Traveling Umpire Problem: Branch and Bound
Geavanceerde Artificiële intelligentie

Jonas Roels en Michiel Vandromme

19 juni 2024

Class: MEIsw

Inhoudsopgave

1 Inleiding **1**
1.1 Het Traveling Umpire Problem 1

2 Implementatie **1**
2.1 Lower bounds 2
2.1.1 Parallellisatie v1 2
2.2 Partial matching 2
2.3 Parallellisatie v2 3

3 Resultaten **4**

4 Conclusie **5**

1 Inleiding

Het Travelling Umpire Problem is een van de meest bekende problemen binnen het domein van optimalisatietechnieken. Onderzoekers strijden al lang om de optimale oplossing voor dit probleem in zo min mogelijk tijd te vinden, en ook wij hebben hiervoor een poging gedaan. In dit verslag bespreken we onze oplossingsmethode, gebaseerd op het onderzoek van Toffolo et al. [1]. Hierin wordt gebruik gemaakt van een Branch-and-Bound aanpak, waarop wij enkele aanpassingen en uitbreidingen hebben toegepast. We bespreken de resultaten en geven inzicht in welke elementen de grootste prestatieverbeteringen opleveren. Tot slot presenteren we enkele ideeën die nuttig kunnen zijn voor toekomstig onderzoek.
De volledige broncode voor dit project is beschikbaar op:
`https://github.com/RoelsJonas/Geadvanceerde-AI-Project-Michiel-VDR-Jonas-R`

1.1 Het Traveling Umpire Problem

TUP is een assignment probleem voor een competitie met n teams waarbij $\frac{n}{2}$ umpires aan $\frac{n}{2}$ games moeten worden toegewezen, gedurende $4n - 2$ rondes . Hierbij moeten we met de volgende constraints rekening houden [1]:

a. elke game moet exact één umpire toegekend worden;

b. elke umpire moet elke ronde werken;

c. elke umpire moet de thuislocatie van elke team minstens eenmaal bezoeken;

d. geen enkele umpire mag binnen q_1 opeenvolgende ronden dezelfde locatie bezoeken;

e. geen enkele umpire mag binnen q_2 opeenvolgende ronden aan een game met hetzelfde team worden toegewezen.

2 Implementatie

De implementatie van dit project in Java begon met het opstellen van een oplossingsvoorstelling. Hiervoor creëerden we van drie klassen, namelijk: *Game, Umpire en Solution*. De *Solution* klasse gebruiken we om de toekenningen van Umpires aan Games vast te leggen en de totale kost bij te houden. Deze oplossingsvoorstelling lijkt op het *Multi-line umpire-oriented format* voorgesteld in de output handleiding van de TUP website [2]. Dit maakt het simpel om gebruik te maken van de voorziene validator voor het controleren van oplossingen.

De *Umpire* klasse bevat simpelweg een ID en twee integer arrays, dit noemen we respectievelijk de Q1Counters en Q2Counters. Deze gebruiken we, zoals de naam suggereert, om in constante tijd de constraints ((d) en (e)) die te maken hebben met de q_1- en q_2-parameters te checken. Verder zorgen deze er ook voor dat we in lineaire tijd constraint (c) kunnen checken. Door simpelweg een van deze arrays te overlopen en checken of deze niet gelijk zijn aan de initiële waarde.

De *Game* klasse bestaat uit een 'thuis'- en 'uit'-team, dat wordt voorgesteld door een integer. Verder vinden we hier ook een adjacency list in die de games van de volgende ronde bevat en de afstanden tot die games. Zo bekomen we dus een gewogen, gerichte graaf van games. Zoals ook vermeld door Tofollo et al. kunnen we hieruit reeds enkele verbinding verwijderen doordat deze altijd constraits (d) en (e) zullen overtreden (volgende game bevat eenzelfde team of is op dezelfde locatie) [1].

2.1 Lower bounds

Voor de generatie van lower bounds implementeerden we het algoritme gepresenteerd door Tofollo et al. [1]. Hieraan maakten we initieel weinig aanpassingen, maar merken we graag nog een inzicht op. Hoewel het niet mogelijk is om met alle constraints rekening te houden bij het oplossen van de subproblemen, hoeven we deze toch niet volledig buiten beschouwing te brengen. Bijvoorbeeld constraint (c) kan in geen enkel subprobleem volledig geverifieerd worden, echter kunnen we deze verzwakken tot de mogelijkheid om alle locaties te bezoeken. Met andere woorden, er moeten dus nog voldoende rondes niet in het subprobleem verwerkt zitten zodat de umpire in kwestie nog alle locaties kan bezoeken.

