Parallelle algoritmen: het handelsreizigersprobleem

Project Algoritmen en Datastructuren III

Bart Middag

Academiejaar 2013-2014

1. IMPLEMENTATIE

1.1 Het algoritme

Het hoofdprogramma werkt als volgt:

Eerst worden alle nodige variabelen geïnitialiseerd. Hierna berekenen we een benedengrens aan de hand van `min_door`, zoals gedefinieerd in de cursus Datastructuren en Algoritmen II. [1]
Deze grens wordt nog verbeterd door de helft van de kleinste afstand vertrekkende uit de eerste en laatste geselecteerde stad er bij op te tellen. Om afrondingsfouten in deze berekening te voorkomen, vermenigvuldigen we ook alle alle afstanden met 2. We verliezen hier ook niets van het gevraagde bereik (`int`), want we gebruiken `unsigned int` om de getallen voor te stellen.

Hierna berekenen we nog een andere, betere benedengrens: de benedengrens van Held-Karp met de methode van Held, Wolfe en Crowder (met deze methode kunnen we de grens berekenen aan de hand van een bovengrens). [2]

We bekijken dan de lengte van de huidige rondreis: 0->1->2->…->N->0
Als dit gelijk is aan de Held-Karp benedengrens, mag het algoritme al stoppen. Zoniet voeren we 2-opt en 3-opt uit (en daarna weer 2-opt) en kijken we opnieuw of de lengte van de beste rondreis gelijk is aan de Held-Karp benedengrens. Voor veel gevallen met minder dan 30 steden is dit zo en moet het hoofdalgoritme dus niet eens beginnen.

Hierna bepalen we een goede stad om de rondreis in te beginnen. We nemen hiervoor de stad met de grootste `min_door` – zo wordt de benedengrens hoger en zullen we vroeg kunnen bounden. De rest van de steden sorteren we op stijgende Held-Karp waarde, zodanig dat we de beste bovengrens vroeg vinden en veel paden kunnen overslaan naar het einde toe.

Voor gevallen met 60 of meer steden proberen we nog een aantal keren een willekeurige rondreis uit, die we ook proberen te verbeteren met een combinatie van 2-opt en 3-opt. Doordat elk proces andere rondreizen kan uitproberen, zullen we met 10+ pogingen waarschijnlijk al een betere bovengrens hebben.

Hierna beginnen we met het branch & bound-algoritme. We selecteren telkens een nieuwe stad om te bezoeken vanuit de huidige stad, en berekenen hiervoor de Held-Karp waarde. Als deze waarde en de waarde van `min_door` kleiner zijn dan de bovengrens, mogen we verdergaan met deze stad. Anders proberen we een andere stad uit. Als we op het einde komen van het pad, passen we de bovengrens aan als de lengte van de huidige rondreis kleiner is dan de huidige bovengrens. Dan proberen we de rondreis nog te verbeteren met 3-opt.

Tijdens het branch & bound-algoritme houden we ook voor twee steden een bepaalde volgorde bij: we mogen de tweede stad enkel binnengaan als we de eerste al bezocht hebben. Zo vermijden we dat we van elk pad ook het omgekeerde pad bekijken.

Van het branch & bound-algoritme zijn er 2 implementaties: een recursieve implementatie en een (ietwat minder elegante) implementatie die minder gebruik maakt van de stack.

1.2 Datastructuren

Omdat we werken met een graaf, lijkt het logisch om hiervan de toppen en bogen bij te houden in een special hiervoor gemaakte datastructuur. Toch doen we dit niet: doordat we in het branch & bound-algoritme telkens andere verbindingen moeten maken tussen 2 verschillende steden, lijkt het geen goede keuze om deze verbindingen expliciet bij te houden. Ook willen we gemakkelijk kunnen weten welke steden nog niet bezocht zijn.

