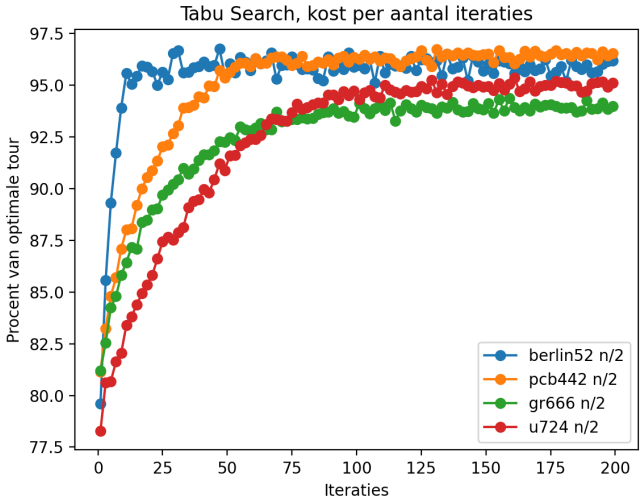
We zien in tabel 3 en figuur 2 dat het kleiner probleem `gr666` slechter presteert dan groter probleem `u724`. We kunnen ons de vraag stellen waarom dit zo is. Een eerste vaststelling die we kunnen maken, is het feit dat de punten in `gr666` verder van elkaar liggen dan `u724`. Dit kunnen we zien aan het feit dat de kost van `gr666` veel hoger ligt dan de kost van `u724`. Een andere reden zou kunnen zijn dat `gr666` een *GEO*-probleem is. Het kan zijn dat in de berekeningen van de coördinaten een fout zit die ervoor

Probleem	Optimum	TS_{2n}	TS_n	$TS_{3n/4}$	$TS_{n/2}$	$TS_{n/4}$	$TS_{n/8}$	$TS_{n/16}$	$TS_{n/32}$	rel. fout
berlin52	7542	**7647**	7864	7844	7888	7893	7863	7830	7873	1.39%
pr107	44303	44545	44556	44591	**44407**	44608	44568	44618	44619	0.23%
tsp225	3919	4101	4079	4080	**4049**	4087	4079	4096	4103	3.32%
pcb442	50778	52292	52454	52451	**52271**	52285	52413	52589	52658	2.94%
gr666	294358	312304	312404	312271	**312077**	312601	312820	313896	312919	6.01%
u724	41910	44228	44096	44177	44049	**43925**	44047	44218	44255	4.81%

Tabel 3: Problemen met hun optima, TS_x staat voor het resultaat van TS na 1000 iteraties en een tabu list van lengte x. We bekijken een tabu list lengte van $2n, n, 3n/4, n/2, n/4, n/8, n/16$ en $n/32$. De relatieve fout wordt berekend met het resultaat in het vet t.o.v. het optimum.



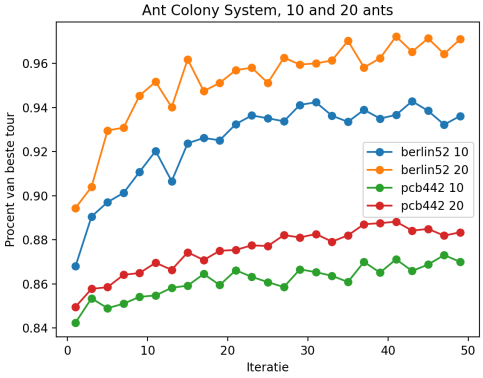
Figuur 1: Grafiek die het effect van verschillende lengtes van tabu list toont na 40 iteraties voor `pcb442`



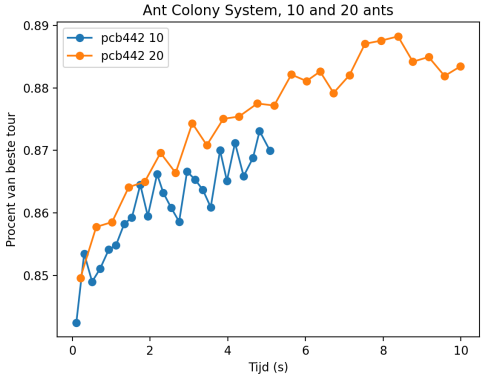
Figuur 2: Tabu list lengte $n/2$, standaardparameters

Probleem	Optimum	ACS_{10}	$t_{ACS_{10}}$ (s)	ACS_{20}	$t_{ACS_{20}}$ (s)	Verbetering kost	Δt
berlin52	7542	8056	0.0700	7767	0.1407	3.59%	0.0707
tsp225	3919	4465	1.2038	4423	1.5042	0.94%	0.3004
pcb442	50778	58366	5.0822	57477	9.9827	1.52%	4.9005
gr666	294358	347037	10.4230	342004	13.2348	1.45%	2.8118
u724	41910	50972	12.4921	49971	15.6924	1.96%	3.2003

Tabel 4: Problemen met hun optima, ACS met 10 ants, de tijd in seconden voor het uitvoeren van ACS met 10 ants, ACS met 20 ants en de tijd in seconden voor het uitvoeren van ACS met 20 ants. Deze resultaten zijn bekomen na 50 iteraties.