2.1.1 Parallellisatie v1

Om aan snelheid te winnen, was één van onze eerste ideeën om parallellisatie toe te passen op het berekenen van de lower- en upper bounds. Als we deze tegelijk zouden laten berekenen, geraken we sneller aan een resultaat. Hierbij is namelijk geen synchronisatie nodig aangezien er slechts één enkele thread zal schrijven naar de matrix en één enkele andere hieruit zal lezen. Dit impliceert wel dat de BnB die de upper bounds berekent meer nodes zal bezoeken doordat initieel de lower bounds zwakker zijn en er dus minder gesnoeid kan worden in de zoekruimte. Eigenlijk staan zelfs veel van de vaak gebruikte waarden uit de lowerbounds matrix op nul omdat hiervoor de nodige subproblemen nog niet zijn opgelost. Om dit probleem toch te verzachten hebben we *intermediary bounds propagation* geïmplementeerd. Wanneer deze feature aanstaat, propageren we bij iedere update in de laatste kolom van de lower bounds matrix de waarde naar boven door telkens het maximum te nemen van de huidige waarde en die eronder.

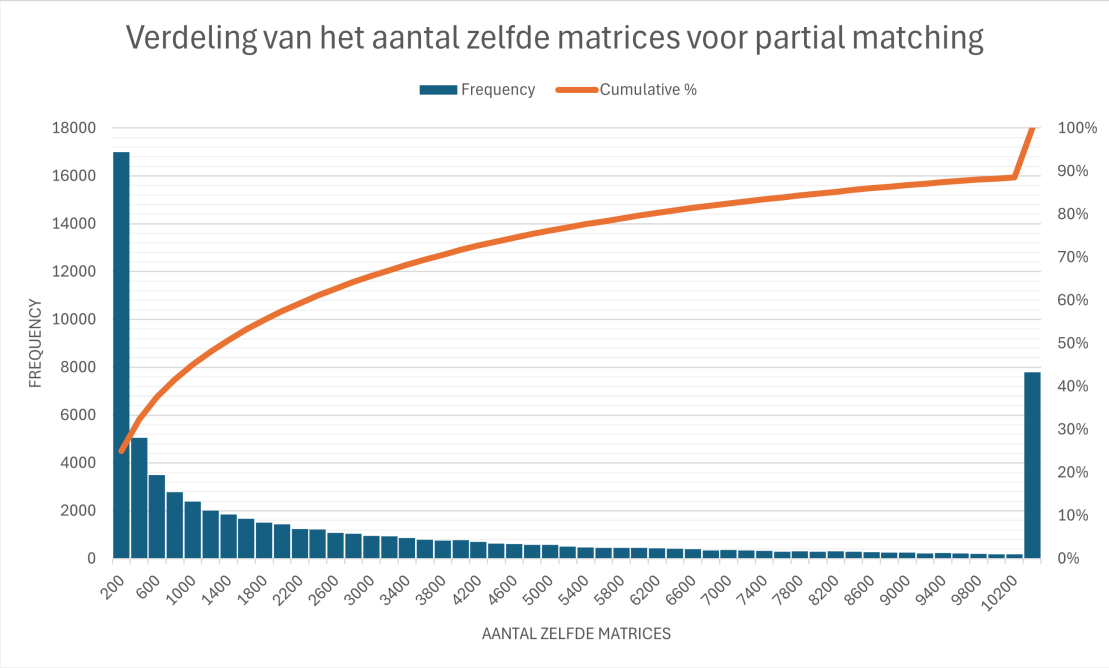
Dit simpel algoritme maakt dus tijdens de berekening de tussentijdse lower bounds sterker. Echter zijn deze vaak te laag om effectief grote stukken van de zoekruimte weg te snijden waardoor er slechts een kleine verbetering in het aantal bezochte nodes en rekentijd is.

2.2 Partial matching

Tofollo et al. maken in hun paper gebruik van partial matching aan de hand van het *Hongaars algoritme* [1]. Echter merkten we al snel op dat hoewel dit het aantal verkende nodes drastisch doet dalen we zonder deze aanpassing vrijwel dezelfde rekentijd bekwamen. Dit kunnen we wijten aan het feit dat dit algoritme een tijdscomplexiteit van $O(n^3)$ heeft [3].