We gebruiken hiervoor een gewone array waarin de nummers van de steden worden bijgehouden. De bezochte steden worden vooraan in de array bijgehouden in de volgorde waarin ze bezocht zijn. Telkens als we dus een stad kiezen achteraan de array, verwisselen we deze gewoon van plaats met de stad net achter de bezochte steden.

De getallen in deze array gebruiken we ook als indices in vele andere arrays, zoals de afstandsmatrix.

1.3 Parallelle implementatie

Op de splitsdiepte worden de taken verdeeld tussen de verschillende processen. Dit is de diepte waar er $2 * (aantal\ processen)^4$ toppen zijn.

We hebben ervoor gekozen om te kunnen bounden vóór de splitsdiepte. Dit is belangrijk voor grote problemen: het aantal toppen stijgt exponentieel als we dalen in de recursieboom.
Om dit te kunnen doen, mogen we geen gebruik maken van een teller die gewoon opgehoogd wordt bij het bezoeken van een top op de splitsdiepte. In plaats hiervan berekenen we aan de hand van het bezochte pad het nummer dat de teller voor elke top op de splitsdiepte zou hebben als we niet zouden gebound hebben voor de splitsdiepte.

Tijdens het algoritme worden betere bovengrenzen doorgestuurd tussen processen: een proces stuurt zijn beste bovengrens eerst naar proces 0, dat dan ook de bovengrens doorstuurt als die intussen nog niet verbeterd is. Dit wordt gedaan met `MPI_Isend` en `MPI_Irecv`, zodat het programma ondertussen kan doorgaan. [3]

Op het einde stuurt elk proces nog eens zijn bovengrens door naar proces 0 met `MPI_Send` (er moet immers niets meer gebeuren), waarna het op een antwoord wacht van proces 0. Dit antwoord is ofwel `'q'` (van “quit”) of `'p'` (van “print”). Enkel het proces dat de laatste keer de bovengrens verbetert heeft, zal dus zijn beste route uitprinten.

1.4 Heuristieken

Zoals eerder vermeld was het startpunt van dit algoritme het algoritme met `min_door` dat beschreven staat in de cursus Datastructuren en Algoritmen II. [1]

Hierbij heb ik de volgende heuristieken geïmplementeerd:

1.4.1 2-opt en 3-opt

2-opt (resp. 3-opt) proberen een betere bovengrens te vinden door telkens alle mogelijke 2 (resp. 3) verbindingen te nemen in de huidige beste rondreis, en die te verwijderen. De stukken van elke rondreis worden dan aan elkaar geplakt op elke mogelijke manier. Als één van deze nieuwe rondreizen een betere bovengrens geeft, wordt deze behouden en begint het algoritme opnieuw.

Deze heuristieken heb ik geïmplementeerd omdat een combinatie van 2-opt en 3-opt met relatief weinig computationele kost een bovengrens kan bepalen die al dicht bij het optimum ligt.

1.4.2 De benedengrens van Held-Karp

Deze benedengrens voor het handelsreizigersprobleem is een verbetering van de benedengrens bepaald door de minimale 1-boom van de graaf met alle steden (d.i. de twee kleinste bogen vertrekkend uit één van de toppen van de graaf + de minimale opspannende boom van de rest van de graaf). We berekenen de minimale 1-boom gedurende verschillende iteraties. Door aan elke stad die niet 2 keer bezocht wordt een ‘straf’ toe te kennen, proberen we ervoor te zorgen dat elke stad maar 2 keer bezocht wordt, zoals in een rondreis het geval zou moeten zijn. Zo bepalen we de benedengrens die momenteel ook in gekende TSP solvers zoals Concorde gebruikt wordt.

In mijn versie van het algoritme werk ik met de formule van Held, Wolfe en Crowder – zo kan ik de huidige bovengrens gebruiken om de benedengrens te bepalen. Het originele algoritme van Held-Karp werkt immers met een schatting van de benedengrens die moeilijk te bepalen is – computationeel is het dus veel interessanter om gewoon de bovengrens te gebruiken.

