Project Algoritmen en Datastructuren III

Stefaan Vermassen

Oktober-December 2013

1 Inleiding

Voor dit project was het de bedoeling het handelsreizigersprobleem op te lossen m.b.v. een gedistribueerde versie van de *branch and bound techniek*. De heuristieken die ik gekozen heb, die worden ingezet voor het bounding gedeelte, zijn *simulated annealing*, met als basis een greedy algoritme, en *Tabu search*.

2 Implementatie

2.1 Branch and bound

Ik ben begonnen met het schrijven van een recursief algoritme waarna ik probeerde de bounding criteria zo streng mogelijk te maken.

Algoritme 1: *Branch and bound*

```
zoek(stad, gewicht, bezochte steden)
als alle steden bezocht zijn dan
    als gewicht + d(stad, s_1) < gewicht_beste_rondreis
        gewicht_beste_rondreis=gewicht+d(stad,s_1)
    eind
anders
    markeer stad als bezocht
    benedengrens = bepaal_benedengrens()
    als boven_splitsniveau || benedengrens < beste_rondreis
        voor i=2; i<=n; i++ doe
            als s_i nog niet bezocht is
                als niet_op_splitsniveau || p_id = b_nr % aantal_processen
                    zoek(s_i,gewicht+d(stad,s_i), bezochte_steden+1)
                eind
                als op_splitsniveau
                    b_nr++
                eind
            eind
        eind
    eind
    markeer stad als onbezocht
eind
```

Om het algoritme zo efficiënt mogelijk te maken, moeten we de recursieve oproep zoveel mogelijk kunnen vermijden. Nog beter is de for-lus te vermijden door er voor te zorgen dat `beste_rondreis` zo klein mogelijk is en `benedengrens` zo groot mogelijk. `beste_rondreis` is het gewicht van de beste rondreis bepaald door alle processen. Deze waarde wordt telkens geüpdated wanneer het een betere waarde ontvangt. De benedengrens wordt als volgt bepaald:

- Voor elke stad s definieer `min_door[s]` als de som van de twee goedkoopste bogen die van s vertrekken.
- `min_door[N]` = $\left\lceil \sum_{s \in N} \texttt{min\_door[s]}/2 \right\rceil$ met N de verzameling van alle onbezochte steden

`min_door[s]` is dus de goedkoopste manier om s binnen te komen en dan over een andere boog te vertrekken. Om het even hoe je door de onbezochte steden reist, de kost zal altijd ten minste `min_door[N]` zijn.

In de for-lus zelf kan ook nog gebound worden. Het is veel beter gewoon niet door te gaan als het gewicht al te groot is: Definieer `min_afstand` als de kleinste van alle afstanden tussen 2 steden. Als `gewicht+d(stad,s_i)+(n-bezochte_steden)*min_afstand` groter of gelijk is aan de best gevonden afstand tot nu toe, kunnen we de recursieve oproep in de for-lus dus gewoon vermijden.

2.2 Simulated annealing

Het *simulated annealing* algoritme is opmerkelijk effectief bij het vinden van een oplossing die dicht bij de optimale oplossing ligt voor het handelsreizigersprobleem. Het idee is dat je in het begin een vrij hoge temperatuur hebt, die je langzaam laat afkoelen terwijl het algoritme loopt. Een hoge temperatuur stelt je in staat oplossingen te accepteren die slechter zijn dan onze huidige oplossing. Hoe meer tijd verstrijkt, hoe minder hoog de temperatuur is en dus hoe minder oplossingen geaccepteerd worden. Door dit proces zal het algoritme zich geleidelijk focussen op het gebied van de zoekruimte waar een goede oplossing gevonden kan worden.

2.2.1 Startroute

Als basis voor het *simulated annealing* algoritme heb ik het resultaat van het *Nearest neighbour algorithm* gebruikt. Dit algoritme zal op een greedy manier een optimale rondreis zoeken: vanaf elke stad wordt de dichtstbijzijnde onbezochte stad gekozen. Omdat het resultaat van dit algoritme afhangt van de stad waarin gestart wordt, heb ik ervoor gekozen om elke stad te proberen als startpunt. De meest optimale rondreis zal gebruikt worden.