In een eerste poging dit te versnellen, maakten we gebruik van een *HashMap* datastructuur om voordien berekende waarden op te slaan. Dit aangezien we merkten dat het Hongaars algoritme vaak meerdere malen op dezelfde matrix werd uitgevoerd, dit zien we ook in Figuur 1. Bij die verdeling merken we op dat we gemiddeld gezien 4935 keer het Hongaars algoritme met dezelfde matrix uitvoeren. Dit is op een instantie met 14 umpires, voor grotere instanties zal deze waarde verder toenemen. Echter slaagde dit er niet in een verbetering op te leveren door de tijdsverspilling in het opstellen en het hashen van de matrix.

Vervolgens probeerden we een greedy aanpak waarbij we telkens de laagste resterende kosten voor de volgende umpires er bij telden ongeacht of dit een feasible suboplossing vormt of niet. Echter leverde dit vrij zwakke bounds op waardoor er meer nodes bezocht werden en de totale rekentijd vrijwel niet afnam. Een verdeling van het verschil in bounds tussen partial matching met het Hongaars algoritme en met deze greedy aanpak zien we in Figuur 2. We merken op dat deze verdeling een mediaan van 1341 heeft wat het aantal prune mogelijkheden beperkt maar verder is er ook een



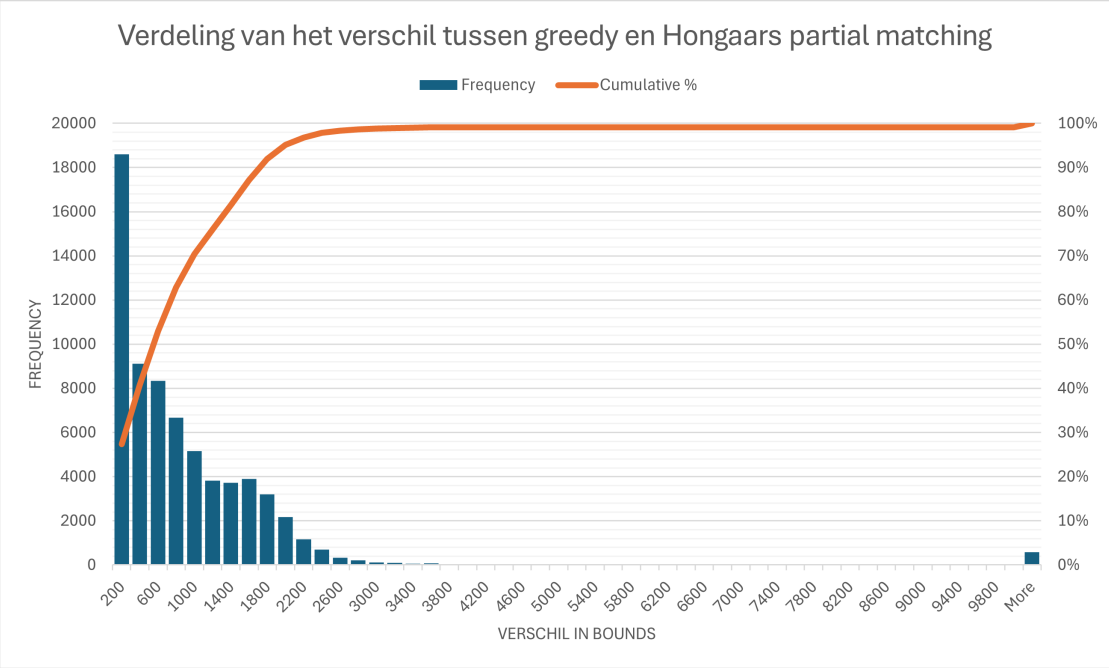
Figuur 1: Verdeling van de matrices gebruikt bij partial matching voor instantie 14 7_2

grote hoeveelheid groter dan 10000, dit stellen de gevallen voor waar er geen mogelijke opvulling gevonden wordt met het Hongaars algoritme en we dus al snel kunnen concluderen dat de huidige oplossing infeasible is.