Het algoritme zelf is vrij complex en de uitleg hiervan valt buiten de scope van dit verslag. Meer uitleg kan u echter vinden in “Estimating the Held-Karp lower bound for the geometric TSP”. [2] In tegenstelling tot wat de titel doet vermoeden, is de bovengrens van Held-Karp ook goed voor het niet-geometrische handelsreizigersprobleem.

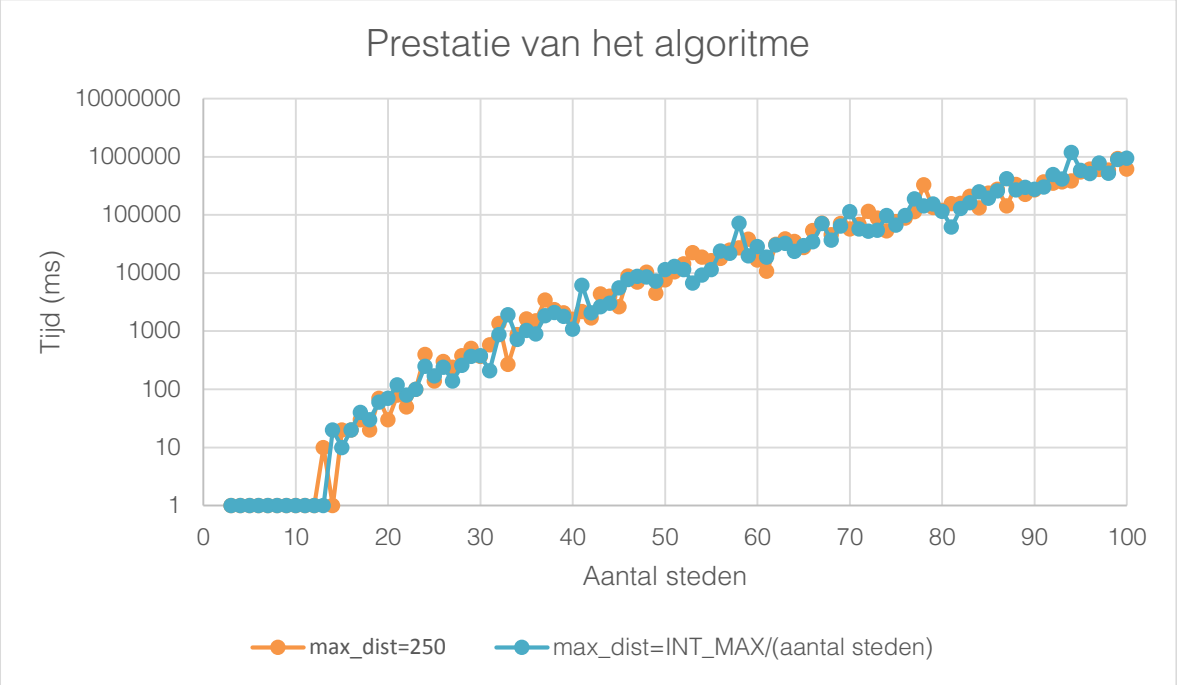
Dit heb ik geïmplementeerd omdat de bovengrens al geen probleem meer was met 2-opt en 3-opt. Het algoritme werd vooral vertraagd doordat de benedengrens (`min_door`) te zwak was. Toch heb ik ook `min_door` in het algoritme gehouden, omdat het algoritme hiermee toch nog een aantal takken kon wegsnijden zonder de Held-Karpwaarde te moeten berekenen (wat computationeel toch duurder is).

2. TESTS

De volgende tests werden uitgevoerd op de STEVIN supercomputerinfrastructuur van Universiteit Gent. Alle tijden zijn gemeten in CPU-tijd. Indien niet anders vermeld, zijn alle tijden bekomen met 2 nodes en 8 processen per node (dus 16 processen).

