Practicum Optimalisering

8-4-2005

Inhoud:

Probleembeschrijving:

In dit practicum gaan we proberen zo efficiënt mogelijk routes te bepalen voor een bedrijf dat binnen een bepaalde periode met verschillende agenten een bepaald aantal adressen moet bezoeken. De kosten van overwerken, reistijden, en het eventueel niet bezoeken van adressen zijn reeds al gegeven. Aan ons de taak om middels een lokaal zoekalgoritme een programma te schrijven dat verschillende problemen van dit soort kan oplossen.
Het programma moet hiertoe een bepaalde XML-inputfile kunnen inlezen, waarin het probleem gedefinieerd is. De opmaak van deze file is ook al gegeven. Het programma moet ook een XML-outputfile leveren met daarin de oplossing. Deze file, waarvan de opmaak ook is gegeven, kan worden gebruikt om de oplossing te visualiseren en eventueel verder te verwerken.

Gebruikte techniek:

We hebben gebruik gemaakt van het zgn. Tabu-search. We gaan ervan uit dat U bekend bent met de principes van dit zoekalgoritme; we gaan hier alleen in op de uitwerking van het algoritme met betrekking tot dit probleem. Alvorens we hiermee beginnen, merken we eerst op dat het probleem in principe kan worden opgedeeld een twee deeloptimalisaties. Het is namelijk mogelijk om één route zelf te optimaliseren, dus een serie adressen die worden bezocht op een dag zo optimaal mogelijk proberen te rangschikken. Het andere optimaliseringsprobleem is dan het zo optimaal mogelijk spreiden van de te bezoeken adressen over alle dagen die in de voorgedefinieerde periode vallen. De details van het zoekalgoritme splitsen we nu dan ook in deze twee subproblemen. Eerst bekijken we echter wat structurele zaken.

We stellen het volgende:
-een oplossing bestaat uit een aantal routes en eventueel een aantal onbezochte adressen.
-een route bestaat uit een aantal adressen die sequentieel worden bezocht, op een bepaalde dag door een bepaalde agent.
-een adres kan alleen worden bezocht door een bepaalde agent.
-een agent heeft voor elke dag dat hij werkt één en precies één route.

Uit bovenstaande volgt gelijk dat het een en ander handig gesplitst kan worden. In principe is het namelijk zo dat dit optimaliseringsprobleem voor elke agent apart kan worden opgelost; een bepaald aantal adressen behoort toe tot een agent, welk al die adressen zal moeten bezoeken. Verder zien we dat elke route apart kan worden geoptimaliseerd (beste volgorde van adresbezoek bepalen); dit noemen we de dagoptimalisatie.
Ook is het zo dat elke agent een andere route heeft voor elke dag in zijn periode. Welke adressen in welke dag (en dus route) zitten, moet ook worden geoptimaliseerd. Dit noemen we de periodeoptimalisatie.

1. Dagoptimalisatie

Het eerste wat we nodig hebben in een tabu-search, is een neighbourhood. Om tot goede resultaten te komen hebben we verschillende neighbourhoods geïmplementeerd en bekeken welke het beste leek te zijn voor dit optimaliseringprobleem. De geïmplementeerde neighbourhoods zijn de standaard 1-opt en 2-opt ruimtes. Ook is er een neighbourhood die probeert beide adressen die betrokken zijn bij de grootste afstand in die route, op een andere plek in te voegen. Dit noemen we de 'worstEdgeNeighbourhood'.
Met behulp van die neighbourhood definiëren we ook de extended 1-opt en de extended 2-opt buurruimtes. Deze ruimtes zijn de standaard 1- resp. 2-opt ruimtes, aangevuld met de worstEdge-ruimte. We laten de tabu-search zeven keer itereren op elk van deze buurruimtes, en verkrijgen daaruit de volgende resultaten:

worstEdge	444898
1-opt	358229
Extended 1-opt	347071
2-opt	330798
Extended 2-opt	314305

We zien dat extended 2-opt de beste oplossingen geeft. Helaas heeft deze ook de grootste buurruimte, dus moeten we uitkijken met wat we doen; het is namelijk misschien beter een kleinere buurruimte te hebben aangezien we dan sneller kunnen itereren. We kunnen later eventueel nog op de uiteindelijke oplossing deze extended 2-opt erop loslaten.