Tot slot verkenden we een tweede greedy aanpak waar we voor aanvang van de hoofd branch-and-bound al partial bounds genereren. Aan de hand van het Hongaars algoritme bepalen we voor elke ronde de optimale toewijzing van umpires, deze optelling geeft $\frac{n}{2}$ afstanden, die we vervolgens in stijgende volgorde sorteren. In de partial bounds matrix slaan de cumulatieve som van deze waarden op.
Hoewel deze partial bounds nog steeds zwakker zijn dan die geproduceerd met de partial matching zoals in de implementatie van Toffolo et al. (2016) [1], wat blijkt uit het toegenomen aantal bezochte nodes, is deze aanpak aanzienlijk sneller. Dit komt doordat het Hongaars algoritme slechts $4n - 2$ keer wordt uitgevoerd in plaats van bij bijna elke node.
Ten opzichte van de vorige greedy aanpak verbeterd dit de partial bounds met een gemiddelde waarde van 294 bij de 14 7_3 umpire.

2.3 Parallellisatie v2

In de laatste fase van ons project maakten we een serieuze uitbreiding op de parallellisatie van onze implementatie wat grotere verbeteringen in rekentijd mogelijk maakte. Hiervoor vonden we meerdere opties. Zo kunnen we eerst en vooral de zoekruimte opsplitsen in delen en elke thread een deel van de zoekruimte laten verkennen. Echter vergt dit een grote hoeveelheid synchronisatie en vormt dit problemen met de correctheid van de oplossing. Ook zien we de mogelijkheid om de generatie van lower bounds te parallelliseren, zo kunnen we verschillende subproblemen tegelijkertijd



Figuur 2: Verdeling van het verschil in partial bounds voor instantie 14 7_2

op verschillende threads oplossen aangezien deze onafhankelijk zijn. Ook dit vereist echter wel aanpassingen aan onze BnB solver.
Desondanks implementeerden we deze techniek wat tot een grote verbetering in prestaties leverde. Specifiek parallelliseren we de upperbounds zoekboom door een thread op te starten waarin we de eerste toewijzing van de eerste ronde fixeren. Dit wil dus zeggen dat er $\frac{n}{2}$ threads zullen starten. Echter zullen hiervan enkele meteen terug afsluiten doordat ze constraint (d) of (e) overtreden.
Voor de parallellisatie van de lower bounds wordt telkens in parallel een rij van de subproblems matrix S ingevuld. Dit zorgt voor verdere verbeteringen in prestaties maar is mogelijks suboptimaal, zo merken we dat vaak de lower bounds pas volledig berekent zijn nadat er al een optimale oplossing gevonden is. Als poging om dit te beperken hebben we geëxperimenteerd met het gebruik van scheduling priorities waarbij de threads die lower bounds berekenen, een hogere prioriteit krijgen (en dus voorrang zouden moeten krijgen) tijdens het executen. Dit gaf jammer genoeg geen merkbare verschillen in de uitvoeringstijd. Daarnaast hebben we ook geëxperimenteerd met parallellisatie binnen één subprobleem. Dit zorgt voor extra synchronisatie-overhead waardoor uiteindelijk ook bleek dat de uitvoeringstijd niet verbeterde. De feature is in de finale versie er dus weer uitgehaald, maar de code is nog te vinden in de 'multi-extended-parallel'-branch.

3 Resultaten

Figuur 3 geeft de invloed van de verschillende verbeteringen weer voor instantie 14 met $q1 = 7$ en $q2 = 3$. Hierbij merken we op dat de grootste verbeteringen in rekentijd niet altijd gelinkt zijn aan de grootste verbeteringen in het aantal bezochte nodes. Ook merken we op dat de voornaamste afname in rekentijd komt door de parallellisatie. Dit zorgt voor een verdubbeling van het aantal