2.1 Tijd en correctheid

Om de tijd te meten, werden er testbestanden gegenereerd van 3 tot en met 100 steden. Van elk aantal steden werden er twee bestanden gegenereerd: één waar de maximumafstand 250 was, en één waar de maximumafstand $INT_{MAX}/(aantal\ steden)$ was. De tijdsprestaties van beide reeksen waren quasi gelijk, zoals te zien op Figuur 1.



Figuur 1: De prestatie van het branch & bound-algoritme. Op de y-as staat de tijd in ms weergegeven in logaritmische schaal. Op de x-as staat het aantal steden.

Om de correctheid te testen van het programma, maakte ik gebruik van tests uit TSPLIB. De optimale oplossingen van deze problemen zijn gekend en ook verkrijgbaar op de website. Het algoritme gaf de optimale oplossing voor alle geteste problemen. [4]

Probleem	Tijd (ms)
p01	10
gr17	10
gr21	40
ulysses22	60
fri26	160
bayg29	530
bays29	940
dantzig42	1030
swiss42	4780

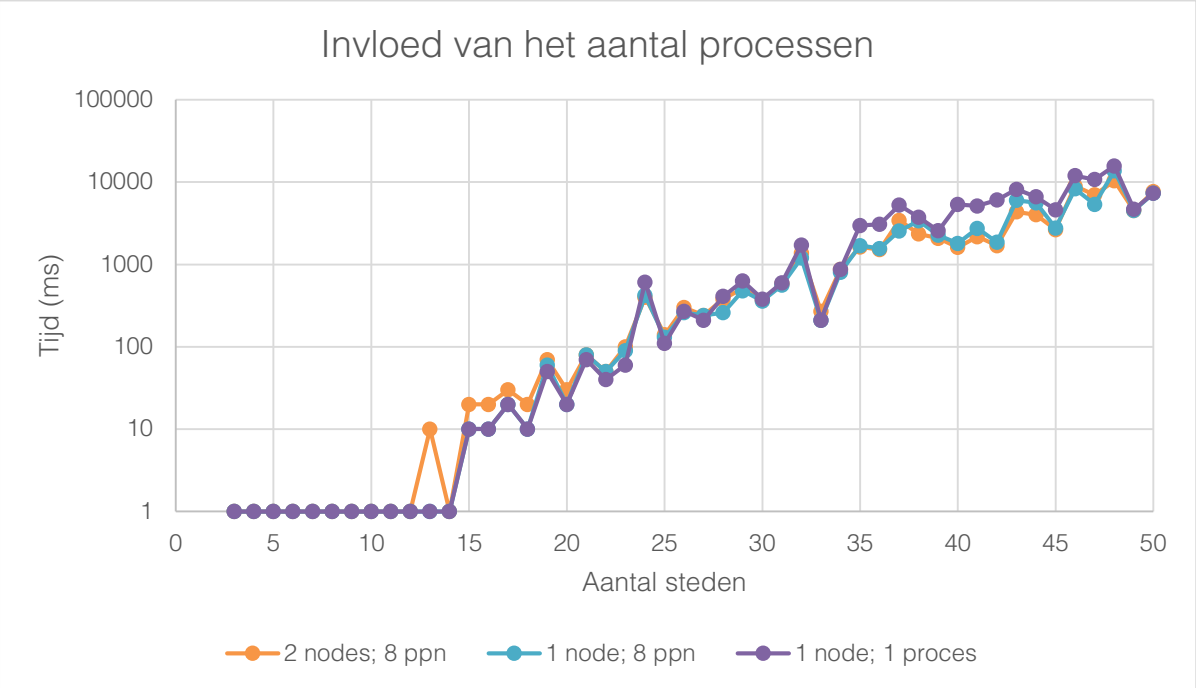
gr48	74880
hk48	8220
brazil58	358730
brg180	1033290

Opmerkelijk hier is dat het programma de oplossing kon vinden voor een probleem van 180 steden in amper 17,2 minuten!