2. Periodeoptimalisatie

Hier gebruiken we een heel wat kleinere neighbourhood. Het is per slot van rekening alleen de moeite waard om een adres uit een bepaalde dagroute te halen en bij een andere erbij te stoppen als de kosten hierdoor omlaag gaan. Dit gebeurt vrijwel alleen als we een adres uit die dagroute halen waarvoor geldt dat die op een of andere manier verantwoordelijk is voor de langste reistijd binnen die route. Als we die dagroute dus zouden representeren als een graaf, dan kiezen we uit die graaf de langste zijde, en kijken we naar de twee adressen die aan de uiteindes van die zijde staan. De neighbourhood wordt dan gedefinieerd door het verplaatsen van één van die slechte adressen naar elke andere dag.
Op deze manier krijgen we een kleine neighbourhood. Op elke oplossing in die neighbourhood passen we vervolgens dagoptimalisering toe (met een 2-opt buurruimte). Van al die daggeoptimaliseerde oplossingen bepalen we diegene met de minste kosten, welk dan het resultaat is van de periodeoptimalisatie.
Op die nieuwe oplossing passen we wederom periodeoptimalisatie toe; deze herhaling wordt uitgevoerd door een zgn. iterator. Zo'n iterator roept doorlopend bovenstaand proces aan terwijl hij bijhoudt wat de beste oplossing was die het ooit zijn tegengekomen, en wat de kosten van de voorlaatste oplossing waren. Ook wordt bijgehouden hoeveel keer er een oplossing *niet* verbeterd is. Is dit hoger dan een bepaald aantal, dan wordt er gestopt met itereren (we zijn dan immers waarschijnlijk in een 'diep' lokaal minimum waar we moeilijk uit kunnen komen).

3. De Tabu's

In dit probleem hebben we ervoor gekozen bepaalde verbindingen als mogelijk tabu in te stellen. Als we er tijdens dagoptimalisatie bijvoorbeeld achterkomen dat een verbinding tussen adres 1 en adres 2 een slecht resultaat tot gevolg heeft (ofwel: de buur waarin deze verbinding is weggehaald heeft de laagste kosten in relatie tot de andere buren), dan wordt deze verbinding toegevoegd aan de tabu-lijst *van die route*. Elke route heeft dus zijn eigen tabu-lijst. Zowel dagoptimalisatie als periodeoptimalisatie houden rekening met tabu's. Nu we weten wat een tabu lijst is, rijst de vraag gelijk hoe groot we deze lijst moeten houden. We hebben gekozen de lijst variabel te maken ten opzichte van de hoeveelheid adressen die in de route voorkomen. Hoe kleiner het aantal routes in de lijst, hoe kleiner de tabu-lijst lengte. Deze lineaire afhankelijkheid wordt gecontroleerd door (ofwel gedeeld door) een parameter 'tabuFactor', welk statisch gedefinieerd is. Deze tabufactor staat in onze gegeven oplossing op 2,5. Dat de tabulijst korter wordt naarmate er minder adressen voorkomen in de routes is gunstig, omdat anders de dag-neighbourhood erg klein wordt (waarschijnlijk vaak ook zelfs een leeg is), aangezien veel bekende mogelijkheden tabu zijn. De tabu-lijst werkt overigens via een standaard first-in first-out methode.