Algoritme 2: *Nearest neighbour algorithm*

Kies *beste oplossing* als oneindig
voor elke stad i
    1. kies i als huidige stad
    2. zoek een onbezochte stad V met minimale afstand van de huidige stad
    3. kies V als huidige stad
    4. markeer V als bezocht
    5. **als** alle steden bezocht zijn
        als deze oplossing beter is dan *beste oplossing*
            kies deze oplossing als *beste oplossing*
        eind
        ga naar volgende stad i
    eind
    6. ga naar stap 2
eind

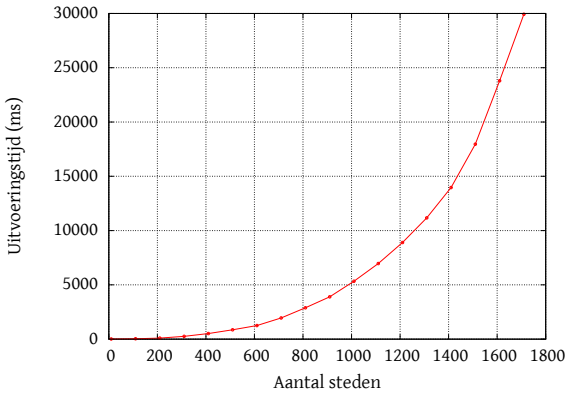
Op die manier zal het *simulated annealing* algoritme niet beginnen met een volstrekt random rondreis.

2.3 Algoritme

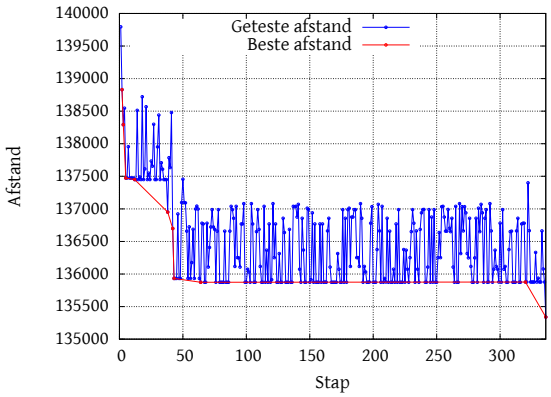
In het begin is de temperatuur zo hoog dat het elke oplossing zal accepteren. Als de gevonden oplossing beter is dan de huidige oplossing zal deze onvoorwaardelijk geaccepteerd worden. Zo niet houden we rekening met de temperatuur en hoeveel slechter de oplossing is.

Algoritme 3: *Simulated annealing*

kies de oplossing van het *Nearest neighbour algorithm* als basis
herhaal volgende stappen tot aan de stopvoorwaarde voldaan is (attempt < STATISFIED).
attempt=0
    wissel 2 random steden om in de huidige oplossing
    als we deze verandering accepteren **dan**
        attempt=0
    anders
        maak de verandering ongedaan
        attempt = attempt+1
    eind
    verlaag de temperatuur
eind



(a) Tijdscomplexiteit



(b) Geteste oplossingen door simulated annealing voor 10 steden

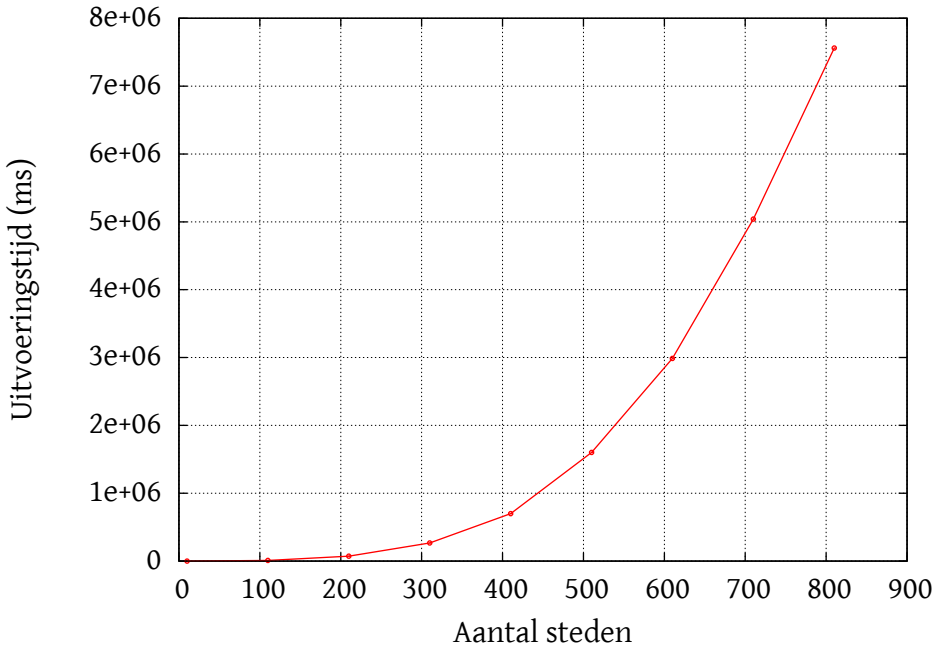
Figuur 1: Simulated annealing

2.4 Tabu search

Tabu Search heeft als voordeel tegenover basis local search algoritmen dat het zal proberen ontsnappen uit lokale minima. Hiervoor wordt gebruik gemaakt van een taboe lijst. Dit is een lijst met verwisseloperaties. Wanneer een betere oplossing gevonden wordt, zal deze verwisseloperatie taboe gemaakt worden voor een bepaald aantal iteraties, zodat we dezelfde verwisseloperatie niet onmiddellijk opnieuw kunnen toepassen.

Algoritme 4: *Tabu search*

kies een startroute
herhaal volgende stappen tot aan de stopvoorwaarde voldaan is (maximum aantal iteraties)
    verwissel 2 steden in de route
    als dit een betere oplossing is en de verwisseling is geen taboe
        accepteer de verwisseling
        maak de verwisseling taboe voor een bepaald aantal iteraties
    eind
    decrementeer tabu lijst
eind

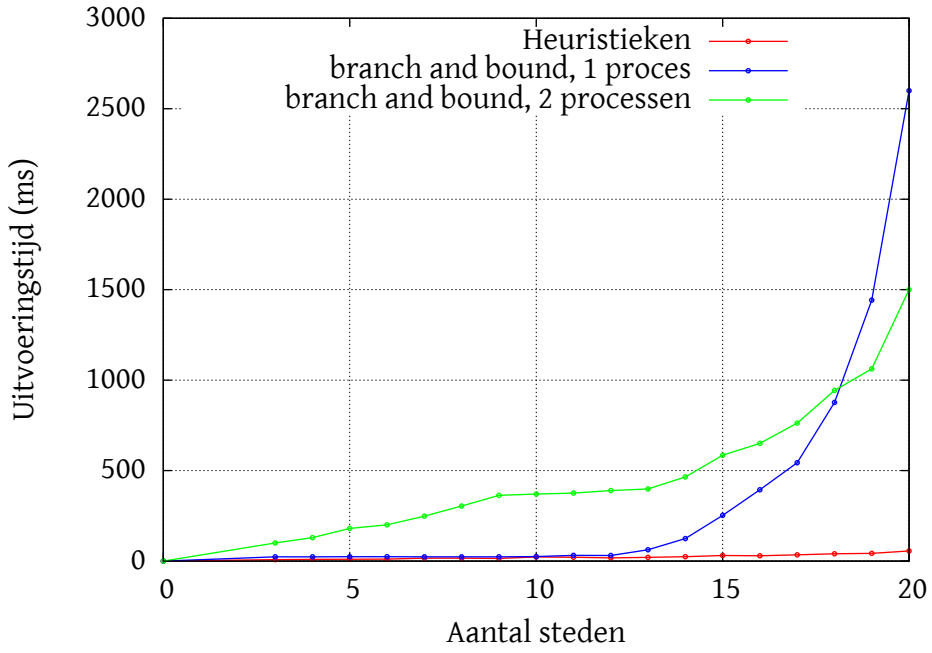


Figuur 2: Uitvoeringstijd van tabu search

Als startroute wordt de oplossing van het *simulated annealing* algoritme gebruikt.

2.5 Processenverdeling

Bij het testen van de heuristieken stelde ik vast dat zij zeer snel een oplossing vinden in vergelijking met de processen die ingezet waren voor het *branch and bound* algoritme. Dit is ook te zien op Figuur 3. Als het programma met meer dan 1 proces gestart wordt, zal er altijd 1 proces gebruikt worden voor de heuristieken, de overige processen worden ingezet voor *branch and bound*.



Figuur 3: Invloed van het aantal steden op de uitvoeringstijd

Het valt op dat meer processen niet noodzakelijk een goede invloed geven op de uitvoeringstijd, dit zal maar zo zijn vanaf een bepaald aantal steden. Dit komt door de overhead van MPI: er moet een afweging gemaakt worden tussen de kost van de communicatie en hoeveel voordeel het algoritme heeft aan deze communicatie. We zien dat 2 processen slechts voordeel bieden vanaf een 19 steden, voor alle kortere rondreizen is het efficiënter om slechts 1 proces te gebruiken.

2.6 Input/output

Het inlezen van het inputbestand vindt plaats in `matrix.h`. Dit gebeurt enkel door proces 0. Tijdens het inlezen worden een aantal handige zaken bepaald voor een zo goed mogelijke boundingcriteria:

- de kleinste afstand nodig tussen 2 steden
- Voor elke stad s definieer `min_door[s]` als de som van de twee goedkoopste bogen die van s vertrekken.

Het aantal steden, een 2D-array met de afstanden, de kleinste afstand en de array `min_door` wordt door proces 0 naar de andere processen gestuurd.

2.7 Communicatie

Tijdens het uitvoeren van het algoritme is er communicatie tussen de verschillende processen ingezet voor *branch and bound*. Wanneer een proces een route gevonden is die korter is dan de

tot nu toe kortste route, wordt de gevonden waarde doorgestuurd naar alle andere processen die bezig zijn met *branch and bound*.

Het proces dat ingezet was voor de heuristieken zal bij afloop van elke heuristiek zijn gevonden afstand sturen naar de overige processen. Dit gebeurt enkel als de oplossing beter is dan een reeds doorgestuurde oplossing in dit proces. De routes meesturen heeft geen zin, aangezien we enkel de afstand nodig hebben om strengere boundingcriteria op te stellen.

Op het einde zal elk proces dat ingezet was voor *branch and bound* zijn oplossing sturen naar proces 0. De beste oplossing wordt bepaald en wordt uitgeschreven naar standaarduitvoer.

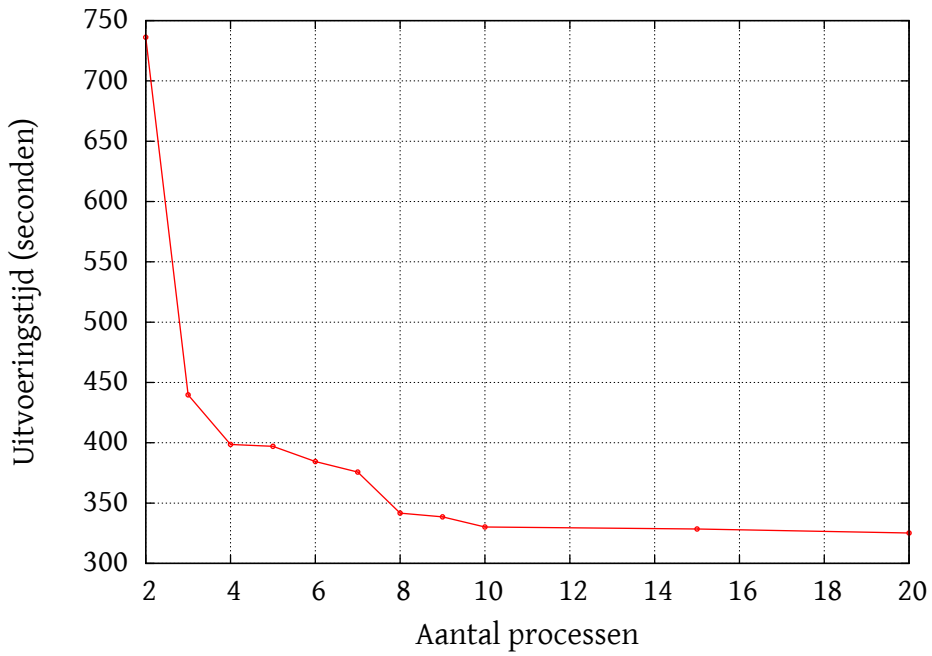
3 Correctheidstesten

Om de correctheid van het *branch and bound* algoritme te testen werd eerst en vooral hun oplossing vergeleken met de voorbeelden op Minerva. Ook heb ik gebruik gemaakt van TSPLIB.[1]. Dit is een bibliotheek met tal van TSP problemen en hun oplossing.

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

4 Performantietesten

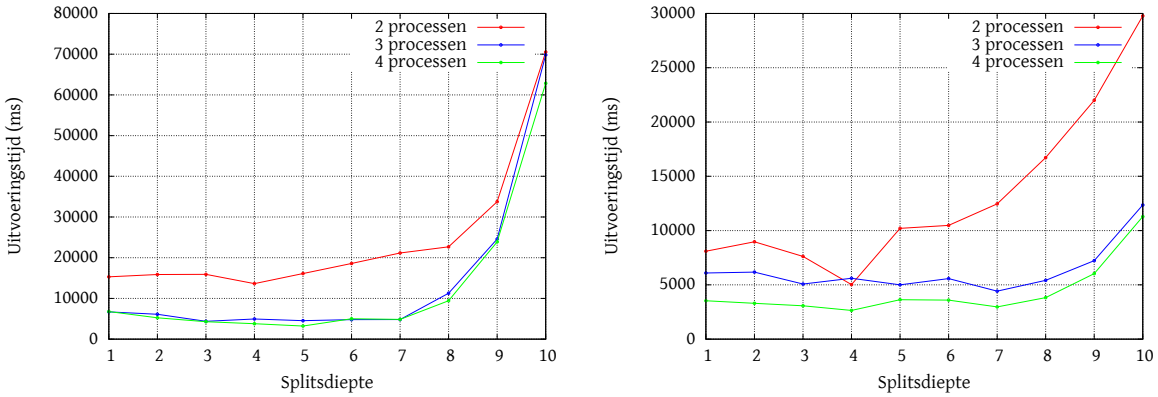
4.1 Invloed van het aantal processen



Figuur 4: Invloed van het aantal processen bij het zoeken naar een kortste rondreis tussen 26 steden

4.2 Invloed van de splitsdiepte

Op Figuur 5 zien we het resultaat van tijdsmetingen op 2 willekeurige afstandsmatrices, waarbij de splitsdiepte gevarieerd wordt.



(a) 21 steden

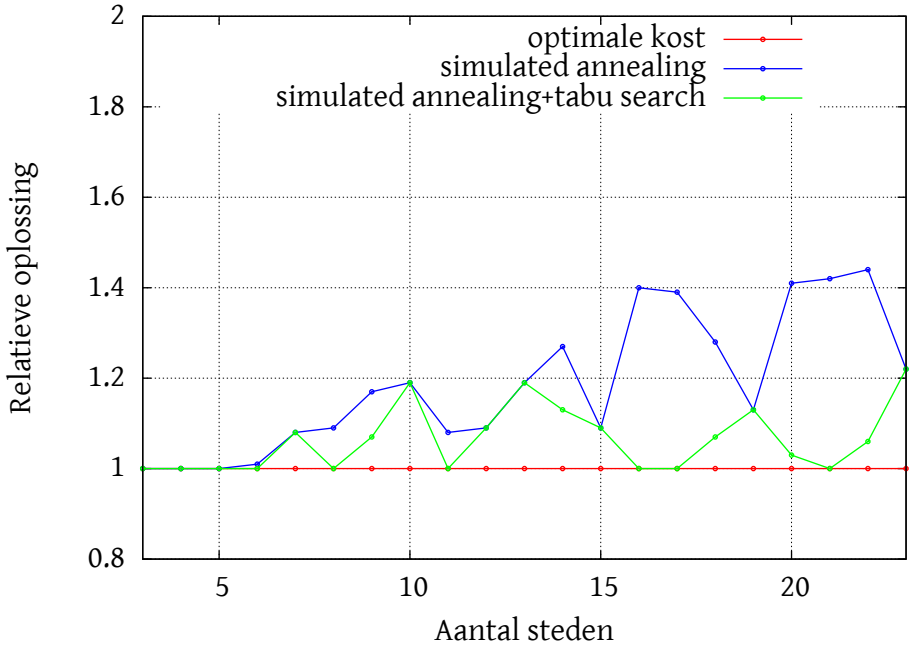
(b) 19 steden

Figuur 5: Invloed van de splitsdiepte op de uitvoeringstijd op een willekeurige afstandsmatrix

4.3 Heuristieken

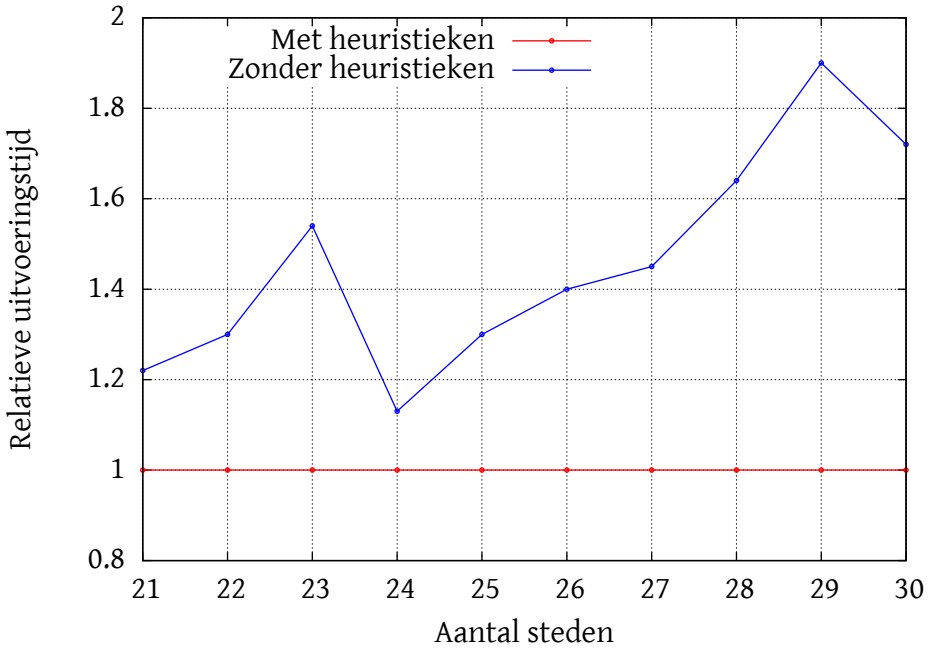
4.3.1 Oplossing heuristieken

Op Figuur 6 zien we hoeveel de gevonden oplossing door de heuristieken afwijkt van de optimale oplossing. We zijn vooral geïntesseert hoe nuttig het is om onze 2 heuristieken na elkaar uit te voeren. We zien dat na het uitvoeren van de tweede heuristiek, *tabu search*, er in veel gevallen een betere oplossing gevonden wordt. Er hangt natuurlijk veel af van de afstandsmatrix: soms vindt de eerste heuristiek al de optimale oplossing of soms kan de tweede heuristiek de oplossing van de eerste heuristiek niet verbeteren.



Figuur 6: Afwijking van de optimale oplossing

4.3.2 Invloed op de uitvoeringstijd



Figuur 7: Uitvoeren van het algoritme met en zonder bounding door heuristieken

We zien dat hoe meer steden er bezocht moeten worden, hoe meer voordeel het *branch and bound* algoritme heeft aan de heuristieken.