(a) `berlin52` en `pcb442` met 10 en 20 ants, volgens iteraties



(b) `berlin52` en `pcb442` met 10 en 20 ants, volgens tijd

Figuur 3: Ant Colony System toegepast op `berlin52` en `pcb442`.

zorgt dat de uiteindelijke kost fouten bevat. We kunnen de eerste veronderstelling ook verifiëren door een aantal problemen te nemen met gelijkaardig aantal steden en verschillen in hun optimale kost.

4.2 Ant Colony System

ACS heeft veel meer parameters die we kunnen wijzigen. Eerst bekijken we wat het aantal ants als effect heeft. Daarna veranderen we de waarden van ξ en ρ. Uiteindelijk voeren we een aantal experimenten uit met verschillende waarden voor β. De experimenten voor ACS worden voor elke iteratie 10 keer uitgevoerd om fluctuaties veroorzaakt door de willekeurige startpositie van een ant weg te filteren.

4.2.1 Aantal ants

We kunnen ons de vraag stellen of het gebruik van meerdere ants leidt tot een betere oplossing in een aanvaardbare tijd. We voeren een aantal experimenten uit op de problemen van onze problemenset en vatten de resultaten samen in tabel 4. Daaruit kunnen we afleiden dat de resultaten telkens beter zijn bij het gebruik van 20 ants. We moeten nu de afweging maken of het gebruik van meerdere ants de uitvoeringstijd niet te negatief beïnvloedt. Voor de experimentenset komen we uit op een gemiddelde verbetering van 1.9% in met een gemiddelde Δt van 2.26 seconden.

4.2.2 Feromonenwaarden updaten

We kunnen ons ook afvragen wat de invloed is op andere verdampingsratio's voor de update van de feromonenwaarden. Enerzijds gebeurt dit lokaal door iedere ant aan de hand van ξ. Anderzijds zullen de feromonenwaarden van de bogen van de tour van de beste ant ook aangepast worden aan de hand van de waarde van ρ. Beide parameters stonden tijdens onze experimenten telkens op 0.1, omdat dit in [4] als optimale waarde werd benoemd.

Probleem	Optimum	$\rho = 0.1$	$\rho = 0.3$	$\rho = 0.5$	$\rho = 0.7$	$\rho = 1$	rel. fout
berlin52	7542	**8056**	8228	8250	8464	8242	6.81%
tsp225	3919	**4465**	4560	4639	4539	4589	14.19%
pcb442	50778	**58366**	59294	59051	59654	59656	14.94%
gr666	294358	**347037**	350464	350284	348329	350181	17.89%
u724	41910	**50972**	51391	51415	51334	51250	21.62%

Tabel 5: Problemen met hun optima, ACS met verschillende waarden voor ρ. De relatieve fout is telkens berekend met de beste oplossing in het vet.

Probleem	Optimum	$\xi = 0.1$	$\xi = 0.3$	$\xi = 0.5$	$\xi = 0.7$	$\xi = 1$	rel. fout
berlin52	7542	8056	**7771**	7805	7991	8073	3.05%
tsp225	3919	4465	**4366**	4380	4446	4575	11.41%
pcb442	50778	58366	57674	**57377**	57706	58889	12.99%
gr666	294358	347037	342020	**338232**	340796	344389	14.90%
u724	41910	50972	49634	**49471**	49493	50744	18.04%

Tabel 6: Problemen met hun optima, ACS met verschillende waarden voor ξ. De relatieve fout is telkens berekend met de beste oplossing in het vet.

We bekijken wat de aanpassing van ρ als effect heeft op de resultaten. Deze resultaten plaatsen we in tabel 5. Het is duidelijk dat de waarde $\rho = 0.1$ voor ieder probleem van onze experimentenset het beste resultaat geeft. Hiermee bevestigen we wat men in [4] als beste waarde voor ρ vastlegde.

Verder kunnen we ook een alternatieve optimale waarde voor ξ bepalen. De resultaten voor verschillende waarden van ξ staan in tabel 6. Hier is het interessant om op te merken dat een verschillende waarde van ξ in sommige gevallen een beter resultaat geeft. Voor `berlin52` en `tsp225` is $\xi = 0.3$ de beste waarde. Voor `pcb442`, `gr666` en `u724` is $\xi = 0.7$ de beste waarde. Een hogere waarde voor ξ betekent dat de feromonenwaarden sterker verdampen. Zoals eerder vermeld zal ξ telkens de feromonenwaarden van iedere boog die een ant neemt laten verdampen. Een hogere verdampingswaarde zal ervoor zorgen dat de ants eerder gestuurd worden naar bogen met een potentieel betere uitkomst. Dit omdat de waarde van ρ in vergelijking 3 er voor zal zorgen dat de feromonenwaarden van de bogen van de beste tour positief versterkt worden. Een mogelijk verklaring voor deze cijfers kan zijn dat wanneer we de bogen die niet behoren tot een beste tour meer laten verdampen, de ants meer gestuurd worden naar de betere bogen.