4. Iteratiestructuur

Zoals eerder gezegd roept de periode-iterator telkens een dagoptimalisatie aan. Aangezien we de bedoeling hebben dat de periode-iterator zeer vaak gaat itereren, volstaat het om per periodeoptimalisering voor elke dag alleen maar één iteratie te doen van de dagoptimalisatie. Naarmate er meer periode-iteraties volgen komen de dagen langzaamaan toch ook wel tegen een (lokale) optimale oplossing. Het feit dat dit heel langzaam gebeurt draagt bij aan het vinden van een goed lokaal minimum; we crashen nu niet zomaar in het eerste de beste lokale minimum. Desondanks is het niet verkeerd om, als de periode-iterator klaar is, nog even alle aparte routes in de uiteindelijke oplossing proberen te verbeteren. Hiervoor gooien we de tabu-lijsten eerst leeg, en passen daarna op elke route een extended 2-opt optimalisatie toe, die we door laten itereren totdat er driemaal geen verbetering is opgetreden. Voor het geval we in een loop terecht komen hebben we ook een maximum ingesteld aan het aantal iteraties die per route mag worden gedaan. In onze oplossing staat dit maximum op acht; wat testen insinueerde dat de optimale oplossing altijd wel eerder te vinden is dan na acht iteraties. Al met al kan het iteratieproces als volgt worden weergegeven:

-1 We verkrijgen een eerste oplossing.
-2 We roepen de periode-iterator aan, die blijft optimaliseren totdat er een bepaald aantal keren geen verbetering meer optreedt. In dat geval wordt de beste tegengekomen oplossing geretourneerd.
-2a Het eerste wat de periode-iterator doet is het verplaatsen van een 'slechtste adres' uit een bepaalde dag naar elke andere dag; dit levert de zgn. periode-neighbourhood. Dit wordt gedaan voor elke agent.
-2b Daarna wordt voor elke verzameling routes in de periodeneighbourhood een dagoptimalisatie uitgevoerd. Dit wordt dus ook gedaan voor alle routes van elke agent.
-2c De buur uit de periode-neighbourhood met de laagste kosten wordt de nieuwe oplossing. We krijgen dus een beste oplossing terug voor elke agent; deze worden samengevoegd in een gezamenlijke oplossing.

-3 Alle routes in die gezamenlijke oplossing worden nu (voor de volledigheid) nog een keer geoptimaliseerd, dit keer gebruikmakend van een grotere dag-neighbourhood als diegene die wordt gebruikt in 2b.

5. Eerste oplossing

Bij stap 1 zien we dat het construeren staan van een eerste oplossing. Dit is natuurlijk nodig omdat we alleen op een al bestaande oplossing kunnen itereren. Onze eerste oplossing wordt als volgt gegenereerd. We maken eerst per agent op elk van zijn werkdagen een lege route (dus van zijn startadres naar zijn eindadres, welke overigens overal gelijk zijn). Vervolgens gaan we alle adressen langs, en voegen elk adres (achteraan) aan de route van zijn default-agent toe. Dit gebeurt zodanig dat de adressen zo goed mogelijk over de werkdagen van de defaultagents wordt verdeeld (dus elke werkdag ongeveer evenveel adressen), en ook zodanig dat de winkel wel open moet zijn op de dag waar hij is ingedeeld. Verder houden we nu nog geen rekening met openingstijden van een adres op een dag en werktijden van de agent.

6. Unvisited adressen

We hebben nog niet rekening gehouden met het feit dat we ook kunnen besluiten een adres uit alle routes te houden. Dit moet het algoritme wel kunnen, evenals het eventueel toevoegen van unvisited adressen aan bestaande routes. Om dit efficiënt te implementeren, hebben we dit gedaan in de periode-iteratie. Aan de periode-buurruimte voegen we ook oplossingen toe waarin de slechtste oplossing (die we aanvankelijk eerst alleen bij andere routes toevoegden) unvisited wordt verklaard. Ook voegen we aan de buurruimte toe, voor elk unvisited adres, de oplossingen waar dat adres is toegevoegd aan elke route van de bijbehorende agent. Dus als een agent twee unvisited adressen heeft en vijf verschillende routes, dan worden er voor die twee unvisited adressen 2*5=10 oplossingen aan de buurruimte toegevoegd.

Als laatst geven nog even een overzichtje van de gebruikte (en mogelijk aan te passen) variabelen:

- TabuFactor (beïnvloedt de grootte van de tabu-lijst), *standaard 2,5*.
- De ‘aantal keren niet verbeterd’-variabelen (stap 2 en 3), *standaard 4 bij stap 2 en 3 bij stap 3*.
- De maximale aantal dagiteraties (in stap 3), *standaard 8*.