2.2 Invloed van het aantal processen

Om de invloed van het aantal processen te testen, gebruikte ik de tests met maximumafstand 250 uit deel 2.1 en varieerde ik het aantal processen. Uit Figuur 2 wordt duidelijk dat bij grotere problemen het aantal processen toch een invloed heeft op de uitvoeringstijd. Meer processen betekent een kleinere uitvoeringstijd voor grotere problemen.

Voor problemen tot ongeveer 25 steden is dit echter niet zo. Dit is logisch – de splitsdiepte wordt hier bijna nooit bereikt omdat de benedengrens van Held-Karp en de bovengrens te goed bepaald zijn. Door de extra kost van het doorsturen van de grenzen, maken meerdere processen het algoritme trager voor weinig steden.

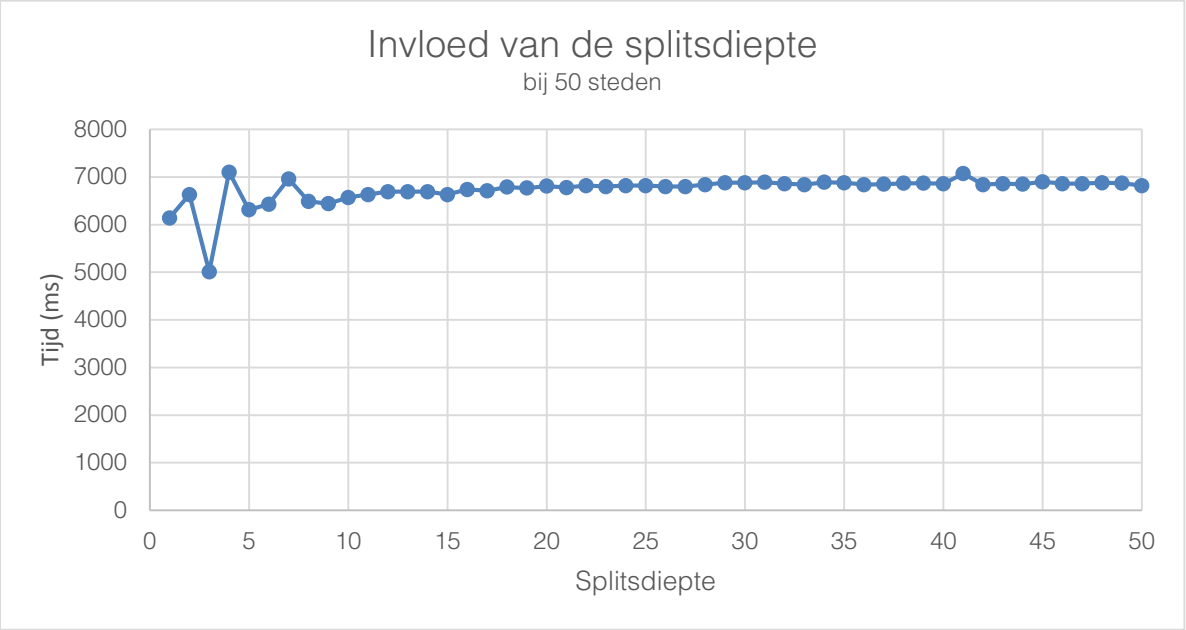


Figuur 2: De invloed van het aantal processen. Op de y-as staat de tijd in ms weergegeven in logaritmische schaal. Op de x-as staat het aantal steden.