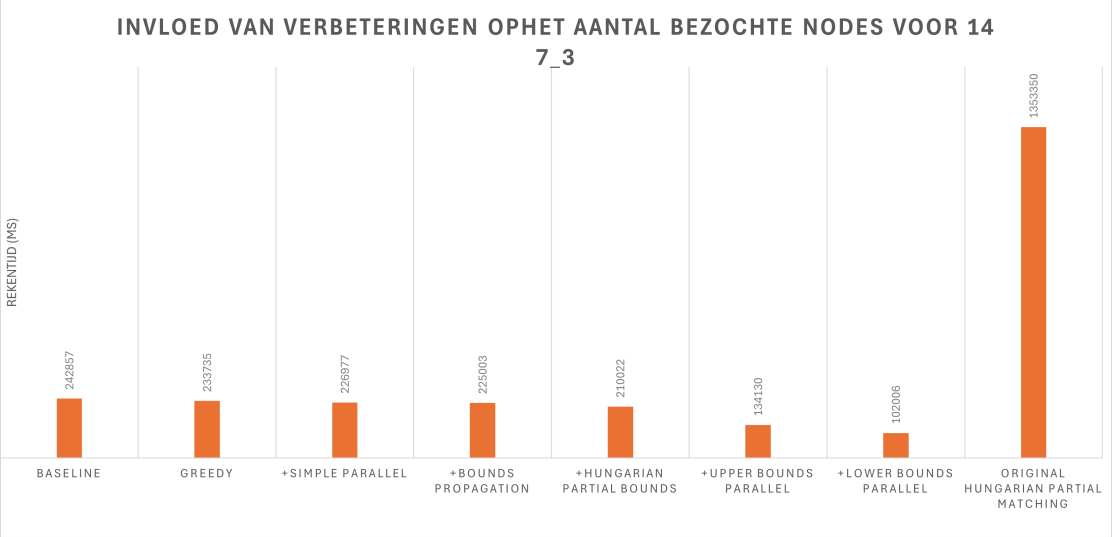
nodes dat per seconde verkent wordt.
Op Figuur 4 zien we een vergelijking tussen ons algoritme en dat van Tofollo et al. [1]. Hierbij vergelijken we zowel met de waarden uit de paper als wanneer we de solver zelf lokaal runnen. Dit doen we om zo afwijkingen in de resultaten die te wijten zijn aan een verschil in rekenkracht in kaart te brengen. We merken echter op dat er slechts een klein verschil in rekentijd is, de lokale versie is gemiddeld 3,62% sneller. Voor verdere vergelijkingen verwaarlozen we dit verschil.

In Figuur 4b merken we op dat ons algoritme aanzienlijk meer nodes bezoekt. Dis is te wijten aan het feit dat onze partial bounds zwakker zijn dan die gecreëerd met partial matching. Echter is er toch een verbetering in rekentijd voor de meeste instanties. Dit bevestigt Figuur 5 verder. Hier zien we dat voor problemen met een groter aantal umpires de winst in rekentijd groter is. Hier meten we namelijk gemiddeld gezien een afname van 79,95% in rekentijd.