Implementatie:

Wij als wiskunde-studenten waren altijd al gecharmeerd van de functionele programmeertaal Haskell. Ook vanwege de onlangs vergaarde kennis over het ontleden (geleerd bij Grammatica's en Ontleden) hebben we voor deze taal gekozen. Ons programma is opgedeeld in een aantal verschillende bestanden:

Bestandsnaam	**Functie**
CalcCost.hs	Het berekenen van de kosten van een oplossing
FirstSol.hs	Het maken van een eerste oplossing
InputParser.hs	Het ontleden van het input-xml-bestand tot een geschikt datatype
OptDataAcces.hs	Aantal hulpfuncties
OptDatatypes.hs	Definitie van de datastructuur binnen ons programma
OptDaySearch.hs	Dagoptimalisatie
OptMain.hs	'Hoofdprogramma'; bevat functie main, die van de input output maakt
OptPeriodSearch.hs	PeriodeOptimalisatie
OptTabu.hs	Aantal functies voor het bijhouden van de Tabu-list, en het updaten van solutions
OutputParser.hs	Het omzetten van ons output-datatype tot een xml-file
ParseLib.hs	De standaard parser-combinator-library (uit vak G&O)
PCTableParser.hs	Het ontleden van de postcodetabel tot een geschikt datatype
XMLDocTree.hs	Functies voor de 'eerste stap' in het ontleden van het input-bestand

Om de structuur van ons programma duidelijk te maken is het misschien handig om een gedeelte van OptDatatypes.hs te behandelen:

```
data InputXMLData = …
```
… (*volgt bijna geheel de structuur van de input-xml-file*)

```
data OutputXMLData = …
```
… (*volgt bijna geheel de structuur van de output-xml-file*)

```
-- [Verzameling routes], [Adres(ID) van unvisited addresses]
data TabuSolutionData
 = TabuSolution [TabuRouteData] [Int]
-- Datum, Dag, Agent(ID), TabuEdge, edges die tabu zijn
data TabuRouteData
 = TabuRoute DateData DayData Int [TabuEdgeData] [TabuEdgeData]
-- Adres1(ID), Adres2(ID), Departure time, Arrival time, Visit time
data TabuEdgeData
 = TabuEdge Int Int TimeData TimeData

type PCTable          = [PCTableEntry]
type PCTableEntry     = (PCData, [(PCData, Int)])
```

(*hulpfuncties die bijvoorbeeld postcodes op gelijkheid kunnen testen*)

De datatypes InputXMLData en OutputXMLData zijn alleen interessant voor het ontleden aan het begin en aan het eind van het programma. Tijdens de iteraties wordt het datatype TabuSolutionData gebruikt (met zijn subdatatypes TabuRouteData en TabuEdgeData). In

FirstSol.hs wordt de ontlede input file (dus van type InputXMLData) omgezet naar een TabuSolutionData. Aan het einde wordt de best oplossing (dus van type TabuSolutionData) weer terug omgezet naar een OutputXMLData (met functies in OptMain), en vervolgens wordt daar dan een xml-bestand van gemaakt met functies in OutputParser.hs.

De structuur van het ‘werk-datatype’ TabuSolutionData zit heel simpel in elkaar. Een solution bestaat uit een aantal routes, en een lijstje van unvisited adressen. Een route heeft een aantal eigenschappen (datum, dag, agent) en bestaat uit een aantal edges (ritje tussen 2 adressen) en heeft als laatste eigenschap de edges die op dat moment ‘tabu’ zijn. Een edge bestaat simpelweg uit twee adressen, een vertrek- en een aankomsttijd.

De postcodetabel zit zo in elkaar dat snel de juiste afstand kan worden gevonden. Stel je wilt de afstand weten tussen pc1 en pc2, dan zoek je in de lijst het tupel met als eerste element pc1 op. Vervolgens zoek je in het tweede element van die tupel (is weer een lijst van tupels) de tupel met als eerste element pc2 op, en het tweede element is dan de gevraagde afstand. De tabel zit dus als een soort woordenboek in elkaar.