4.2.3 Verschillende β-waarden

In [4] bespreken ze een waarde van β tussen 2 en 5. Daarom bekijken we wat het effect is van de waarden 2, 3, 4 en 5. De resultaten van dit experiment staan in tabel 7. Deze verschillende waarden zullen onderling geen verandering in de uitvoeringstijd aanbrengen.

Een eerste opmerkelijk resultaat is dat van `berlin52`. ACS verslaat hier TS uit tabel 2. Echter blijft TS (0.0178 s) veel sneller dan ACS (0.1130 s), al spreken we hier van zeer kleine verschillen. We zien in het algemeen dat hogere waarden voor β beter presteren. Het is moeilijk om uit deze resultaten een goede waarde uit te kiezen, maar starten met een $\beta = 4$ of 5 is zeker een goed begin.

Probleem	Optimum	$\beta = 2$	$\beta = 3$	$\beta = 4$	$\beta = 5$	rel. fout
berlin52	7542	8056	7820	**7723**	7841	2.40%
tsp225	3919	4465	4395	4340	**4339**	10.71%
pcb442	50778	58366	57846	**56933**	57163	12.10%
gr666	294358	347037	346055	342115	**338091**	14.85%
u724	41910	50972	49732	**49232**	49460	17.47%

Tabel 7: Problemen met hun optima, ACS met verschillende waarden voor β. De relatieve fout is telkens berekend met de beste oplossing in het vet.

Probleem	Optimum	TS	ACS	PSO	FA	BA	rel. fout
eil51	426	438	444	**437**	442	**437**	2.58%
berlin52	7542	7834	8022	**7667**	7678	7711	1.66%
kroA100	21282	**22221**	22951	22335	22586	22528	4.41%
eil101	629	**651**	703	673	670	670	3.49 %
pr107	44303	44490	46607	**46592**	46336	46727	5.17%
pr124	59030	**60363**	62691	64150	64505	64436	2.72%

Tabel 8: Verkregen optimalisatieresultaten voor TS met standaardparameters, ACS met standaardparameters, Particle Swarm Optimization (PSO), Firefly Algorithm (FA) en Bat Algorithm (BA). PSO, FA en BA resultaten komen uit [5]. De relatieve fout is telkens berekend met de beste oplossing in het vet.

4.3 Tabu Search, Ant Colony System en andere heuristieken

Ten slotte vergelijken we de resultaten van de eigen implementatie van TS en ACS met de resultaten van andere technieken samengevat in paper [5]. Die resultaten zijn te vinden in tabel 8. We zien dat ACS, FA en BA nooit een beste oplossing genereert, en dat TS en PSO het beter doen. Als we kijken naar de snelheid van de algoritmen, is TS zeker beter. Daarna volgen in volgorde FA, BA, PSO en ACS.

Besluit

We hebben TS en ACS op een efficiënte manier geïmplementeerd en leerden hieruit dat er voor de gegevensstructuren van TS belangrijke aspecten zijn zoals de tabu list en het uitvoeren van een 'move' waarop we moeten letten. Het implementeren van ACS was veel eenvoudiger in termen van gegevensstructuren. Er zijn wel meerdere parameters en formules nodig om tot een oplossing te komen. De experimenten lieten duidelijk zien dat de optimale parameters voor beide oplossingsmethodes afhangen van probleem tot probleem. Voor sommige problemen en parameters zijn er wel duidelijkere patronen zichtbaar. Over het algemeen presteert ACS slechter dan TS. Zeker voor grote problemen, waar de fout groter is en de uitvoeringstijd langer is. We kunnen dus besluiten dat TS met de juiste parameters zeker de winnaar is. Daarnaast zijn er zeker nog meerdere experimenten uit te voeren om te verifiëren of een conclusie met meer zekerheid correct is. Ten slotte kunnen we besluiten dat metaheuristieken zeker een relatief goede oplossing kunnen bieden voor het Traveling Salesman Problem.

Referenties

[1] Sumanta Basu. Tabu search implementation on traveling salesman problem and its variations: A literature survey. *American Journal of Operations Research*, 02, 01 2012.

[2] Wikipedia contributors. 2-opt — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=2-opt&oldid=1113143213`, 2022. [Online; accessed 3-December-2022].

[3] Wenbao Qiao and Jean-Charles Créput. Parallel 2-opt local search on gpu. *World Academy of Science, Engineering and Technology, International Journal of Electronics and Communication Engineering*, 4:307–311, 2017.

[4] Marco Dorigo and Stutzle Thomas. *3. Ant Colony Optimization Algorithms for the Traveling Salesman Problem.* MIT Press, 2004.

[5] Eneko Osaba, Xin-She Yang, and Javier Del Ser. *Traveling salesman problem: a perspective review of recent research and new results with bio-inspired metaheuristics*, pages 135–164. 05 2020.