2.3 Invloed van de splitsdiepte

De beste splitsdiepte is meestal niet te diep. Dit is duidelijk op Figuur 3, waarin we alle splitsdieptes getest hebben op een voorbeeld van 50 steden. De beste splitsdiepte hangt

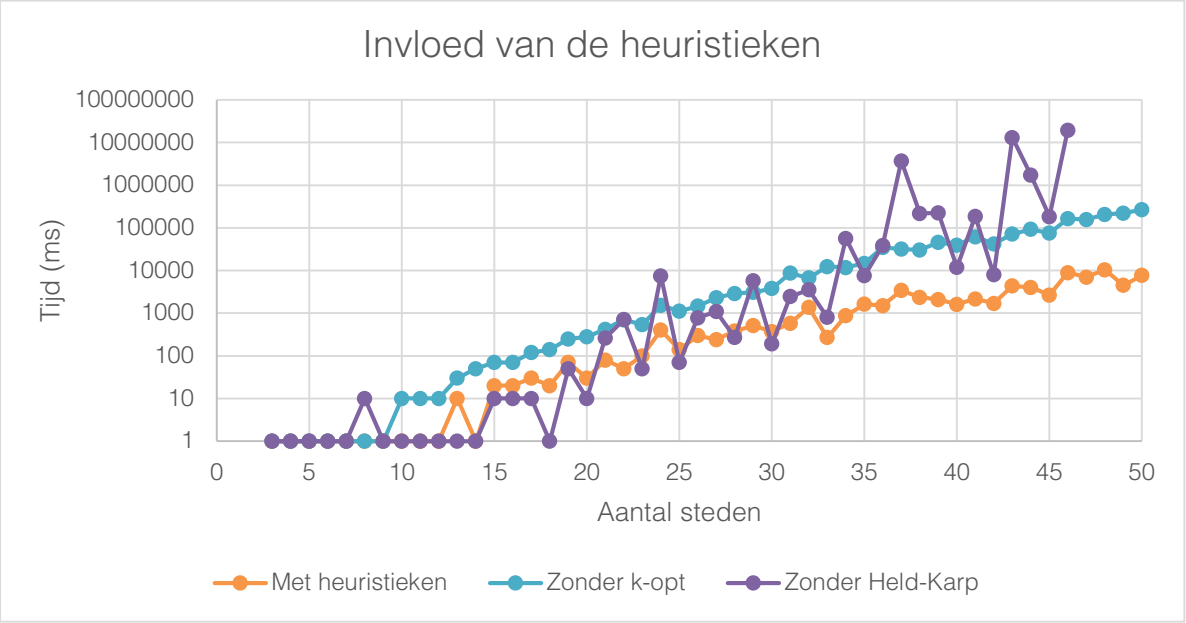
echter af van voorbeeld tot voorbeeld. Zoals eerder vermeld gebruiken we de splitsdiepte waarop er $2 * (aantal\ processen)^4$ toppen zijn.



Figuur 3: De invloed van de splitsdiepte (bij 50 steden). Op de y-as staat de tijd in ms weergegeven. Op de x-as staat de splitsdiepte.

2.4 Nauwkeurigheid van de heuristieken

Dankzij de heuristieken kan het algoritme veel sneller lopen. Dit is duidelijk zichtbaar op Figuur 4. Merk op dat zonder Held-Karp problemen van groter dan 45 steden niet meer oplosbaar zijn binnen redelijke tijd! Dit biedt dus een hele grote versnelling.