Nog een laatste opmerking. We gebruiken de GHC-compiler. We kwamen erachter dat Hugs niet overweg kon met de aanzienlijke hoeveelheid geheugen die ons programma tijdens het parsen van de input-file nodig heeft.

Resultaten:

Het oplossen van het probleem met bovenstaande standaardvariabelen heeft helaas meer tijd nodig dan aanvankelijk gedacht. Tijdens het schrijven van dit verslag loopt het algoritme al bijna 2,5 uur; dit is 1,5 uur langer dan verwacht. Daarom is er op een tweede computer zo vaak mogelijk geïtereerd, voordat de deadline overschreden wordt. Op die manier hebben we tenminste een redelijke beste oplossing om in te leveren. Hieronder volgt wat output over de kosten van de oplossing:

Kosten:
625465 – 1e oplossing
466475 – 1e iteratie
417115 – 3e iteratie
392112 – 5e iteratie
383891 – 7e iteratie
355765 – 9e iteratie
359160 – 11e iteratie
357662 – 13e iteratie
359475 – 14e iteratie

We zien dat de oplossingen nagenoeg altijd blijven dalen, tenminste in de eerste paar iteraties. De bijgevoegde oplossing (xml) is die van de 14e iteratie. De 13e iteratie was helaas niet opgeslagen; dit gebeurde alleen per zeven iteraties. Merk op dat de dagoptimalisatie niet is uitgevoerd bij deze oplossing, dus de uiteindelijke kosten komen zouden misschien nog iets lager kunnen liggen. Ook is er niet gebruik gemaakt van de iterator (anders zouden we geen output kunnen opslaan terwijl hij nog verder itereert). We vinden het heel jammer dat onze oorspronkelijke iteratiestructuur te lang duurt; we hadden waarschijnlijk ‘het aantal keren niet verbeterd’-variabele wat lager moeten kiezen.

Verbeteringen:

Het programma zoals het er nu staat kan wat schoner en hier en daar misschien ook wat efficiënter geschreven worden. Het was zeer interessant om te kijken wat er gebeurde bij de verschillende neighbourhoods; misschien is het goed om verder te zoeken naar betere. Vooral bij de periodeoptimalisering; we zijn niet verder gekomen dan de neighbourhood zoals die nu is geïmplementeerd, wellicht is hier een veel betere neighbourhood te bedenken. Een handig extraatje zou een mogelijkheid zijn om een outputfile in te lezen zodat we daar verder op kunnen optimaliseren met bijvoorbeeld andere parameters.
Ook zouden we het programma meer moeten testen in combinatie met verschillende parameters; op die manier kunnen we waarschijnlijk een beeld vormen over het verband tussen de looptijd van het algoritme, en de parameters van het programma.

Conclusie:

Het programmeren van een lokaal zoekalgoritme op een groot probleem zoals deze bleek moeilijker dan aanvankelijk gedacht. Tijdens het implementeren en testen kwamen we veel onverwachte en moeilijk te verbeteren fouten tegen. Desondanks is het uiteindelijk toch (naar onze mening) redelijk goed gelukt een leuk programma te leveren.
Zoals eerder gezegd hadden we meer moeten weten over de looptijd van het algoritme door middel van wat meer experimenten; nu kwamen we voor de verassing te staan dat het algoritme veel langzamer werkte dan aanvankelijk gedacht. Mogelijk komt dit omdat er steeds meer adressen unvisited worden verklaard, wat tot gevolg heeft dat te periode-buurruimte te groot werd om snel door heen te lopen. Dit zou verbeterd kunnen worden door alleen maar een klein deel van die buurruimte te doorzoeken.
Al met al was dit een uitdagend probleem, en we zijn benieuwd of we zelfs ook maar dichtbij de nu ‘in het echt’ gebruikte oplossing zijn gekomen.

Casper Zelissen & Albert-Jan Yzelman