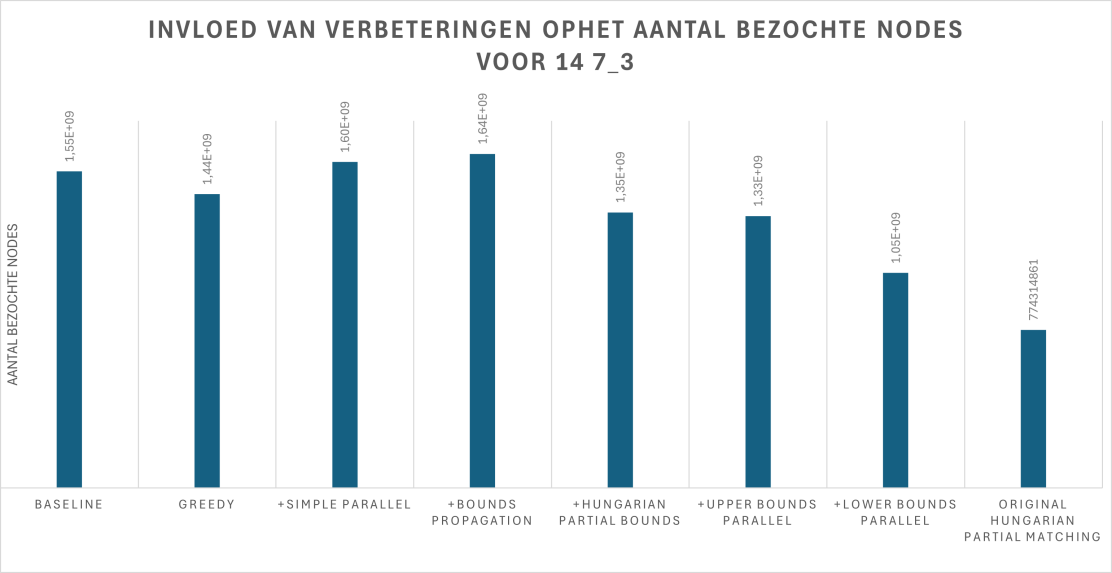
4 Conclusie

Na het zien van deze resultaten kunnen we vaststellen dat onze finale versie de rekentijden van Toffolo et al. verpulverd voor grotere instanties met meer umpires. Bij kleinere instanties zijn de rekentijden vrij competitief. Dit komt door de overhead bij het aanmaken van threads en het feit dat onze lower bounds pas berekend zijn nadat de effectieve BnB al klaar is. De pogingen die gedaan zijn om dit probleem te verbeteren, bleken geen significante invloed te hebben. In toekomstig onderzoek kan op zoek gegaan worden naar nog andere manieren om lower bounds te berekenen die mogelijks sneller goede resultaten opleveren. We denken hierbij aan heuristieken die parallel lopen met de branch-and-bound.

Merk op dat in onze implementatie geen gebruik gemaakt van een local search in de leaf-nodes van de branch-and-bound-boom zoals Toffolo et al. wel beschrijft. In toekomstig onderzoek kan hier nog op verder gewerkt worden. Daarnaast kan ook verder geëxperimenteerd worden met de volgorde waarin de branch-en-bound-boom wordt afgehandeld. Nu werd altijd op kortste afstand gesorteerd, maar misschien zijn er andere volgordes die sneller tot een optimale oplossing leiden. Tot slot willen we graag benadrukken dat onze benchmarks op vier verschillende PCs zijn uitgevoerd en telkens 100% van de CPU werd gebruikt bij het uitvoeren. Dit wilt zeggen dat er wellicht met verticale schaling van de hardware nog significante winst te halen valt. Enkele minieme procentjes zouden nog gehaald kunnen worden door onze benchmarks te doen op basis van JARs in plaats van IntelliJ Applications. Ook een herschrijving naar een taal zoals Rust of C++ zal naar grote waarschijnlijkheid snelheidswinst opleveren.