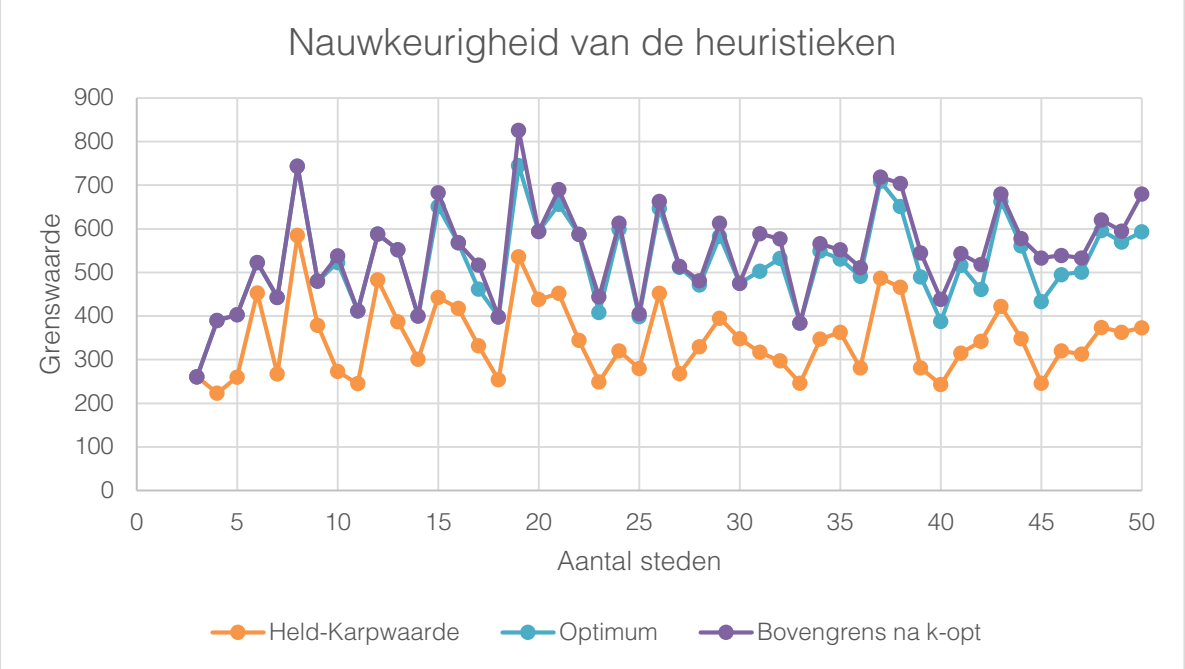


Figuur 4: De invloed van de heuristieken op de tijd. Op de y-as staat de tijd in ms weergegeven in logaritmische schaal. Op de x-as staat het aantal steden.

Beide grenzen liggen relatief dicht bij het optimum voor het branch & bound-algoritme begint. Op Figuur 5 wordt het verschil duidelijk weergegeven.

We zien dat de Held-Karpwaarde in het begin toch nog niet zo dicht ligt bij het optimum als men zou verwachten van de best gekende benedengrens voor het handelsreizigersprobleem. Dit is omdat we werken met willekeurige waarden die afstanden voorstellen maar eigenlijk geen afstanden zijn – als we testen met een probleem dat aan de driehoeksongelijkheid voldoet, zullen we zien dat de Held-Karpwaarden veel dichter bij het optimum liggen.

Toch is dit geen groot probleem – na het selecteren van een tak stijgt de Held-Karpwaarde toch snel, zodanig dat we niet veel verkeerde paden moeten overlopen.



Figuur 5: De heuristieken bieden oplossingen die dicht bij de optimale oplossingen liggen.

3. REFERENTIES

[1] G. Brinkmann, „Datastructuren en Algoritmen II," 2 september 2013. [Online]: http://twicaagt.ugent.be/~gbrinkma/DA2_2013/lesnotas.pdf. (pp. 97-103) [Geopend oktober 2013].

[2] A. J. Jones en C. L. Valenzuela, „Estimating the Held-Karp lower bound for the geometric TSP," 21 juli 2001. [Online]: http://orca.cf.ac.uk/31865/1/HeldKarp.pdf. [Geopend 17 november 2013].

[3] G. Brinkmann, „Datastructuren en Algoritmen III," 9 september 2013. [Online]: http://twicaagt.ugent.be/~gbrinkma/DA3_2013/lesnotas.pdf. (pp. 124-146) [Geopend oktober 2013].

[4] G. Reinelt, „TSPLIB," 1995. [Online]: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. [Geopend oktober 2013].