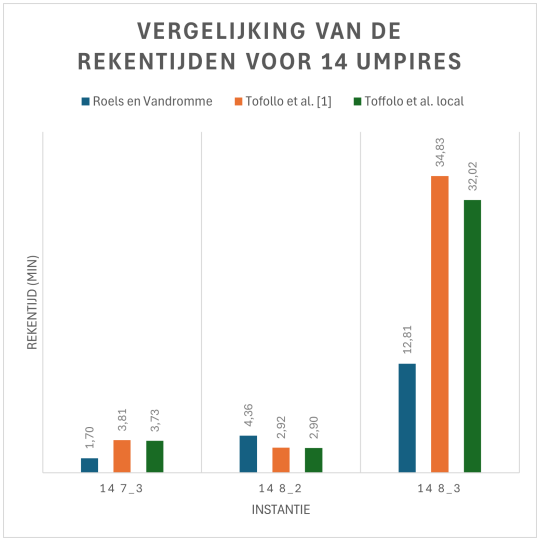


(a) Rekentijd

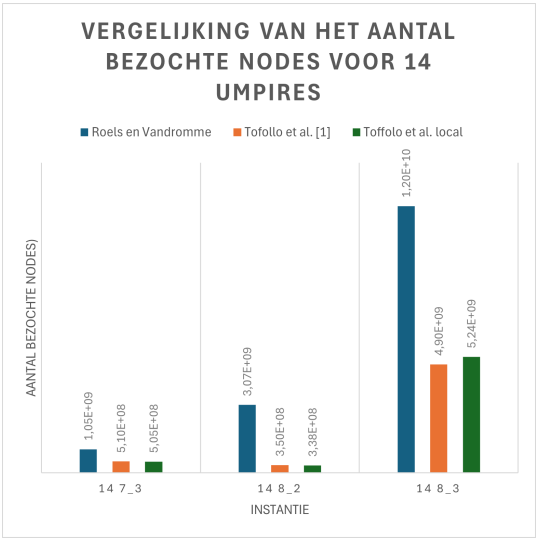


(b) Aantal bezochte nodes

Figuur 3: Invloed van de verschillende verbeteringen voor instantie 14 7_3

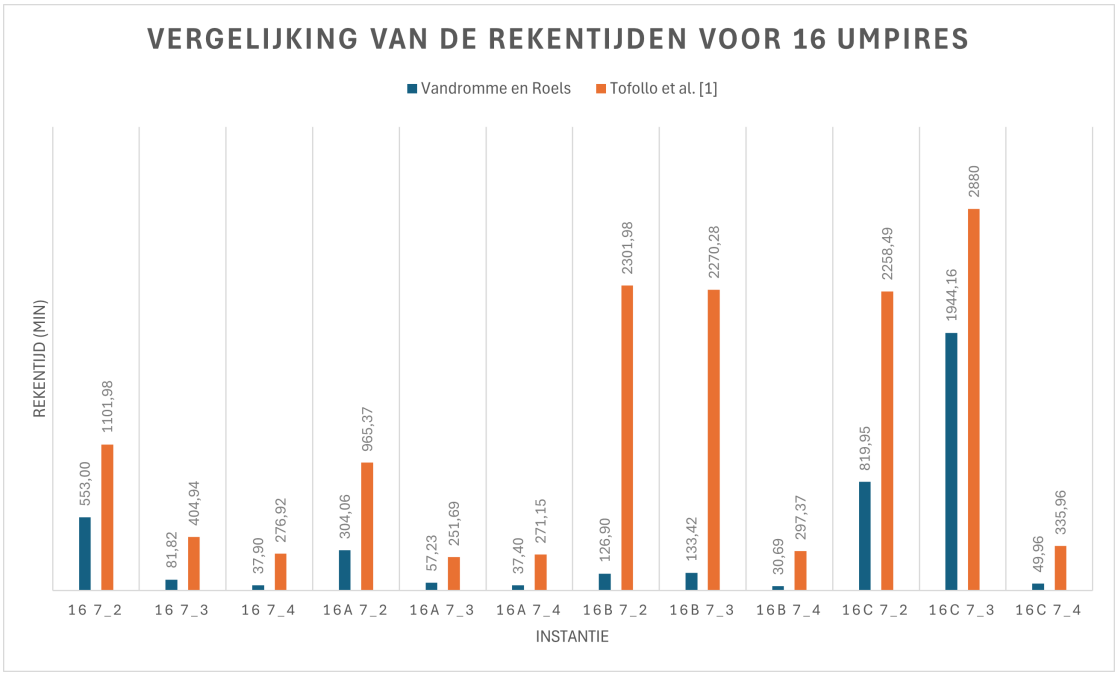


(a) Rekentijd



(b) Aantal bezochte nodes

Figuur 4: Vergelijkingen voor 14 umpires



Figuur 5: Vergelijking van de rekentijden voor 16 umpires

Instantie	Upper bound	Rekentijd (min)	Nodes
12 7_2	86889	0,009016667	4029728
12 6_3	Infeasible	0,00395	761146
12 5_3	93679	0,004033333	1236951
12 4_3	89826	0,0107	4931058
14 8_3	172177	12,81338333	1,20E+10
14 8_2	147824	4,357233333	3,07E+09
14 7_3	164440	1,7001	1,05E+09
14 7_2	146656	0,963716667	7,31E+08
14 6_3	158875	0,627866667	516894783
16 8_2	178428*	2880	2,02834E+12
16 7_4	197028	37,89968333	22912369540
16 7_3	165765	81,81973333	76504664229
16 7_2	150433	552,9979	4,95144E+11
16A 8_3	193302*	2880	2,184426+E12
16A 7_4	213416	37,40246667	27602032725
16A 7_3	178511	57,2344	56230001156
16A 7_2	163709	304,0559167	2,34715E+11
16B 7_4	223868	30,68685	30331978509
16B 7_3	180204	133,4216833	1,09697E+11
16B 7_2	167377	126,9049167	1,1308E+11
16C 7_4	209088	49,96365	29412692249
16C 7_3	178215	1944,162217	1,76094E+12
16C 7_2	166855	819,9527167	5,06105E+11

Tabel 1: Resultaten van de verschillende instanties

Referenties

[1] T. A. Toffolo, T. Wauters, S. Van Malderen, and G. Vanden Berghe, "Branch-and-bound with decomposition-based lower bounds for the traveling umpire problem," *European Journal of Operational Research*, vol. 250, no. 3, pp. 737–744, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037722171500911X

[2] T. A. M. Toffolo, T. Wauters, and M. Trick. (2015, Jun.) An automated benchmark website for the traveling umpire problem. [Online]. Available: http://gent.cs.kuleuven.be/tup

[3] Y. Varyani. Hungarian algorithm for assignment problem | set 1 (introduction). Section: DSA. [Online]. Available: https://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduction/