KU LEUVEN

FACULTEIT INDUSTRIELE
INGENIEURSWETENSCHAPPEN

TECHNOLOGIECAMPUS GENT

Optimalisatietechnieken

Professor: Vanden Berghe Greet

Lennart VAN DAMME

Stiaan UYTTERSPROT
Maxim DEWEIRDT
Gilles CALLEBAUT

Academiejaar 2015 - 2016

©Copyright Lennart Van Damme

Zonder voorafgaande schriftelijke toestemming van de auteur(s) is overnemen, kopiëren, gebrui-
ken of realiseren van deze uitgave of gedeelten ervan verboden.

Inhoudsopgave

1 Inleiding **1**

1.1 Terminologie . . . 1

1.1.1 Basistermen . . . 1

1.1.2 Zoekmethoden . . . 1

1.1.3 Optima . . . 2

1.1.4 Heuristieken . . . 2

2 Wiskundige optimalisatietechnieken **4**

2.1 Lineair Programmeren . . . 4

2.1.1 Inleiding . . . 4

2.1.2 Wiskundige formulering . . . 6

2.1.3 Geometrische interpretatie . . . 7

2.1.4 Algemene formulering . . . 9

2.1.5 Gevoeligheidsanalyse . . . 11

2.2 Dynamisch Programmeren . . . 13

2.2.1 Knapzak probleem . . . 13

2.3 Integer Programming . . . 15

3 Heuristische Optimalisatie **16**

3.1 Definitie en classificatie . . . 16

3.2 Lokaal zoeken . . . 17

3.3 Ontwerp van lokale zoekmethoden . . . 18

3.3.1 Voorstelling van oplossingen . . . 18

3.3.2 Omgeving . . . 20

3.3.3 Zoekproces . . . 21

3.3.4 Lokale optima vermijden . . . 22

3.3.5 Overzicht lokaal zoeken . . . 23

3.4 Evaluatie van oplossingen . . . 23

3.4.1 Gewichten in de evalutaiefunctie . . . 23

3.4.2 Belang van de voorstelling van de oplossing . . . 23

3.4.3 Delta-evalutatie . . . 24

3.5 Samenvatting 24

4 Metaheurstieken 25

4.1 Situering optimalisatiemodellen en optimalisatiemethoden 25

4.1.1 Optimalisatiemodellen 25

4.1.2 Optimalisatiemethoden 26

4.1.3 Metaheuristieken 26

4.2 Simulated annealing 26

4.2.1 Basiscomponenten simulated annealing 28

4.2.2 Acceptatiefunctie 28

4.2.3 Koelschema 29

4.2.4 Methoden gebaseerd op simulated annealing 30

4.2.5 Samenvatting drempelalgoritmen 33

4.3 Tabu search 34

4.3.1 Kenmerken van tabu search 34

4.3.2 Kortetermijngeheugen 35

4.3.3 Aspiratievoorwaarde 35

4.3.4 Tabu-lijstlengte 35

4.3.5 Middellangetermijngeheugen 35

4.3.6 Langetermijngeheugen 35

4.3.7 Iterated Local Search 36

4.3.8 Variable neighbourhood search 36

4.3.9 Guided local search 37

4.3.10 Late acceptance strategie 38

4.3.11 Step counting hill climbing 38

4.3.12 Andere 38

5 Complexiteit 39

Lijst van figuren

1.1 Lokale optima en globaal optimum in een zoekruimte met een te maximaliseren functie 2

2.1 Voorbeeld van een voorstelling van een lineair programming probleem 4
2.2 Netwerk van oliepijpen 5
2.3 Tweedimensionale simplex 8
2.4 Driedimensionale simplex 8
2.5 Analyse van het gebied waarin c_1 optimaal blijft 11
2.6 Voorbeeld voorwerpen knapzak 14

3.1 Niet-lineaire voorstelling 19
3.2 Voorstelling TSP 20
3.3 Voorstelling personeelsrooster 20

4.1 Overzicht optimalisatiemodellen 25
4.2 Overzicht optimalisatiemethoden 26
4.3 Het proces van snel afkoelen 27
4.4 Het proces van traag afkoelen 27
4.5 Maxwell-Boltzmann distrubutie 28
4.6 Andere methoden die gebaseerd zijn op simulated annealing 30

Lijst van tabellen

2.1 Oplossing van het knapzak probleem 14
2.2 Samenstelling knapzak voor een capaciteit $k = 15$ 14

3.1 Binaire voorstelling oplossing knapzak 18
3.2 Geheeltallige voorstelling scheduling probleem 18
3.3 Floating-point voorstelling 19

4.1 Vergelijking van een fysisch systeem en een optimalisatieprobleem 27
4.2 Soorten geheugen 34

Hoofdstuk 1

Inleiding

1.1 Terminologie

In deze sectie verduidelijken we enkele termen die fundamenteel zijn.

1.1.1 Basistermen

Harde beperking Deze beperking zegt aan wat de oplossing MOET voldoen.

Zachte beperking Deze beperkingen zijn niet noodzakelijk, oplossingen moeten dus niet voldoen aan deze beperkingen. We kunnen wel zeggen dat oplossingen die voldoen aan meerdere zachte beperkingen als beter worden beschouwd dan oplossingen die niet aan deze beperkingen voldoen.

Feasible - Infeasible Als een oplossing op een optimalisatieprobleem voldoet aan alle harde beperkingen, dan zeggen we dat de oplossing feasible. Analoog kunnen we zeggen dat een oplossing infeasible is als de oplossing niet voldoet aan de harde beperkingen.

Evaluatiefuntie[1] Deze functie beschrijft in welke mate zachte beperkingen doorwegen. De kwaliteit van een oplossing, zoals hierboven beschreven, hangt af van de mate waarin aan zachte beperkingen voldaan is.

Optimalisatie Dit is het proces dat de beste oplossing zal proberen vinden uit alle beschikbare oplossingen. Hiervoor moet het probleem gemodelleerd worden in termen van een evaluatiefunctie die de kwaliteit van een oplossing bepaalt.

Complexiteit Dit verwijst naar de moeilijkheidsgraad van een optimalisatieprobleem (heeft vaak te maken met de tijdscomplexiteit van het algoritme). We gebruiken hier voor de 'grote O' notatie.

1.1.2 Zoekmethoden

Deterministisch zoeken Bij het gebruik van deze zoekmethode zal steeds dezelfde oplossing gevonden worden voor een probleem. Men is niet zeker dat de gevonden oplossing de optimale oplossing is.

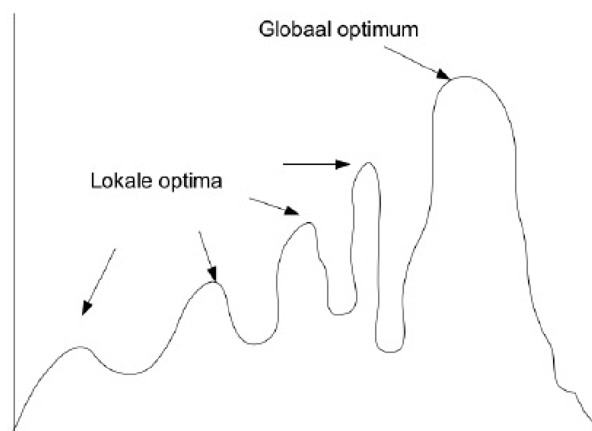
[1] Andere termen voor evaluatiefunctie: doelfunctie, objectfunctie, kostfunctie

Exhaustief zoeken Bij deze zoekmethode worden alle mogelijk oplossingen gegenereerd en uit al deze oplossingen wordt de beste gezocht. Deze zoekmethode is vaak niet gewenst aangezien het aantal oplossingen snel te groot wordt om allemaal te controleren.

1.1.3 Optima

Lokaal optimum De oplossing is een lokaal optimum als alle buren van deze oplossing een slechtere oplossing zijn. Een lokaal optimum kan een globaal optimum zijn, maar is dit niet altijd het geval. Dit is te zien in figuur 1.1.

Globaal optimum Dit is de beste oplossing die kan gevonden worden voor het probleem.



Figuur 1.1: Lokale optima en globaal optimum in een zoekruimte met een te maximaliseren functie

1.1.4 Heuristieken

Constructieve heuristieken Deze heuristieken dienen om een initiële oplossing te construeren van het begin. Voorbeelden zijn: in sport timetabling: alle wedstrijden in een wedstrijdrooster plaatsen zonder rekening te houden met de beperkingen; in timetabling: aan alle lessen een tijdstip en een lokaal toekennen.

Lokale zoekmethoden Hierbij worden de buren van een oplossing beschouwd al potentiële vervangers van de huidige oplossing. Een buur zal geselecteerd worden als nieuwe oplossing als deze een beter is dan de huidige oplossing. Er wordt dan verder gegaan met de huidige oplossing en alle buren worden weer bekeken,

Hill climbing Dit is een van de makkelijkste lokale zoekmethoden maar raakt makkelijk vast in een lokaal optimum. Volgende stappen worden ondernomen bij hill climbing.

- Genereer een oplossing in de buurt OF
- Genereer alle/enkele oplossingen in de buurt en kies de beste

Een buur wordt enkel geselecteerd als deze beter is dan de huidige oplossing (de oplossing heeft dus een betere waarde voor de evaluatie functie). Het algoritme stop wanneer geen beter oplossing in de buurt is.

Metaheuristieken Hier worden twee mogelijk definities gegeven:

- F. Glover: strategie die andere heuristieken leidt en aanpast om betere oplossingen te genereren dan deze die met lokale zoektechnieken te bereiken zijn
- I. Osman: een iteratief proces dat een onderliggende heuristiek leidt

Evolutiemethoden We redeneren op mechanismen in de natuur, met name de natuurlijke selectie. We gaan een groep oplossing beschouwen als een populatie en deze laten muteren onderling om zo beter oplossingen te krijgen. Oplossingen die slechter zijn dan nieuwe gemuteerde oplossingen worden mogelijks uit de populatie gehaald.

Hyperheuristieken Deze heuristieken zoeken naar heuristieken waarmee een oplossingen verbeterd kunnen worden.

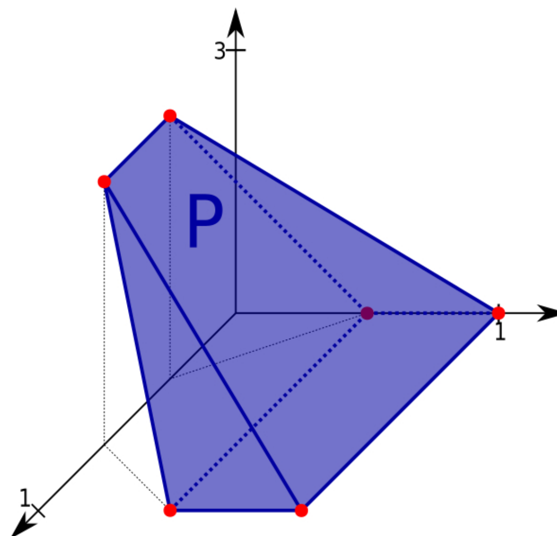
Hoofdstuk 2

Wiskundige optimalisatietechnieken

2.1 Lineair Programmeren

2.1.1 Inleiding

Bij lineaire optimalisatieproblemen is er een probleem waarbij de doelfunctie en de beperkingen voor te stellen zijn in de vorm van lineaire uitdrukkingen van de beslissingsvariabelen.
Wanneer een probleem n variabelen heeft, dan zeggen we dat dit probleem zich in een n-dimensionale ruimte bevindt. De beperkingen van het probleem stellen dan hypervlakken voor in deze n-dimensionale ruimte. Deze vlakken vormen dan de grenzen van gebied waarbinnen oplossingen 'feasible' zijn.



Figuur 2.1: Voorbeeld van een voorstelling van een lineair programming probleem

Zoals te zien is in figuur 2.1 zijn er drie beslissingsvariabelen, aangezien het probleem voorgesteld kan worden in een drie dimensionale ruimte. Het blauwe gebied stelt het gebied voor waarbinnen oplossingen 'feasible' zijn.

Voorbeeld: Oliepijpen
Gegeven een netwerk van oliepijpen (te zien in 2.2) is het de bedoeling de stroom in het netwerk te maximaliseren. We moeten rekening houden met het feit dat de hoeveelheid olie die in het netwerk gaat, ook onderaan het netwerk terug naar buiten moet komen. Olie wordt in node A binnen gepompt en komt onderaan naar buiten uit node F. Als beslissing variabele nemen we het debiet dat door een pijp kan stromen. Het mathematische model kan geschreven worden als volgt:

$$max: x_{AB} + x_{AC}$$

$$x_{AB} \leq 6 \qquad x_{CD} \leq 3$$

$$x_{AC} \leq 8 \qquad x_{CE} \leq 3$$

$$x_{BD} \leq 6 \qquad x_{DF} \leq 8$$

$$x_{BE} \leq 3 \qquad x_{EF} \leq 6$$

$$x_{BD} + x_{BE} = x_{AB}$$

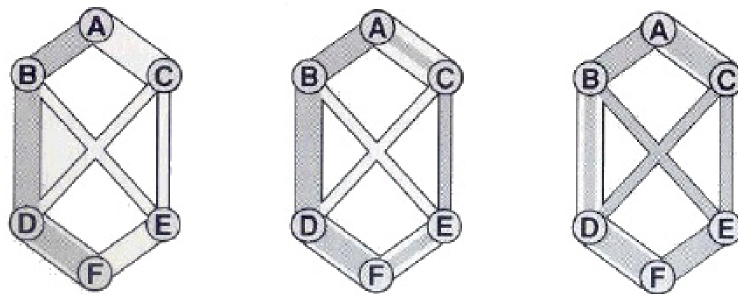
$$x_{CD} + x_{CE} = x_{AC}$$

$$x_{BD} + x_{CD} = x_{DF}$$

$$x_{BE} + x_{CE} = x_{EF}$$

$$x_{AB}, x_{AC}, x_{BD}, x_{BE}, x_{CD}, x_{CE}, x_{DF}, x_{EF} \geq 0$$

De stromen moeten groter zijn dan 0 omdat de stromen allemaal richting de F node moet gaan. Moesten we negatieve stromen toelaten, dan zou dit willen zeggen dat we stromen toelaten van beneden het netwerk richting de A node. Zo zouden we meer capaciteit kunnen creëren in het netwerk



Figuur 2.2: Netwerk van oliepijpen

Voorbeeld: T-shirt fabriek
Een kledingbedrijf maakt 3 types T-shirts.

- Type 1: 7,5 minuten snijden, 12 minuten naaien, 3 minuten verpakken, 3 € winst
- Type 2: 8 minuten snijden, 9 minuten naaien, 4 minuten verpakken, 5 € winst
- Type 3: 4 minuten snijden, 8 minuten naaien, 2 minuten verpakken, 4 € winst

Het bedrijf wil nu weten hoeveel T-shirts het van elk type moet maken om de gemaakte winst te maximaliseren. We moeten hierbij rekening houden met de volgende beperkingen.

- Er is een bestelling van 1000 T-shirts van type 1.
- Er kunnen maximaal 10.000 min gespendeerd worden aan snijden.
- Er kunnen maximaal 18.000 min gespendeerd worden aan naaien.
- Er kunnen maximaal 9.000 min gespendeerd worden aan verpakken.

2.1.2 Wiskundige formulering

We gaan hier verder met het voorbeeld van de t-shirt fabriek. De beslissingsvariabelen in dit probleem zijn van het aantal t-shirts dat van elk type moeten gemaakt worden. Deze worden voorgesteld door x_1, x_2 en x_3. Als we nu het volledige probleem mathematisch gaan voorstellen, dan krijgen we volgend model:

$$\textit{maximaliseer} \qquad 3x_1 + 5x_2 + 4x_3 \tag{2.2a}$$

$$7,5x_1 + 8x_2 + 4x_3 \leq 10.000 \tag{2.2b}$$

$$12x_1 + 9x_2 + 8x_3 \leq 18.000 \tag{2.2c}$$

$$3x_1 + 4x_2 + 2x_3 \leq 9.000 \tag{2.2d}$$

$$x_1 \geq 1.000 \tag{2.2e}$$

$$x_1\,, x_2\,, \,, x_3 \geq 0 \tag{2.2f}$$

Hierbij definieert (2.2a) de doelfunctie van het optimalisatie probleem, namelijk de winst die gemaximaliseerd moet worden. De beperking (2.2b), (2.2c) en (2.2d) leggen de beperkingen op voor het snijden, naaien en het verpakken. De bestelling van de type 1 t-shirts is te zien in beperking (2.2e), en we dienen nog te zeggen dat het aantal geproduceerde t-shirts positief moet zijn((2.2f)).

De uitdrukkingen in deze formulering zijn allemaal lineair. Het zijn lineaire combinaties van de beslissingsvariabelen vermenigvuldigd met hun kost. In het algemeen kan een LP model opstelt worden als:

$$max\ (of\ min) \quad \sum_{i=1}^{n} c_i x_i$$

$$zodanig\ dat \quad \sum_{j=1}^{m} \sum_{i=1}^{n} a_{ji} x_i \sim b_j$$

hierbij staat $\sim$ voor $\geq$,= of $\leq$

Wanneer de variabelen niet negatief zijn, dan bevind de optimale oplossing zich op een extreem punt van de oplossingsruimte. Dit bevindt zich op de rand van de beperkingen.

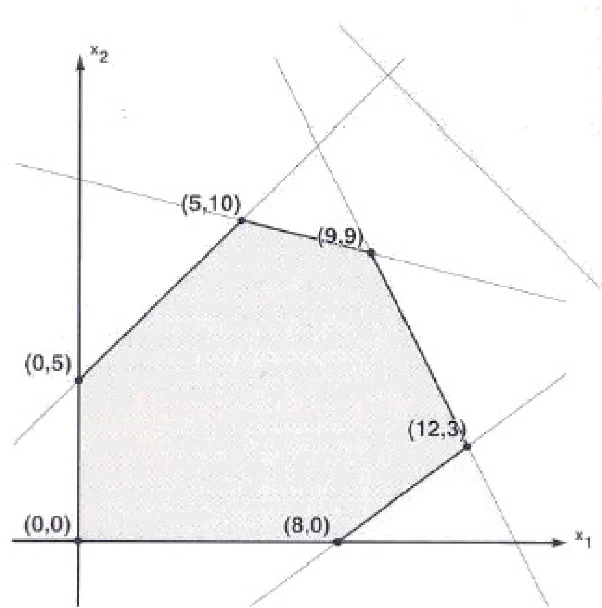
2.1.3 Geometrische interpretatie

We tonen aan dat lineaire programma's kunnen worden voorgesteld op een geometrische manier. We doen die aan de hand van een voorbeeld:

$$Maximaliseer: x_1 + x_2$$
$$-x_1 + x_2 \leq 5$$
$$x_1 + 4x_2 \leq 45,$$
$$2x_1 + x_2 \leq 27$$
$$3x_1 - 4x_2 \leq 24$$
$$x_1, x_2 \geq 0$$

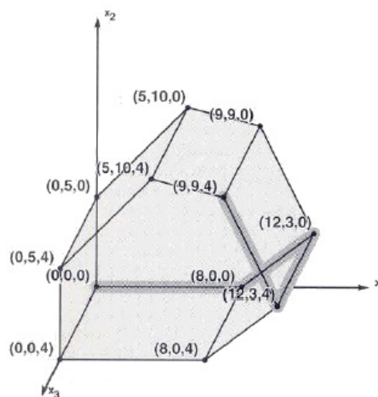
Dit voorbeeld speelt zich af in twee dimensies, omdat we twee beslissingsvariabelen hebben. We kunnen nu het volgende zeggen:

- Elke ongelijkheid definieert een half vlak waarin elke oplossing voor het lineaire programma zich moet bevinden.
- Het convexe omhulsel dat door de constraints gevormd wordt, noemen we de simplex
- De doelfunctie kan bekeken worden als een lijn met gekende helling die verschoven wordt van oneindig tot ze de simplex (het 'feasible' gebied) raakt. Dit altijd op een hoekpunt zijn van de simplex, of eventueel op een lijn van de simplex.



Figuur 2.3: Tweedimensionale simplex

Stel dat we een beslissingsvariabele x_3 toevoegen aan ons model. Stellen we dan dat $x_3 \leq 4$ en $x_3 \leq 0$, dan wordt de simplex uit 2.3 een driedimesionaal object. Dit is te zien in figuur 2.4



Figuur 2.4: Driedimensionale simplex

Wanneer er niet-lineaire functies betrokken zijn is het veel moeilijker of wanneer de ongelijkheden ook niet-lineair zijn. Als dit het geval is, dan zijn er complexe geometrische vormen die ontstaan.

Wanneer een nieuwe beperking een vlak definieert waarbinnen de simplex al ligt, dan is deze beperking redundant. Dit betekent niet dat de beperking niet belangrijk is, misschien verandert een beperking van waarde waardoor de vroegere redundante beperking niet meer redundant wordt.

De simplex kan ook open, onbegrensd zijn. Dit wil niet zeggen dat er geen oplossing gevon-

den kan worden. Het kan enkel moeilijker worden voor bepaalde algoritme om een een oplossing te vinden.

2.1.4 Algemene formulering

Om een algemene formulering te bekomen, kunnen we niet toestaan dat alle combinaties van ongelijkheden en gelijkheden voor de beperkingen. We zullen om een standaardvorm te bekomen spreken over een maximalisatie probleem. Alle beperking (behalve de 'niet-negatieve' beperkingen) dienen we uit te drukken als gelijkheden. De rechterleden $b_1, \ldots, b_m$ moeten niet-negatief zijn, alsook de beslissingsvariabelen.

2.1.4.0.1 Conversie naar algemene vorm

- Indien het een minimalisatie probleem is, dan veranderen we het teken van alle coëfficienten in de doelfunctie $(c_1, \cdot, c_n)$
- Elke variabele die niet-negatief is moet uitgedrukt worden als het verschil van twee niet-negatieve variabelen.
- Een beperking met een negatief rechterlid moet vermenigvuldigd worden met -1
- Ongelijkheden dienen weggewerkt worden door gebruikt te maken van een slack variabele (stel: $x \leq 4 \rightarrow x + s = 4$). De variabele wordt opgeteld in het geval van $\leq$, in het geval van $\geq$, wordt de variabele afgetrokken.

De algemene voorstelling is vaak in matrix vorm:

$$maximaliseer \quad CX$$
$$die voldoen aan \quad AX = b$$
$$X \geq 0$$

Hierin is $C = (c_1, \cdot, c_n)$, $b = (b_1, \cdot, b_m)^T$, $X = (x_1, \cdot, x_n)^T$ en $A = (a_{ij})_{m \times n}$
We sluiten dit deeltje af met een voorbeeld:

Voorbeeld: Giapetto's speelgoedwinkel
Een houtbewerkingsbedrijf maakt twee soorten speelgoed: soldaatjes en treintjes

1. Gegevens per soldaatje:
   - gebruikte grondstof: 10 €
   - arbeids- en werkingskosten: 14 €
   - 1 uur timmerwerk, 2 uur afwerking
   - verkoopprijs: 27 €
2. Gegevens per treintje
   - gebruikte grondstof: 9 €
   - arbeids- en werkingskosten: 10 €
   - 1 uur timmerwerk, 1 uur afwerking
   - verkoopprijs: 21 €
3. Beschikbaar
   - Grondstoffen onbeperkt
   - maximum 80 uur timmerwerk per week
   - max 100 uur afwerking per week
4. Vraag
   - max 40 soldaatjes per week
   - onbeperkte vraag naar treintjes

Het doel is om de winst te maximaliseren. We stellen nu het mathematische model voor.
Als we x_1 nemen als het aantal soldaatjes en x_2 als het aantal treintjes dan krijgen we:

$$maximaliseer: \quad 3x_1 + 2x_2 \tag{2.6a}$$

$$x_1 + x_2 \leq 80 \tag{2.6b}$$

$$x_1 + 2x_2 \leq 100 \tag{2.6c}$$

$$x_1 \leq 40 \tag{2.6d}$$

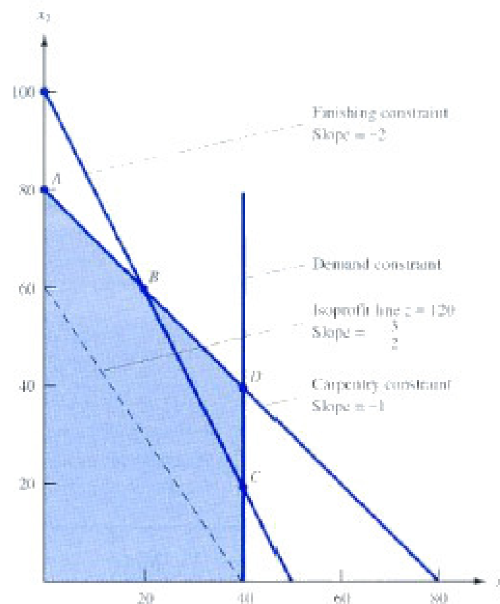
$$x_1, x_2 \geq 0 \tag{2.6e}$$

Als we dit model willen omzetten naar een algemene vorm, dan krijgen we:

$$\begin{gathered} maximaliseer: \quad 3x_1 + 2x_2 \\ x_1 + x_2 + s_1 = 80 \\ x_1 + 2x_2 + s_2 = 100 \\ x_1 + s_3 = 40 \\ x_1, x_2, s_1, s_2, s_3 \geq 0 \end{gathered}$$

2.1.5 Gevoeligheidsanalyse

Bij gevoeligheidsanalyses willen we weten op welke manier de LP parameters de optimale oplossing beïnvloeden. De optimale oplossing voor het houtbewerkingsbedrijf is een winst van 180 €, waarbij er 20 soldaatjes ($x_1 = 20$) en 60 treintjes ($x_2 = 60$) worden gemaakt. Dit komt overeen met het punt B in figuur 2.5.



Figuur 2.5: Analyse van het gebied waarin c_1 optimaal blijft

Stel nu dat de bijdrage in de winst van een soldaatje zou wijzigen, dan is het misschien optimaal om het aantal geproduceerde soldaatjes te gaan veranderen, analoog voor het aantal treintjes.
Stellen we c_1 gelijk aan de bijdrage in de winst voor elk soldaatjes, dan kunnen we gaan kijken voor welke waarden van c_1 winst optimaal blijft. Momenteel is $c_1 = 3$, want ieder soldaatje levert het bedrijf een winst op van 3 €.

We bepalen nu op welke lijn de winst constant blijft, dit noemen we de 'iso-winst' lijn.

$$3x_1 + 2x_2 = constante$$
$$x_2 = -\frac{3x_1}{2} + \frac{constante}{2}$$

We kunnen hieruit besluiten dat elke 'iso-winst' lijn een helling zal hebben gelijk aan $-3/2$. En wanneer de winst voor een soldaatje c_1 is, dan is de helling van elke 'iso-winst' lijn voor een soldaatje dan gelijk aan $-c_1/2$. De helling voor de timmerwerkbeperking voor een soldaatje is gelijk aan -1. Dus de 'iso-winst' lijnen zullen vlakker zijn dan de timmerwerkbeperking wanneer $-\frac{c_1}{2} \geq -1$ of dus wanneer $c_1 \leq 2$.
Dus wanner de winst van een soldaatjes zou dalen tot 2 € euro of minder, dan moet het bedrijf de productie van het aantal soldaatjes gaan veranderen. In dit geval wordt de nieuwe optimale oplossing van het punt A op figuur 2.5.

Stel dat de 'iso-winst' lijnen steiler zijn dan de afwerkingsbeperking, dan verandert de optimale winst van het punt B naar het punt C. Dit kunnen we zien door een analoge berekening te doen zoals hierboven, enkel met dit verschil dat de helling van de afwerkingsbeperking gelijk is aan -2. Nu kunnen ze stellen dat de huidige basis optimaal is voor de waarden $2 \leq c_1 \leq 4$ (indien alle andere parameters dezelfde blijven).

Er kunnen analoge berekeningen gedaan worden als de bewerkingstijden zouden veranderen. Ik verwijs hiervoor graag naar de slides waar een volledig bepaling gegeven wordt.

2.1.5.0.1 Schaduwprijs

De schaduwprijs gaat uitdrukken hoeveel een verandering in het rechterlid van een beperking de optimale waarde gaat beïnvloeden. In het algemeen is de schaduwprijs voor de *i*de beperking van een LP is de hoeveelheid toename of afnamen (in een max of min probleem), wanneer het rechterlid van de *i*de beperking toeneemt met 1. Dit geld enkel wanneer de verandering geen invloed heeft op de huidige basis

> **Voorbeeld**: Schaduwprijs
> Wanneer $100+\Delta$ uren afwerking beschikbaar zijn, in de veronderstelling dat de huidige basis optimaal is, dan is de LP optimale oplossing $x_1 = 20+\Delta$ en $x_2 = 60-\Delta$.
> De optimale waarde voor de winst wordt dan gegeven door:
>
> $$3x_1 + 2x_2 = 3\,(20+\Delta) + 2\,(60-\Delta) = 180+\Delta$$
>
> Dit leert ons dat een eenheidstoename van het aantal beschikbare afwerkingsuren de winst zal doen stijgen met 1 €.
>
> Voor de vraag beperking (2.6d) geldt, als het rechterlid gelijk is aan $40+\Delta$ en de huidige basis optimaal blijft, dat de optimale waarden van de beslissingsvariabelen onveranderd blijven. De optimale winst blijft onveranderd en de schaduwprijs van de vraag (2.6d) is 0 €.

Een algemene uitdrukking voor de schaduwprijs bekomen we door:

- Veronderstel dat de huidige basis optimaal blijft terwijl we het rechterlid van de i-de beperking van het LP verhogen met Δb_i (Δb_i betekent dat we het rechterlid van de i-de beperking verminderen).
- Elke eenheid waarmee het rechterlid van de i-de beperking toeneemt zal de optimale winst waarde (voor een max probleem) doen toenemen met zijn schaduwprijs.

We kunnen dan de nieuwe optimale waarde als volgt bepalen: Voor een maximalisatie probleem

$$(nieuwe\ optimale\ z-waarde) = (oude\ optimale\ z-waarde) + \left(beperking\ i's\ schaduwprijs\right).\Delta b_i$$

Voor een minimalisatie probleem

$$(nieuwe\ optimale\ z-waarde) = (oude\ optimale\ z-waarde) - \left(beperking\ i's\ schaduwprijs\right).\Delta b_i$$

Voorbeeld: Berekening schaduwprijs
Veronderstel dat er 95 uren houtbewerking beschikbaar zijn (origineel waren er 80 uren), dan is $\Delta b_2 = 15$.
We krijgen dus een nieuwe maximale winst gelijk aan $180 + 15(1) = 195$

Gevoeligheidsanalyse is belangrijk omdat in vele toepassingen de waarden van LP parameter kunnen veranderen. Als dit gebeurt, moeten we niet noodzakelijk het gehele LP terug opnieuw oplossen. Dit is zeker van belang bij problemen waarbij er een groot aantal variabelen zijn.

2.2 Dynamisch Programmeren

Bij dynamisch programmeren gaan we een groot probleem opsplitsen in meerdere kleine deelproblemen die onafhankelijk van elkaar kunnen opgelost worden. Dit resulteert in een doorgedreven gebruik van recursie.
Als we niet precies weten welke kleine deelproblemen we dienen op te lossen om tot de uiteindelijk oplossing te koemen van het gehele probleem, dan lossen we alle alle kleine deelproblemen op en bewaren we de oplossingen voor later gebruik. Dit leidt wel tot een iets andere betekenis voor 'dynamisch programmeren'. In optimalisatie techniek is de betekenis het proces van beperkingen formuleren zodanig dat de methode toepasbaar is.
We dienen nog op te merken dat het misschien niet altijd mogelijk is om kleinere problemen te combineren en tot dus tot een oplossing te komen voor het groter probleem. Of, het aantal kleinere problemen wordt onaanvaardbaar groot om verder op te lossen.

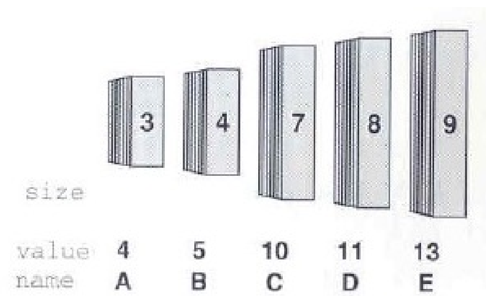
2.2.1 Knapzak probleem

In het knapzak probleem beschikken we over een zak met een bepaald capaciteit k. We willen nu voorwerpen in de zak stoppen zodanig dat de totale waarde van de voorwerpen in de zak zo groot mogelijk is. We kunnen dit probleem opdelen in deel problemen. Dit kan als volgt gezien worden:
We kunnen de grootte van onze zak telkens vergroten met 1 tot we aan de maximum capaciteit k komen, en we kunnen per eenheid volume het aantal voorwerpen dat we kiezen laten toenemen.
We kunnen dit formuleren door middel van volgend algoritme:

```
for (int j = 1; j <= N; j++) {
  for (int i = 1; i <= M; i++) {
    if(i >= size[j]){
      if(cost[i] < cost[i-size[j]]+val[j]){
        cost[i] = cost[i-size[j]]+val[j];
        best[i] = j;
      }
    }
  }
}
```

Enige duiding bij het algoritme:

- cost[i] is de hoogste waarde die met de knapzakcapaciteit i verkregen kan worder.
- best[i] is het laatste toegevoegde object om dat maximum te bekomen.



Figuur 2.6: Voorbeeld voorwerpen knapzak

Als we dit doen, dan krijgen we tabel 2.1 voor de voorwerpen gegeven in figuur 2.6

k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$j = 1$															
cost[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
best[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
$j = 2$															
cost[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
best[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
$j = 3$															
cost[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
best[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
$j = 4$															
cost[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
best[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
$j = 5$															
cost[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
best[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Tabel 2.1: Oplossing van het knapzak probleem

We kunnen de samenstelling van de knapzak bepalen door voor een bepaalt capaciteit k door in de rij best[k] te kijken. Nemen we bijvoorbeeld een capaciteit $k = 15$, dan zijn de samenstellingen te zien in tabel 2.2.

Aantal voorwepen	Samenstelling knapzak	totale waarde
$j = 1$	A-A-A-A-A	20
$j = 2$	A-A-A-A-A	20
$j = 3$	A-A-A-A-A	20
$j = 4$	D-C	21
$j = 5$	D-C	21

Tabel 2.2: Samenstelling knapzak voor een capaciteit $k = 15$

Om de samenstelling te bepalen moeten we kijken wat het beste voorwerp is bij die capaciteit. Eenmaal het voorwerp bepaalt te hebben, moeten we het volume van dat voorwerp van de capaciteit aftrekken. Zo bekomen we het volgende beste voorwerp dat we bij die capaciteit in de knapzak kunnen steken. We herhalen dit process tot we geen capaciteit meer over hebben om voorwerpen uit de knapzak te halen. Door deze methode toe te passen is het duidelijk dat het gehele knapzak probleem opgesplitst werd kleinere deelproblemen.

2.2.1.1 Eigenschappen

Eigenschap 1 De tijd nodig voor het oplossen van het knapzakprobleem met dynamisch programmeren is evenredig met NM (N: soorten objecten; M grootte van de knapzak).

- Het probleem is dus eenvoudig op te lossen als M niet groot is
- Als M of de groottes van de objecten reële getallen zijn i.p.v. gehele getallen, dan werkt dynamisch programmeren niet (fundamenteel probleem)
- voordeel: vroeger opgeloste deelprolemen moeten niet opnieuw bekeken worden

Eigenschap 2 Dynamisch programmeren leidt tot een optimale oplossing

2.3 Integer Programming

Voor de onderwerpen branch-and-bound en integer programmingin verwijs ik graag naar het artikel op het einde van deze samenvatting.

Hoofdstuk 3

Heuristische Optimalisatie

3.1 Definitie en classificatie

Een heuristiek is een informele methode die toepasbaar is op een complexe problemen en dienen om een 'voldoende goede' oplossingen te bekomen. Een heuristiek gaan we gebruiken wanneer de problemen slecht gestructureerd kunnen worden (hiermee bedoelen we moeilijk wiskundig te modelleren), of wanneer de problemen zo groot worden dat wiskundige solvers geen snelle oplossing kunnen leveren. Ze berusten meestal op vuistregels, methodes die logisch zijn voor de mens. Aan heuristieken zijn natuurlijke enkele nadelen verbonden. Hieronder zijn er enkele te vinden:

- Geen garantie voor het vinden van optimale oplossingen.
- Vaak enkel bruikbaar om de problemen waarvoor ze ontwikkeld zijn op te lossen.
- Leiden mogelijks tot een slechte oplossing

Algemene definitie:

Definitie *Heuriskein:* de kunst om nieuwe strategieën (regels) te vinden voor het oplossen van problemen.

Binnen de heuristieken kunnen we twee categorieën onderscheiden:

1. Constructieve heuristieken: Deze heuristieken bouwen een oplossing voor een probleem.
2. Pertubatieve heuristieken: De heuristieken gaan wijzigingen aan brengen binnen een oplossingen om zo een mogelijk betere oplossing te bekomen.

Enkele voorbeelden van heuristieken:

Bin packing: plaat x voorwerpen in zo weinig mogelijk containers.

- Constructieve heuristiek: neem het grootste voorwerp uit de lijst van voorwerpen, plaats het in de eerste container waarin voldoende plaats is, ga zo verder tot alle voorwerpen geplaatst zijn.
- Pertubatieve heuristiek: verwissel twee voorwerpen van container.

Scheduling: voer x taken uit door ze toe te kennen aan daarvoor geschikte machines, elke taak heeft een duur en een *due date*.

- Constructieve heuristiek: plan de taak met de vroegste due date op de machine die het vroegst beschikbaar is, herhaal tot alle taken gepland zijn.
- Pertubatieve heuristiek: verwissel twee taken op dezelfde machine, verwissel de machines van twee taken die op verschillende machines gepland zijn.

3.2 Lokaal zoeken

Dit soort heuristiek wordt gebruikt om problemen op te lossen waarvoor de exacte mathematische algoritmen te traag werken. Het voordeel van lokaal zoeken is dat we binnen een aanvaardbare rekentijd een oplossing bekomen, maar het nadeel is wel dat we geen garantie hebben dat de oplossing optimaal is.

Voor we overstappen op het algemene algoritme voor lokaal zoeken, geven we nog twee termen die vaak aan bod komen:

- *Oplossingenruimte (zoekruimte):* Verzameling van alle mogelijke oplossingen van het probleem. Binnen de zoekruimte bevinden zich ook de infeasible oplossingen voor het probleem.
- *Neighbourhood of omgeving $N(S_k)$ van een oplossing S_k:* Dit is een deelverzameling van de oplossingenruimte met oplossingen bekomen door bepaalde *moves* toe te passen op de huidige oplossing.

Algorithm 1 Algoritme lokaal zoeken

1: **STAP 1:** Initialiseren
2: $k = 0$
3: Selecteer een staroplossing $S_0 \in S$
4: Bewaar de best gekende oplossing $S_{best} = S_0$ en de waarde $best_{cost} = F(S_{best})$
5: **STAP 2:** Kiezen en vervangen
6: Selecteer een oplossing $S * k + 1 \in N(S_k)$
7: **if** $N(S_k)$ bevat geen enkel element dat aan de selectiecriteria voldoet **then**
8:     **STOP**
9: **if** $F(S_{k+1}) < best_{cost}$ **then**
10:     $S_{best} = S_{k+1}, best_{cost} = F(S_{k+1})$
11: **STAP 3:** Stoppen
12: **if** Voldaan aan de stopvoorwaarden **then**
13:     **STOP**
14: **else**
15:     $k = k + 1$, ga naar **STAP 2**

3.3 Ontwerp van lokale zoekmethoden

3.3.1 Voorstelling van oplossingen

Om een oplossing te modelleren dient men altijd de volgende zaken in het achterhoofd te houden om een zo goed mogelijk model te bekomen:

- Zo eenvoudig mogelijke voorstelling
- De manipulatie van de voorstelling moet zo snel en zo eenvoudig mogelijk kunnen
- Het evalueren van de oplossing (feasible of niet) moet zo snel mogelijk kunnen gebeuren.

3.3.1.1 Lineaire representatie

3.3.1.1.1 Binaire codering

Voor het knapzak probleem met n objecten kunnen ze onze oplossing als volgt voorstellen:

$$s_i = \begin{cases} 1 & \text{indien object } i \text{ in de knapzak zit} \\ 0 & \text{indien niet} \end{cases}$$

0	1	0	0	1	1	1	0	0	0

Tabel 3.1: Binaire voorstelling oplossing knapzak

3.3.1.2 Geheeltallige codering

Voor het schedulingprobleem met n objecten wordt de beslissingsvariabele:

$$s_i = j \text{ als persoon } i \text{ toegekend is aan taak } j$$

2	1	7	5	9	4	3	8	6

Tabel 3.2: Geheeltallige voorstelling scheduling probleem

3.3.1.3 Permutatie

Hierbij is de volgorde in de voorstelling belangrijk. Zo kunnen ze we de voorstelling van de oplossing in tabel 3.2 opvatten als de volgorde van taken die gedaan moeten worden. Deze voorstelling kan gebruikt worden bij bevoorbeeld:

- TSP.
- Volgordeproblemen: volgorde bewerkingen van CNC machine.
- schedulingproblemen.

3.3.1.4 Floating-point voorstelling

Deze voorstelling wordt gebruikt bij continue optimalisatie problemen. Voorbeelden:

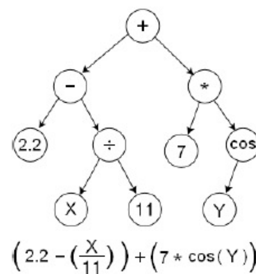
- Samenstelling bepalen van veevoeders.
- Parameteroptimalisatie in robotica.

2.5	1.3	7.7	5.1	9.6	4.8	3.8	8.4	6.2

Tabel 3.3: Floating-point voorstelling

3.3.1.5 Niet-lineaire voorstelling

Deze voorstelling van oplossingen is vooral gebaseerd op grafen.



Figuur 3.1: Niet-lineaire voorstelling

3.3.1.6 Directe ↔ indirecte voorstelling

Voorbeeld van deze voorstelling is job scheduling:

Directe voorstelling Voorbeeld hiervan:

- M_1 $(J_{11},0,3)$, $(J_{13},3,6)$, $(J_{14},6,8)$, $(J_{15},8,12)$
- M_2 $(J_{22},0,5)$, $(J_{21},5,6)$, $(J_{23},6,9)$
- M_3 $(J_{33},0,3)$, $(J_{32},5,8)$

Deze voorstelling geeft de sequentie van jobs op elke machine: ((1,3,4,5),(2,1,3),(3,2)).

Indirecte voorstelling Voor deze voorstelling is een decoder nodig. Voorbeelden:

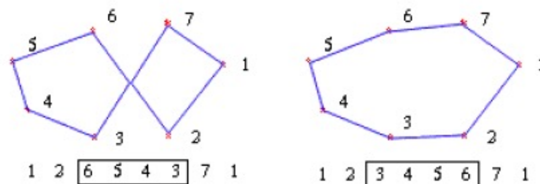
- Jobgebaseerde voorstelling, bv. (1,2,3)
  Alle opreatis van de eerste job in de lijst worden eerst gepland; daarna de operaties van de tweede job in de lijst enz.
- Operatiegebaseerde voorstelling
  Een schedule bevat een opeenvolging van operaties, elke operatie verwijst naar de machien waarom het deel van de job moet uitgevoerd worden.
  Voorbeeld: J_1: M_1,M_2; J_2: M_2,M_3; J_3: M_3,M_1,M_2,M_2; J_4: M_1; J_5: M_1

3.3.1.7 Eigenschappen van een voorstelling

Een voorstelling dient te volden aan volgende eigenschappen:

- Volledigheid: Alle mogelijke oplossingen van een probkleem zijn voor te stellen.
- Connectiviteit: Er bestaat een ’zoekpad’ tussen elke twee oplossingen in de zoekruimte.
- Efficiëntie: De voorstelling laat toe eenvoudige wijzigingen aan te brengen door zoekoperatoren.

3.3.1.8 Enkele voorbeelden van een oplossingsvoorstelling



Figuur 3.2: Voorstelling TSP

	Mo	Tu	We	Th	Fr	Sa	Su
P1	E	E	L	L	N		
P2	N		N	L	L		
P3	E	E	E	E	E	E	E
P4	E		L	N	N	N	
P5	EL	L	L	L			

Figuur 3.3: Voorstelling personeelsrooster

3.3.2 Omgeving

Definitie *Omgeving:* De omgeving (neighbourhood) van een oplossing s is een verzameling oplossingen die bereikt kunnen worden vanuit s met eenvoudige operatoren (move operatoren).

De volgende subsecties beschrijven voorbeelden van omgevingen.

3.3.2.1 Binaire voorstelling

Omgeving (Hamming): flip 1 bit van de oplossing

$$1000110000 \rightarrow \mathbf{0}000110000$$

De Hamming afstand tussen 2 strings van gelijke lengte is het aantal posities waarop de overeenkomstige bits verschillen. De grootte van de omgeving: n voor strings van lengte n.

3.3.2.2 Geheeltallige voorstelling

Omgeving: een discrete waarde wordt vervangen door een andere waarde

$$57644324 \rightarrow 576\mathbf{5}43284$$

De grootte van de omgeving wordt gegeven door k het aantal mogelijke waarden. Dan is de grootte van de omgeving gelijk aan $(k-1).n$

3.3.2.3 Permutatie

Paarsgewijze verwisseling van buren Grootte van de omgeving: $n-1$ voor een permutatie van lengte n.

$$51432 \rightarrow \mathbf{15}432$$

Invoegoperator Neem een willekeurig element van de permutatie en voeg het in op een andere positie.

$$51432 \rightarrow 14\mathbf{5}32$$

Grootte van de omgeving: $n-1$ voor een permutatie van lengte n.

Uitwisseling verwissel twee willekeurige elementen.

$$51432 \rightarrow \mathbf{3}14\mathbf{5}2$$

inversie-operator Selecteer twee willekeurige elementen en inverteer de sequentie ertussen

$$51432 \rightarrow \mathbf{5}341\,\mathbf{2}$$

3.3.3 Zoekproces

3.3.3.1 STAP 1: initiële oplossing

Hoe bepalen we de initiële oplossing waarvan vertrokken zal worden?

- Vertrekken van een gekende oplossing, bijvoorbeeld een manuele constructie.
- Een oplossing bekomen na het toepassen van een eenvoudige heuristiek of door gebruik te maken van een aantale eenvoudige vuistregels.
- Willekeurig

3.3.3.2 STAP 2: Kies en vervang

Vervang de huidige oplossing door een oplossing uit de omgeving. Welke?

Tournament factor Indien de omgeving uit veel elementen bestaat dan kunnen we een deelverzameling van die omgeving uitkommen. Het voordeel hierbij is dat het sneller is dan telkens alle elementen te evalueren.

Steepest descent (voor een minimalisatie probleem: steepest descent; voor een maximalisatie probleem: hill climbing)

Definitie *Steepest descent:* Vervang de huidige oplossing door de beste oplossing uit de omgaveing, STOP wanneer er in de omgeving geen oplossing is dan de huidige.

Deze methode heeft als voordeel dat ze snel convergeert en stop in een lokaal optimum. De oplossing die gevonden is heeft dus een grotere kans dat het een goed oplossing is. Maar het feit dat dit algoritme stop in een lokaal optimum is tevens ook een nadeel. We kunnen niet uit dit optimum geraken doormiddel van dit algoritme en zijn dus in het gewisse van eventuele betere oplossingen.

- Eerste verbetering
- Willekeurige selectie
- Gebaseerd op kennis

3.3.3.3 Intensificatie en diversificatie

Deze termen komen uit de metaheuristieken. Dit wordt verder nog besproken in hoofdstuk 4 op pagina 25

- Intensificatie: Exploiteer de beste gevonde oplossingen.
- Diversificatie: Exploreer de zoekruimte.

3.3.4 Lokale optima vermijden

3.3.4.1 Voorbeelden van algoritmen (metaheurstieken)

Itereer met verschillende oplossingen
- Multistart lokaal zoeken: herhaal het algoritme voor verschillende oplossingen
- iteratief lokaal zoeken
- GRASP

Verander het landschap van de zoekruimte Wijzig de waarde van de doelfuctie of gebruik veschillende omgevingen: *variable neighbourhood search*

Accepteer niet-verbeterende oplossingen
- Probabilistisch: *simulated annealing*
- Deterministisch: *tabu seach*

3.3.5 Overzicht lokaal zoeken

3.3.5.1 Stopcriterium

Absolute rekentijd Dit stopcriterium is eenvoudig en makkelijk uit te leggen aan de gebruiker. Het heeft echter wel als nadeel dat de oplossin niet noodzakelijk lokaal optimaal is en de rekentijd is niet gerelateerd aan de probleemkaraktristieken.

Tijd na de laatste verbetering Dit criterium is ook eenvoudig en we zijn zeker dat de oplossing zich in een lokaal optimum bevindt. Het is echter niet eenvoudig om uit te leggen aan de gebruiker en de oplossing is niet noodzakelijk het beste lokale optima.

Aantal iteraties na de laatste verbetering We zijn hierbij we zeker dat we een lokaal optima hebben en we zijn in staat om het aantal iteraties te relateren aan de probleemgrootte. Maar ook hier is dit moeilijk uit te leggen aan de gebruiker.

3.4 Evaluatie van oplossingen

De evalutaiefunctie zal de kost gaan bepalen van een bepaalde oplossing. Maar hoe stellen we deze functie op? We kunnen gaan gebruik maken van een gewogen som van overtredingen van beperkingen, maar welke gewichten geven we dan aan de termen van de som? Het kan ook mogelijk zijn om een andere combinatie te maken van kwaliteitsmaten. Hiervoor kijken we naar een stock cutting probleem:
Snijd een aantal bestellingen van profielen voor ramen en deuren uit basisprofielen met een handelslengte. De bedoeling is om zo weingi mogelijk basisprofielen te moeten bestellen.
Enkele mogelijkheden voor de doelfunctie:

- Som van de reststukken
- Som van de kwadraten van de reststukken (werkt veel beter!)

Voor het voorbeeld van de Eternity II puzzel verwijs ik graag naar de presentaties.

3.4.1 Gewichten in de evalutaiefunctie

Het is belangrijk om het relatief gewicht van overtredingen te bepalen om zo een correct gewicht te kunnen bekomen. Hoe gaan we dit doen? Geen concrete wetenschap hierrond, maar het is handig om een expert te raadplegen. Een expert in de materie waarrond de optimalisatie moet gebeuren wel te verstaan.
Het kan zijn dat we te maken krijgen met eenheden die we niet met elkaar kunnen optellen, zoals bijvoorbeeld het aantal gereden kilometers en het aantal vrachtwagens.

3.4.2 Belang van de voorstelling van de oplossing

De evaluatiefunctie kan vaak opgeroepen worden om oplossingen te evalueren. Hoe sneller we via de voorstelling van de oplossing tot een kost kunnen komen hoe beter. Het moet dus mogelijk worden om zo snel en eevnvoudig mogelijk harde en zachte beperkingen te controleren.

3.4.3 Delta-evalutatie

Wanneer een oplossing wijzigt, da is het vaak niet nodig om de volledige evaluatiefuctie opnieuw aan te roepen, er zal slechts een deel van de kost verandert zijn. Er dient dus een evenwicht gezocht te worden in het opslaan van informatie van de huidige oplossing en het opnieuw berekenen. Enkele voorbeelden:

- Personeelsroosterplanning: Enkel de persoonlijke roosters opnieuw berekenen waaraan iets gewijzigd is; opletten met verbanden zoals samenwerken, balanceren van de werkdruk, enz.
- Traveling tournament probleem: amper delta-evalutatie mogelijk want bijna elke kleine wijziging verandert de opeenvolging van wedstrijden.
- Timetabling evaluatie voor studenten: Klasgroepen, lesgevers, lokalen niet opnieuw uitvoeren indien er niets aan dat rooster gewijzigd is.

3.5 Samenvatting

Modellering

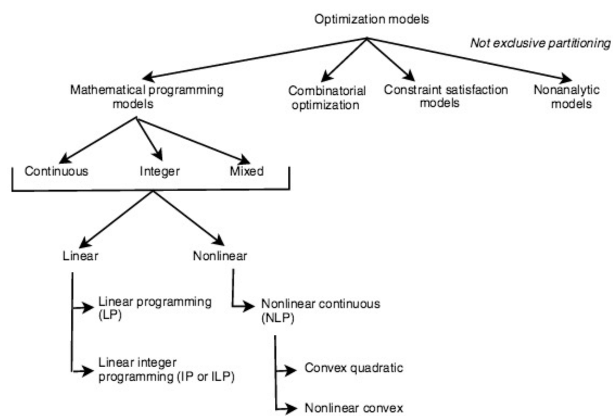
- Voorstellen van een oplossing.
- Ontwerp van lokale zoekmethoden.
- Ontwerp van een evalutatiefunctie.
- Vaak moeilijker dan ontwikkeling van een lokaal zoekalgoritme.
- Vaak van meer belang voor de oplossingskwaliteit van een optimalisatietechniek dan het algoritme zelf.

Hoofdstuk 4

Metaheurstieken

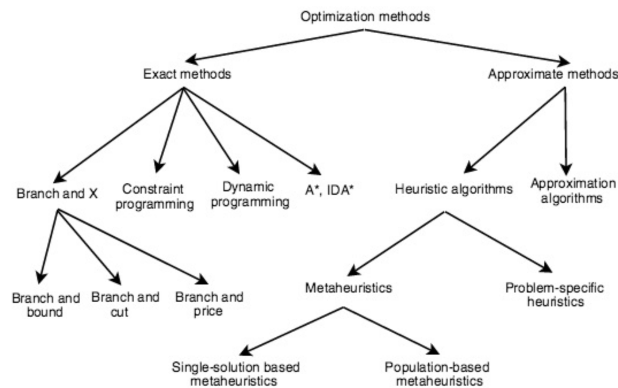
4.1 Situering optimalisatiemodellen en optimalisatiemethoden

4.1.1 Optimalisatiemodellen



Figuur 4.1: Overzicht optimalisatiemodellen

4.1.2 Optimalisatiemethoden



Figuur 4.2: Overzicht optimalisatiemethoden

4.1.3 Metaheuristieken

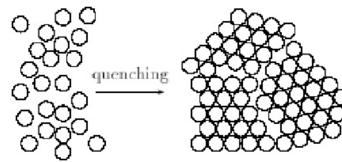
Definitie Een **metaheuristiek** is een methode die een probleem optimaliseert door iteratief te proberen een kandidaatoplossing te vebetern met betrekking tot een gegeven kwalisteitsmaat. Metaheuristieken zijn niet probleemspecifiek. Ze zijn in staat om grote ruimten met kandidaat-oplossingen te doorzoeken, zonder garantie op het vinden van een optimale oplossing.

4.2 Simulated annealing

- Origineel bedacht door Kirkpatrick, Gelatt, Vecchi in 1983
- Het is een drempelalgoritme
- Gebaseerd op het proces van koelen en kristallisatie

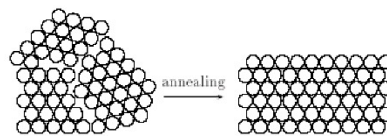
Wanneer een gesmolten materiaal gekoeld wordtm dan neemt de bewegingsgraad van de moleculen af. De structuur van de resulterende vaste stof hangt af van de snelheid waarmee de temperatuur daalt. Snel koelen veroorzaakt het ontstaan van vele kiemen vaste stof waartussen heel wat spanningen blijven bestaan. Bij traag koelen, uitgloeien of *annealing*, probeert men tot een zo zuiver mogelijk kristal te komen, met een minimum aan inwendige spanningen.

Het proces van snel afkoelen is visueel voorgesteld in figuur 4.3 op de pagina hierna. Het resultaat van snel afkoelen is een polykristal



Figuur 4.3: Het proces van snel afkoelen

Bij traag koelen of *annealing* kunnen we, wanneer we oneindig traag afkoelen krijgen we een monokristal. Dit is voorgesteld in figuur 4.4.



Figuur 4.4: Het proces van traag afkoelen

Fysische toestand vs. oplossing

Opeenvolgende oplossingen van een combinatorische optimalisatieprobleem kunnen we voorstellen op basis van overeenkomsten met een fysisch systeem met verschillende deeltjes.

- Oplossingen van het optimalisatieprobleem zijn equivalent aan toestanden van het fysische systeem.
- De kwaliteit van een oplossing is omgekeerd evenredig met de inwendige energie van een toestand.

Hierbij is de controleparameter de temperatuur (T).

Fysische toestand vs. optimalisatieprobleem

De vergelijking tussen een fysisch systeem en een optimalisatieprobleem is te zien in tabel 4.1.

Fysische systeem	Optimalisatieprobleem
Systeemstatus	Oplossing
Moleculaire posities	Beslissingsvariabelen
Energie	Doelfunctie
Toestand van laagste energie	Globale optimale oplossing
Metastabiele toestand	Lokaal optimum
Afschrikken	Lokaal zoeken
Temperatuur	Controleparameter T
Uitgloeien	Simulated annealing

Tabel 4.1: Vergelijking van een fysisch systeem en een optimalisatieprobleem

Algorithm 2 Simulated annealing

1: **Input:** Koelschema
2: $s = s_0$ ▷ Genereer een initiële oplossing
3: $T = T_{max}$ ▷ Starttemperatuur
4: **repeat**
5: **repeat**
6: Genereer een random *neighbour* $s' \in N(s)$ ▷ Herhaal op een vaste temperatuur
7: $\Delta E = f(s') - f(s)$
8: **if** $\Delta E \leq 0$ **then**
9: $s = s'$
10: **else**
11: Accepteer s' met een probabiliteit $e^{\frac{-\Delta E}{T}}$
12: **until** Evenwichtsvoorwaarde bereikt ▷ bv. aantal iteraties uitgevoerd op T
13: $T = g(T)$
14: **until** Voldoen aan stopvoorwaarde ▷ bv. $T < T_{min}$
15: **Output:** Beste gevonden oplossing

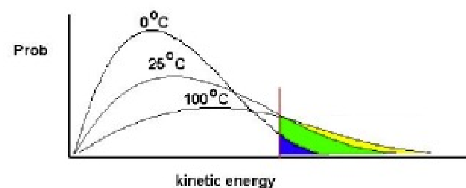
4.2.1 Basiscomponenten simulated annealing

- Acceptatiefunctie: probabiliteitsfunctie die toelaat dat *simulated annealing* niet-verbeterende oplossingen aanvaardt
- Koelschema: definieert de temperatuur in elke stap van het algoritme

Beide componenten zijn essentieel en dienen goed fijngestelde te worden om een efficiënt en effectief algoritme te ontwikkelen.

4.2.2 Acceptatiefunctie

De waarschijnlijkheid om naar een toestand van hogere energie te gaan, neemt af met de temperatuur. Dit wordt gegeven door de Maxwell-Boltzmann distributie. Deze distributie is te zien in figuur 4.5



Figuur 4.5: Maxwell-Boltzmann distrubutie

In simulated annealing is dit de waarschijnlijkheid om naar een oplossing met slechtere kwaliteit te gaan. Deze neemt af met de T parameter, met R een uniform random getal tussen 0 en 1. De

waarschijnlijkheid wordt dan gegeven door volgende formule:

$$P(\Delta E, T) = \exp\left(\frac{-\Delta E}{kT}\right) > R$$

4.2.3 Koelschema

Voor het het koelschema moeten we een aantal parameters bepalen:

- Initiële temperatuur
- Evenwwichtstoestand
- Koeling
- Stopconditie

4.2.3.1 Initiële temperatuur

We kunnen deze te hoog nemen dan leidt dit tot random zoeken, alle zetten worden geaccepteerd. Maar dit leidt tot hoge computationele kost. Het andere extreem is de initiële temperatuur zeer laag nemen. Dit leidt tot *first improvement*.
Het probleem bij de iniële temperatuur is het zoeken tussen beide extremen.

Er zijn een paar belangrijke strategieën die we kunnen toepassen:

- Een hoge start temperatuur opdat bij het begin van het zoeken alle oplossingen aanvaard zouden worden
- *Acceptatieafwijking*: bv. starttemperatuur $= k\delta$, na initiële experimenten waarbij δ de standaarddeviatie voorstelt van het verschil tussen de waarden van de doelfunctie en $k = -\frac{3}{ln(p)}$, met acceptatieprobabiliteit p groter dan 3δ.
- *Acceptatieratio:* de starttemperatuur bepalen zodanig dat de appectatieratio van oplossingen groter is dan een voorgedefinieerde waarde a_0 (ook te bepalen na initeële experimenten).

4.2.3.2 Evenwichtstoestand

Om op elke temperatuur een evenwicht te bereikenm zijn voldoende transities (iteraties) nodig. Het aantal iteraties dient in verhouding te staan tot de probleemgrootte en tot de grootte van de omgeving $|N(s)|$. Dit kan op twee manieren:

- Statisch: Het aantal iteraties per T wordt bepaald voor het algoritme start, bv. een deel y van de omgeving wordt geëxploreerd.
- Adaptief: het aantal onbezochte buren is afhankelijk van de zoekkarakteristieken, bv. functie van de waarde van de beste en de slechtste oplossing gevonden tijdens de binnenste lus; evenwicht op elke T is niet noodzakelijk.

4.2.3.3 Koeling

Compromis: Kwaliteit oplossging $\leftrightarrow$ koelsnelheid. De mogelijke T-afnames:

- Lineair: $T_i = T_0 - i \times \beta$, met T_i de temperatuur op iteratie i en β een constante.
- Geometrisch: $T = \alpha T$ met $\alpha \in]0,1[$ (het meest gebruikelijke koelschema, met α tussen $0,5$ en $0,99$)
- Logaritmisch: $T_i = \frac{T_0}{\log(i)}$ (te traag voor praktische bruikbaarheid, maar er bestaat en convergentiebewijs voor)
- Heel trage afname: ipv. veel iteraties per T, 1 iteratie per T en trage koeling: $T_{i+1} = \frac{T_i}{1+\beta T_i}$
- Niet-monotoon: vaak is het wenselijk om af en toe de temperatuur weer te laten toenemen *reheating*
- Adaptief: Wanneer het koelschema niet a priori vastligt (bv. een laag aantal iteraties op hoge T en een hoog aantal iteraties op lage T).

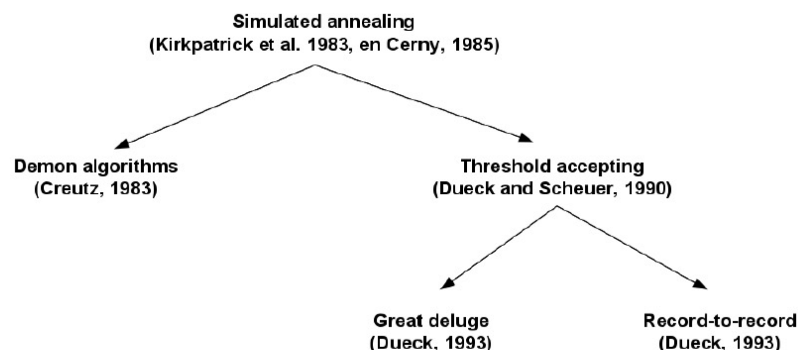
4.2.3.4 Stopcriterium

De theoretische eindtemperatuur is 0, maar in de praktijk gebruiken we dit niet. We gaan eerder volgende strategieën toepassen:

- Bereiken van een lage eindtemperatuur T_F
- Aantal iteraties zonder verbetering
- aantal temperatuurafnames zonder verbetering
- ...

4.2.4 Methoden gebaseerd op simulated annealing

Er zijn andere methoden die gebaseerd zijn op simulated annealing. Een aantal zijn gegeven in figuur 4.6.



Figuur 4.6: Andere methoden die gebaseerd zijn op simulated annealing

4.2.4.1 Threshold accepting

Dit is een deterministische variant van simulated annealing. Het algoritme voor deze methode is te zien in algorime 3. De oplossingen worden aanvaard indien ze niet meer dan Q slechter zijn dan de huidige oplossing.

$$P_i\left(\Delta(s,s')\right) = \begin{cases} 1, & \text{if } Q_i \geq \Delta(s,s') \\ 0, & \text{else} \end{cases}$$

Algorithm 3 Threshold accepting

1: **Input:** Drempel *annealing*
2: $s = s_0$ ▷ Genereer een initiële oplossing
3: $Q = Q_{max}$ ▷ Startdrempel
4: **repeat**
5: **repeat**
6: Genereer een random *neighbour* $s' \in N(s)$ ▷ Herhaal op een vaste temperatuur
7: $\Delta E = f(s') - f(s)$
8: **if** $\Delta E \leq 0$ **then**
9: $s = s'$
10: **else**
11: Accepteer s' met een probabiliteit $e^{\frac{-\Delta E}{T}}$
12: **until** Evenwichtsvoorwaarde bereikt ▷ bv. aantal iteraties uitgevoerd met drempel Q
13: $Q = g(Q)$ ▷ Aanpassing drempel
14: **until** Voldoen aan stopvoorwaarde ▷ bv. $Q < Q_{min}$
15: **Output:** Beste gevonden oplossing

4.2.4.2 Old bachelor accepting

De drempel gebruikt is deze methode is adaptief en monotoon. Hij zal dynamisch verandere (naar boven of naar beneden) op basis van de zoekhistoriek.

- Na aanvaarden van een slechtere oplossing, wordt de drempel lichtjes verhoogd; de bedoeling is om op die manier uit lokale optima te ontsnappen.
- Na aanvaarden van een betere oplossing, wordt drempel lichtjes verlaagd met de bedoeling op zoek te gaan naar het lokale optimum.

Algorithm 4 Old bachelor accepting

1: $s = s_0$ ▷ Genereer een initiële oplossing
2: $Q_0 = 0$ ▷ Startdrempel
3: *leeftijd* $= 0$
4: **for** $i = 0$ to $M - 1$ **do**
5: genereer een random oplossing $s' \in N(s)$
6: **if** $f(s') < f(s) + Q_i$ **then**
7: $s = s', leeftijd = 0$
8: **else**
9: leeftijd = leeftijd + 1
10: $Q_{i+1} = \left(\left(\frac{leeftijd}{a}\right)^b - 1\right) \times \Delta \times \left(1 - \frac{i}{M}\right)^c$
11: **Output:** Beste gevonden oplossing

4.2.4.3 Record-to-record travel

Dit is ook een deterministische variant van *simulated annealing*.

- *RECORD* stelt de waarde van de doelfunctie voor van de beste gevonde oplossing
- Niet betere oplossingen worden geaccepteerd wanneer de waarde van de doelfunctie kleiner is dan het record min een deviatie D: $RECORD - D$

Algorithm 5 Record-to-record travel

1: **Input:** Afwijking $D > 0$
2: $s = s_0$ ▷ Genereer een initiële oplossing
3: $RECORD = f(s)$
4: **repeat**
5: Genereer een random *neighbour* $s' \in N(s)$ ▷ Herhaal op een vaste drempel
6: **if** $f(s' < RECORD + D)$ **then**
7: $s = s'$
8: **if** $RECORD > f(s')$ **then**
9: $RECORD = f(s')$
10: **until** stopcriterium bereikt
11: **Output:** Beste gevonden oplossing

4.2.4.4 Great deluge

Deze methode heeft ook een deterministisch acceptatiecriterium. Het is geïnspireerd door het pad dat een wandelaar zou volgen tijdens een zondvloed

- Het regent onophoudelijk, waardoor het waterniveau blijft stijgen.
- De wandelaar neemt nooit een stap waarbij zijn voeten nat worden.

Algorithm 6 Great deluge

1: **Input:** LEVEL L
2: $s = s_0$ ▷ Genereer een initiële oplossing
3: Kies snelheid regen UP ▷ $UP > 0$
4: Kies initeel waterniveau $LEVEL$
5: **repeat**
6: Genereer een random *neighbour* $s' \in N(s)$
7: **if** $f(s') < LEVEL)$ **then**
8: $s = s'$
9: $LEVEL = LEVEL - UP$
10: **until** stopcriterium bereikt
11: **Output:** Beste gevonden oplossing

4.2.4.5 Demon algoritme

Dit algoritme heeft een eenvoudigere acceptatiefunctie dan in *simulated annealing*

- Gebaseerd op het energieniveau van de demon (krediet).
- De demon wordt geïnitialiseerd met een waarde D.
- Wanneer een slechtere oplossing aanvaard wordt, gaat dat ten koste van de demon
- Wanneer een betere oplossing aanvaard wordt, verkrijgt de demon extra krediet

Algorithm 7 Demon algoritme

1: **Input:** initiële waarde DEMON D
2: $s = s_0$ ▷ Genereer een initiële oplossing
3: **repeat**
4: Genereer een random *neighbour* $s' \in N(s)$
5: $\Delta E = f(s') - f(s)$
6: **if** $\Delta \leq D$ **then**
7: $s = s'$
8: $D = D - \Delta E$
9: **until** stopcriterium bereikt
10: **Output:** Beste gevonden oplossing

4.2.5 Samenvatting drempelalgoritmen

- (relatief) weinig parameters fijn te stellen
- Goede convergentie-eigenschappen
- *Simulated annealing*: theoretische convergentie naar het optimum in geval van oneindig traag koelen

4.3 Tabu search

Tabu seacht is ontworpen door Fred Glover (1986) en het is een uitbreiding op lokaal zoeken. Het algoritme voor tabu search is te zien in algoritme 8 Deze metaheursitiek maakt gebruik van een bepaald geheugen en voorkomt zo het vastzitten in een lokaal optimum, dit in tegestelling tot *hill climbing, steepest descent*. Tabu seacht is theoretisch minder ondersteund dan *simulated annealing*

4.3.1 Kenmerken van tabu search

- Het algoritme sturen zodat een zet van s naar s' in $N(s)$ kan uitvoeren, zefls als $f(s') > f(s)$ voor een minimalisatieprobleem.
- Probleem: Er is gevaar om in een lus terecht te komen door, na aanvaarden van een slechtere oplossing, terwijl bij een vorige beter oplossing terecht te komen.
- Tabu search gebruikt een geheugenstructuur die zetten verbiedt/bestraft wanneer ze terug zouden leiden naar een recente bezochte oplossing.
- flexibel (adaptief) geheuden

Algorithm 8 Tabu search

1: $s = s_0$ ▷ Genereer een initiële oplossing
2: $best = s$
3: **Initialiseer de tabu-lijst, middellange- en langetermijngeheugen**
4: **repeat**
5: Zoek de beste toegelaten oplossing in de buurt ▷ niet tabu of voldoet aan aspiratievoorwaarde
6: $s = s'$
7: **if** $f(s) < best$ **then**
8: $best = s$
9: Pas aan: tabu-lijst, aspiratievoorwaarden, middellange- en langetermijngeheugen
10: **if** voldaan aan intersificatievoorwaarde **then**
11: intensificatie
12: **if** voldaan aan diversificatievoorwaarde **then**
13: diversificatie
14: **until** Voldan aan stopvoorwaarde
15: **Output:** Beste gevonden oplossing

Zoekgeheuden	Rol	Voorstelling
Tabu-lijst	Voorkomt lussen	Bezochte oplossingen, kenmerken van *moves* (zetten), kenmerken van oplossingen
Middellangetermijn	Intensificatie	recente geheugen
Langetermijn	Diversificatie	Frekwentiegebaseerd geheugen

Tabel 4.2: Soorten geheugen

4.3.2 Kortetermijngeheugen

Het doel hiervan is informatie over recent bezochte oplossingen bij te houden met de bedoeling om lussen te vermijden. We kunnen bijvoorbeeld alle bezochte oplossingen gaan bijhouden. Dit heeft als voordeel dat het met zekerheid lussen voorkomen worden, maar als nadeel dat het niet werkbaar is wegens teveel geheugengebruik en rekentijd.
Er zijn betere alternatieven:

- De grootte van het kortetermijngeheugen beperken (lengte tabu-lijst)
- Hash codes gebruiken om oplossingen op te slaan
- Kermerken opslaan van de *move* zodat omgekeerde *moves* verboden worden door de tabu-lijst. Als move m toegepast wordt op oplossing s_i om oplossing $s_j = s_i \oplus m$ te genereren, dan wordt m (of m^{-1} zodanig dat $(s_i \oplus m) \oplus m^{-1}$) opgeslagen in de tabu-lijst.

4.3.3 Aspiratievoorwaarde

Definitie Aspiratie wanneer een tabu-oplossing een betere waarde voor de doelfunctie heeft dan de beste gevonden oplossing, dan wordt de tabu-status genegeerd.

4.3.4 Tabu-lijstlengte

Er zijn enkele mogelijkheden:

- Statisch: De tabu heeft een vaste lengte, die meestal afhangt van de probleemgrootte of van de grootte van de omgeving.
- Dynamisch: De lijstlengte verandert gedurende het zoeken onafhankelijk van het zoekproces. Bijvoorbeeld *robuuste tabu search* waarvij de tabu-lijstlengte verandert random uniform in een interval $[T_{min}, T_{max}]$
- Adaptief: De lengte van de tabu-lijst wordt op basis van informatie in het zoekproces aangepast. Bijvoorbeeld *reactieve tabu search* waarbij de tabu-lijst toeneemt wanneer er indicaties zijn van lussen (wanneer er gedurende een zekere tijd geen betere oplossingen gevonden wordt); De tabu-lijst neemt af wanneer er betere oplossingen gevonden worden.

4.3.5 Middellangetermijngeheugen

Hier wordt er aan intensificatie gedaan. Dit is het exploiteren van informatie over de beste gevonden oplossingen. Goede kenmerken gaan we gaan fixeren en we gaan zoeken naar andere oplossingen met deze kenmerken.

4.3.6 Langetermijngeheugen

Hier wordt er aan diversificatie gedaan. Dit is de zoektocht naar niet-geëxploreerde gebieden van de zoekruimte. We gaan hierbij **herstarten** met oplossingen die kenmerken vertonen die het minst vaak zijn voorgekomen in de bezochte oplossingen.

4.3.7 Iterated Local Search

Kenmerken van iterated local search:

Pertubatiemethode
- Moet meer effect hebben dan random herstarten
- Grootte van de pertubatie: vast, variabel, dynamisch, adaptief
- Random of gebruikmakend van informatie uit het geheugen

Acceptatievoorwaarde
- Probalistisch
- Deterministisch

Algorithm 9 Iterated local search

1: s_* =lokaal zoeken(s_0) ▷ Pas een lokaal zoekalgoritme toe
2: **repeat**
3: s' =perturbeer($s_*, zoekgeheugen$) ▷ Pertubeer het gevonden lokaal optimum
4: s'_* =lokaal zoeken(s') ▷ Lokaal zoeken op de gepertubeerde oplossing
5: s_* =accepteer($s_*, s'*, zoekgeheugen$)
6: **until** Voldan aan stopvoorwaarde
7: **Output:** Beste gevonden oplossing

4.3.8 Variable neighbourhood search

Bij deze metaheuristiek gaan we verschillende omgevingen exploreren. Het basis idee is dat verschillende omgevingen, verschillenden landschappen genereren. Een algoritme is te zien in algorimte 10, dit is een algoritme voor Variable neighbourhood descend.

Algorithm 10 Variable neighbourhood descent

1: **Input:** Een verzameing *neighbourhood* structuren N_l voor $l = 1 \cdots 1_{max}$
2: $x = x_0$
3: $l = 1$
4: **while** $l \leq l_{max}$ **do**
5: bepaal de beste *neighbour* x' van x in $N_l(x)$
6: **if** $f(x') < f(x)$ **then**
7: $x = x^{'}$
8: $l = 1$
9: **else**
10: $l = l + 1$
11: **Output:** Beste gevonden oplossing

Een algemenere vorm bevat stochastische componenten. Het basisalgoritme is te zien in algoritme 11 op de pagina hierna. Het bestaat uit drie stappen:

- Shake
- Lokaal zoeken
- Move

Algorithm 11 Basisalgoritme Variable Neighbourhood Search

Input: Een verzameling *neighbourhood* structuren N_l voor $l = 1, ..., l_{max}$
$x = x_0$
repeat
  $k = 1$
  repeat
    Shaking: neem een random oplossing x' uit de $k$de *neighbourhood* $N_k(x)$ van x
    x'' =lokaal zoeken (x')
    if $f(x'') < f(x)$ **then**
      $x = x''$
      $k = 1$
    else
      $k = k + 1$
  until $k = k_{max}$
until Voldaan aan stopcriterium
Output: Beste gevonde oplossing

Algorithm 12 Algemeen Variable neighbourhood search

N_k voor $k = 1, ..., k_{max}$ voor shaking, N_l voor $l = 1, ..., l_{max}$ voor lokaal zoeken.
$x = x_0$
repeat
  for $k = 1$ to k_{max} **do**
    Shaking: random oplossing x' uit $N_k(x)$ lokaal zoeken met VND.
    for $l = 1$ tot l_{max} **do**
      Zoek de beste *neighbour* x'' van x' in $N_l(x')$
      if $f(x'') < f(x')$ **then**
        $x' = x''$
        $l = 1$
      else
        $l = l + 1$
      Move of niet
      if lokaal optimum is beter dan x **then**
        $x = x''$
        $k = 1$
      else$k = k + 1$
until Voldaan aan stopcriterium
Output: Beste gevonden oplossing

4.3.9 Guided local search

Het basisprincipe is het dynamisch aanpassen van de doelfunctie zodat ze slechter wordt voor de reeds gegenereerde lokale optima. Hierbij laten we de veranderingen aan het zoeklandschap toe. Het algoritme is te zien in algoritme 13 op de volgende pagina.

- Definieer een verzameling van m eigenschappen van een oplossing $ft_i(i = 1, ..., m)$

- Elke eigenschap i wordt gekenmerkt door een kost p_i
- De doelfuntie van een oplossing wordt gepenaliseerd: $f''(s) = f(s) + \lambda \sum_{i=1}^{m} p_i I_i(s)$
  λ stelt de gewichten voor die overeenkomen met de kosten en $I_i(s)$ bepaalt de of kenmerk ft_i aanwezig is in de oplossing

$$I_i(s) = \begin{cases} 1 & \text{als kenmerk } ft_i \in s \\ 0 & anders \end{cases}$$

Als initiatie: $p_i = 0$ voor alle i.

Algorithm 13 Guided local search

Input: Lokaal zoeken metaheuristiek, λ, kenmerlen I, kosten c
$s = s_0$
$p_i = 0$
repeat
  s^* oplossing bekomen na toepassing van de metaheuristiek
  for elk kenmerk i van s^* **do**
    $u_i = \frac{c_i}{1 + p_i}$ ▷ bereken de utiliteit
  $u_j = max_{i=1,\ldots,m(u_i)}$
  $p_j = p_j + 1$
until Stopvoorwaarde bereikt
Output: Beste gevonden oplossing

4.3.10 Late acceptance strategie

Er is beperkt geheugen, nl. een lijst met waarden van de meest recent bezochte oplossingen. Een oplossing zal aanvaard worden als de waarde van de doelfunctie beter is dan de oudste waarde in de lijst. De lijst is een fifo lijst, het oudste element verdwijnt, de waarde van de nieuw bezochte oplossing wordt toegevoegd.

4.3.11 Step counting hill climbing

Het basisidee is om het hill climbing niet af te breken bij het eerste lokaal optimum. De kwaliteit van de huidige oplossing geldt als acceptatiegrens voor de volgende iteratie/stap in hill climbing. De kwaliteit van de huidige oplossing geldt als acceptatiegrens voor een aantal opeenvolgdende iteraties.

4.3.12 Andere

Noising Random *noise* toevoegen aan de waarde van de doelfuntie $[-r, +r]$, het interval verkleinen na iedere iteratie

GRASP *Greedy Randomised Adative Search Procedure*, tijdens elke iteratie een constructief en een lokaal zoekalgoritme toepassen.

Hoofdstuk 5

Complexiteit

5.1 No free lunch

De stelling zegt dat geen enkel zoekalgoritme beter is dan een ander wanneer het gedrah beschouwd wordt voor alle mogelijke discrete functies. **Complexiteitstheorie** bestudeert de tijd en geheugencomplexiteit van algoritmen.

5.2 Complexiteit , P en NP

De probleemklasse P is de verzameling problemen die in polynomiale tijd opgelost kunnen worden op een deterministische Turing machine. Algemeen worden die problemen beschouwd als haalbare problemen.
De probleemklasse NP is de verzameling problemen die in polynomiale tijd opgelost kunnen worden op een niet-deterministische Turing machine.
In een niet-deterministische Turing machine kunnen keuzes gemaakt worden zodanig dat niet alle berekeningen nodig zijn, de berekening wordt voorgesteld door een zoekboom. Een probleem is in NP wanneer de zoekboom polynomiaal is in hoogte. Merk op dat het aantal knopen exponentieel kan zijn.
Wanneer alle zoekpaden in parallel zouden kunnen doorlopen dan bekomen we in polynomiale tijd een oplossing. Wanneer *deterministisch* het juiste pad in de zoekbom bepaald kan worden voor elk probleem, dan behoort het probleem tot de klasse P.
Alle problemen in P behoren ook tot NP.

5.2.1 Algoritmische kloof

Er is een algortimische kloof indien de bewezen complexiteit van een probleem lager is dan de complexiteit van het beste algoritme tot nu toe gekend. Zo is bijvoorbeeld de complexiteit van sorteren $O(N \log n)$, maar er bestaan algoritmen die sorteren in $O(N \log n)$, dus is er geen algoritmische kloof.
Wanneer er algoritmen zijn die sneller zijn dan $O(N \log n)$, dan wil dit zeggen dat deze speciaal ontworpen zijn voor een bepaalde deelverzameling van problemen.

Voor het *traveling salesman problem (TSP)* bestaat er wel een algoritmische kloof. Het enige

algoritme dat gegerandeerd naar een optimale oplossing leidt komt neer op enumeratie. De beste gekende methode heeft dus in het slechste geval een complexiteit van $0(N!)$ voor een probleem net N steden. Toch is er nog niet bewezen dat de complexiteit van het TSP zodanig is dat het probleem niet in polynomiale tijd kan opgelost worden.

5.2.2 Klasse NP compleet

- $NPcompleet \subset NP$
- Een probleem R is NP compleet als
  1. R NP-hard is en
  2. R $\in$ NP
- R is NP-hard als het minstens even moeilijk op te lossen is als elk ander probleem in NP
- R is NP-hard als er een NP-compleet probleem R_0 bestaat zodanig dat het mogelijk is om elk geval van R_0 te herformuleren als een geval van R in deterministische polynomiale tijd.
- R moet even moeilijk zijn als R_0

Vermist we niet weten hoe de optimale oplossing te zoeken voor NP-harde probleen, moeten we ons beperken tot benaderende oplossingen, die wel in polynomiale tijd te berekenen zijn.
We moeten hierbij elke discrete functie beschouwen. Hierbij is een van de minst intelegente algoritmen *random enumeratie*.
Bij dit algoritme worden steekproeven in de oplossingsruimte gedaan (met risico op herhaaldelijk bekijken van dezelfde oplossing). Volgens de *no free lunch* stelling is geen enkel algoritme beter dan de random enumeratie.

Het verschil tussen stochastische en deterministische algortimen.

- Deterministisch algoritme: Bereikt vanuit een gegeven oplossin altijd dezelfde oplossing (vb:steepest descent)
- Stochastisch algoritme: Tijdens het zoeken gebeuren er veel random beslissingen (vb:simulated annealing)

5.3 Tijd- en geheugencomplexiteit

Bepalen hoeveel uitvoeringstijd en geheugenruimte gebruikt wordt bij het uitvoeren van een algoritme.

5.3.1 Tijdcomplexiteit

Wat zijn invloedsfactoren:

- Hardware: snelheid CPU, bus, randapparatuur

- Competitie met ander gebruikers van de resources
- Programmeertaal en kwaliteit van de code
- Programmeervaardigheid

Het aantal stappen zijn het aantal basisbewegingen om een zekere grootte van input te verwerken.
De uitvoeringstijd mag niet afhankelijk zijn van de specifieke waarden van de operanden.
De inputgrootte zijn het aantal inputgegevens die verwerkt worden.
Het model voor tijdscomplexiteti geldt enkel voor sequentieële verwerking

- Gaat uit van een standaard instructieset; alle basisinstructies (optellen, vermenigvuldiging, toekenning) nemen 1 tijdseenheid in beslag
- Aanmenen dat alle gehelen getalen (int) eenzelfde formaat hebben
- onbepert RAM-geheugen

Voorbeeld: Voorbeeld 1

- Afhalen van een bestand van het internet
- Stel dat er 2 seconden nodig zijn voor het opstarten van de verbinding en dat daarna de transfer gebeurt aan 1.6K/s als het bestand n Kbytes groot is dan is de tijd nodig voor het afhalen gegeven door $T(n) = \frac{n}{1.6} + 2$
- Afhalen van een bestand van 80K duurt ongeveer 52s, voor een bestand van 160K is ongeveer 102s nodig (lineaire uitvoeringstijd).

Voorbeeld: Voorbeeld 2
Methode om $\sum_{i=1}^{n} i^3$ te berekenen

```
static int som (int n) {          //1
   int res = 0;                   //2
   for (int i = 1; i ≤ n; i++)    //3
      res += i*i*i;               //4
   return res;                    //5
}                                 //6
```

- Analyse: n is een maat voor de probleemgrootte
- Lijnen 2 en 5 tellen voor 1 eenheid
- Elk van de n van lijn 4 vergt 3 tijdseenheden: $3n$ eenheden in totaal
- Lijn 3 veroorzaakt een kost voor initialisatie van i; voor de test of $i \leq n$ en voor het incrementeren van i in elke stap: de kost is $2n+1$ eenheden
- Als we de oproep van de functie en de terugkeer van het oproepende programma verlozen is de tijdskost $T(n) = 5n+3$

- *cn:* met c een constante: **lineaire** orde van toename
- n^2: **kwadratisch** algoritme.
  De functie van de uitvoeringstijd heeft een dominante term in n^2.
  In de praktijk zijn kwadratische algoritmen meestal onbruikbaar wanneer de inputgrootte meer dan een paar duizend is.
- n^3: **kubisch**
  Een inputgrootte van enkele 100-den is niet meer haalbaar
- $n \log n$: **logaritmisch**
  De dominantie ter is $n \times \log_2 n$
- 2^n: **exponenieel**
  Dergelijke algoritmen zijn onaanvaarbaar duur, zelfs voor kleinde waarden van n.

5.4 Asymptotische analyse

- Constanten van de lagere orde termen negeren wanneer een schatting gemaakt wordt van de uitvoeringstijd.
- bv. $f(n) = 10n^3 + n^2 + 40n + 80$
- voor $n = 1000$ is $f(n) = 1001040080$
- Door enekl de kubsiche term in rekening te brengen is de fout slechts $0,01\%$
- Het in rekening brengen van de leidende constante is weinig zinvol omdat de exacte waarde van de uitvoeringsrijd toch afhankelijk is van de computer
- Het gedrag van de functie is weinig zinvol kleine inputwaarden in dat geval wordt best het eenvoudigste algoritme gekozen.
- Belangrijk: de orde van toename= O-notatie

De bovengrens voor de uitvoeringstijd geeft aan wat de hoogste orde van toename is die een algoritme kan hebben.

Chapter 3

INTEGER PROGRAMMING

Robert Bosch
Oberlin College
Oberlin OH, USA

Michael Trick
Carnegie Mellon University
Pittsburgh PA, USA

3.1 INTRODUCTION

Over the last 20 years, the combination of faster computers, more reliable data, and improved algorithms has resulted in the near-routine solution of many integer programs of practical interest. Integer programming models are used in a wide variety of applications, including scheduling, resource assignment, planning, supply chain design, auction design, and many, many others. In this tutorial, we outline some of the major themes involved in creating and solving integer programming models.

The foundation of much of analytical decision making is linear programming. In a linear program, there are *variables*, *constraints*, and an *objective function*. The variables, or decisions, take on numerical values. Constraints are used to limit the values to a feasible region. These constraints must be linear in the decision variables. The objective function then defines which particular assignment of feasible values to the variables is optimal: it is the one that maximizes (or minimizes, depending on the type of the objective) the objective function. The objective function must also be linear in the variables. See Chapter 2 for more details about Linear Programming.

Linear programs can model many problems of practical interest, and modern linear programming optimization codes can find optimal solutions to problems with hundreds of thousands of constraints and variables. It is this combination of modeling strength and solvability that makes linear programming so important.

Integer programming adds additional constraints to linear programming. An integer program begins with a linear program, and adds the requirement that some or all of the variables take on integer values. This seemingly innocuous change greatly increases the number of problems that can be modeled, but also makes the models more difficult to solve. In fact, one frustrating aspect of integer programming is that two seemingly similar formulations for the same problem can lead to radically different computational experience: one formulation may quickly lead to optimal solutions, while the other may take an excessively long time to solve.

There are many keys to successfully developing and solving integer programming models. We consider the following aspects:

- be creative in formulations,
- find integer programming formulations with a strong relaxation,
- avoid symmetry,
- consider formulations with many constraints,
- consider formulations with many variables,
- modify branch-and-bound search parameters.

To fix ideas, we will introduce a particular integer programming model, and show how the main integer programming algorithm, branch-and-bound, operates on that model. We will then use this model to illustrate the key ideas to successful integer programming.

3.1.1 Facility Location

We consider a facility location problem. A chemical company owns four factories that manufacture a certain chemical in raw form. The company would like to get in the business of refining the chemical. It is interested in building refining facilities, and it has identified three possible sites. Table 3.1 contains variable costs, fixed costs, and weekly capacities for the three possible refining facility sites, and weekly production amounts for each factory. The variable costs are in dollars per week and include transportation costs. The fixed costs are in dollars per year. The production amounts and capacities are in tons per week.

The decision maker who faces this problem must answer two very different types of questions: questions that require numerical answers (for example, how many tons of chemical should factory i send to the site-j refining facility each week?) and questions that require yes–no answers (for example, should the site-j facility be constructed?). While we can easily model the first type of question by using continuous decision variables (by letting x_{ij} equal the

Table 3.1. Facility location problem.

		Site 1	Site 2	Site 3	Production
Variable cost	factory 1	25	20	15	1000
	factory 2	15	25	20	1000
	factory 3	20	15	25	500
	factory 4	25	15	15	500
Fixed cost		500 000	500 000	500 000	
Capacity		1 500	1 500	1 500	

number of tons of chemical sent from factory i to site j each week), we *cannot* do this with the second. We need to use integer variables. If we let y_j equal 1 if the site-j refining facility is constructed and 0 if it is not, we quickly arrive at an IP formulation of the problem:

$$
\begin{aligned}
\text{minimize} \quad & 52 \cdot 25x_{11} + 52 \cdot 20x_{12} + 52 \cdot 15x_{13} \\
& + 52 \cdot 15x_{21} + 52 \cdot 25x_{22} + 52 \cdot 20x_{23} \\
& + 52 \cdot 20x_{31} + 52 \cdot 15x_{32} + 52 \cdot 25x_{33} \\
& + 52 \cdot 25x_{41} + 52 \cdot 15x_{42} + 52 \cdot 15x_{43} \\
& + 500\,000y_1 + 500\,000y_2 + 500\,000y_3 \\
\text{subject to} \quad & x_{11} + x_{12} + x_{13} = 1000 \\
& x_{21} + x_{22} + x_{23} = 1000 \\
& x_{31} + x_{32} + x_{33} = 500 \\
& x_{41} + x_{42} + x_{43} = 500 \\
& x_{11} + x_{21} + x_{31} + x_{41} \le 1500y_1 \\
& x_{12} + x_{22} + x_{32} + x_{42} \le 1500y_2 \\
& x_{13} + x_{23} + x_{33} + x_{43} \le 1500y_3 \\
& x_{ij} \ge 0 \quad \text{for all } i \text{ and } j \\
& y_j \in \{0, 1\} \quad \text{for all } j
\end{aligned}
$$

The objective is to minimize the yearly cost, the sum of the variable costs (which are measured in dollars per week) and the fixed costs (which are measured in dollars per year). The first set of constraints ensures that each factory's weekly chemical production is sent somewhere for refining. Since factory 1 produces 1000 tons of chemical per week, factory 1 must ship a total of 1000 tons of chemical to the various refining facilities each week. The second set of constraints guarantees two things: (1) if a facility is open, it will operate at or below its capacity, and (2) if a facility is not open, it will not operate at all. If the site-1 facility is open ($y_1 = 1$) then the factories can send it up to $1500y_1 = 1500 \cdot 1 = 1500$ tons of chemical per week. If it is not open

($y_1 = 0$), then the factories can send it up to $1500y_1 = 1500 \cdot 0 = 0$ tons per week.

This introductory example demonstrates the need for integer variables. It also shows that with integer variables, one can model simple logical requirements (if a facility is open, it can refine up to a certain amount of chemical; if not, it cannot do any refining at all). It turns out that with integer variables, one can model a whole host of logical requirements. One can also model fixed costs, sequencing and scheduling requirements, and many other problem aspects.

3.1.2 Solving the Facility Location IP

Given an integer program (IP), there is an associated linear program (LR) called the *linear relaxation*. It is formed by dropping (relaxing) the integrality restrictions. Since (LR) is less constrained than (IP), the following are immediate:

- If (IP) is a minimization problem, the optimal objective value of (LR) is less than or equal to the optimal objective value of (IP).

- If (IP) is a maximization problem, the optimal objective value of (LR) is greater than or equal to the optimal objective value of (IP).

- If (LR) is infeasible, then so is (IP).

- If all the variables in an optimal solution of (LR) are integer-valued, then that solution is optimal for (IP) too.

- If the objective function coefficients are integer-valued, then for minimization problems, the optimal objective value of (IP) is greater than or equal to the ceiling of the optimal objective value of (LR). For maximization problems, the optimal objective value of (IP) is less than or equal to the floor of the optimal objective value of (LR).

In summary, solving (LR) can be quite useful: it provides a bound on the optimal value of (IP), and may (if we are lucky) give an optimal solution to (IP).

For the remainder of this section, we will let (IP) stand for the Facility Location integer program and (LR) for its linear programming relaxation. When

we solve (LR), we obtain

$$\begin{array}{|ccc|}\hline \multicolumn{3}{|c|}{\text{Objective}} \\ \hline x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \\ x_{41} & x_{42} & x_{43} \\ \hline y_1 & y_2 & y_3 \\ \hline \end{array} = \begin{array}{|ccc|}\hline \multicolumn{3}{|c|}{3340\,000} \\ \hline \cdot & \cdot & 1000 \\ 1000 & \cdot & \cdot \\ \cdot & 500 & \cdot \\ \cdot & 500 & \cdot \\ \hline \frac{2}{3} & \frac{2}{3} & \frac{2}{3} \\ \hline \end{array}$$

This solution has factory 1 send all 1000 tons of its chemical to site 3, factory 2 send all 1000 tons of its chemical to site 1, factory 3 send all 500 tons to site 2, and factory 4 send all 500 tons to site 2. It constructs two-thirds of a refining facility at each site. Although it costs only 3340 000 dollars per year, it cannot be implemented; all three of its integer variables take on fractional values.

It is tempting to try to produce a feasible solution by rounding. Here, if we round y_1, y_2, and y_3 from 2/3 to 1, we get lucky (this is certainly not always the case!) and get an integer feasible solution. Although we can state that this is a good solution—its objective value of 3840 000 is within 15% of the objective value of (LR) and hence within 15% of optimal—we cannot be sure that it is optimal.

So how can we find an optimal solution to (IP)? Examining the optimal solution to (LR), we see that y_1, y_2, and y_3 are fractional. We want to force y_1, y_2, and y_3 to be integer valued. We start by *branching* on y_1, creating two new integer programming problems. In one, we add the constraint $y_1 = 0$. In the other, we will add the constraint $y_1 = 1$. Note that any optimal solution to (IP) must be feasible for one of the two subproblems.

After we solve the linear programming relaxations of the two subproblems, we can display what we know in a tree, as shown in Figure 3.1.

Note that the optimal solution to the left subproblem's LP relaxation is integer valued. It is therefore an optimal solution to the left subproblem. Since there is no point in doing anything more with the left subproblem, we mark it with an "×" and focus our attention on the right subproblem.

Both y_2 and y_3 are fractional in the optimal solution to the right subproblem's LP relaxation. We want to force both variables to be integer valued. Although we could branch on either variable, we will branch on y_2. That is, we will create two more subproblems, one with $y_2 = 0$ and the other with $y_2 = 1$. After we solve the LP relaxations, we can update our tree, as in Figure 3.2.

Note that we can immediately "× out" the left subproblem; the optimal solution to its LP relaxation is integer valued. In addition, by employing a *bounding* argument, we can also × out the right subproblem. The argument goes like this: Since the objective value of its LP relaxation ($3636\,666\frac{2}{3}$) is greater than the objective value of our newly found integer feasible solution

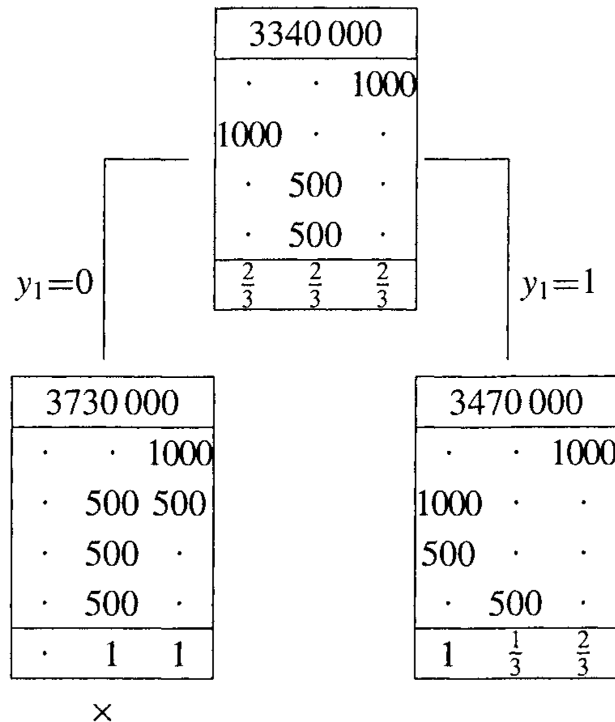


Figure 3.1. Intermediate branch and bound tree.

(3470 000), the optimal value of the right subproblem must be higher than (worse than) the objective value of our newly found integer feasible solution. So there is no point in expending any more effort on the right subproblem.

Since there are no active subproblems (subproblems that require branching), we are done. We have found an optimal solution to (IP). The optimal solution has factories 2 and 3 use the site-1 refining facility and factories 1 and 4 use the site-3 facility. The site-1 and site-3 facilities are constructed. The site-2 facility is not. The optimal solution costs 3470 000 dollars per year, 370 000 dollars per year less than the solution obtained by rounding the solution to (LR).

This method is called *branch and bound*, and is the most common method for finding solutions to integer programming formulations.

3.1.3 Difficulties with Integer Programs

While we were able to get the optimal solution to the example integer program relatively quickly, it is not always the case that branch and bound quickly solves integer programs. In particular, it is possible that the *bounding* aspects of branch and bound are not invoked, and the branch and bound algorithm can then generate a huge number of subproblems. In the worst case, a problem with n binary variables (variables that have to take on the value 0 or 1) can have 2^n subproblems. This exponential growth is inherent in any algorithm for integer programming, unless P = NP (see Chapter 11 for more details), due to the range of problems that can be formulated within integer programming.

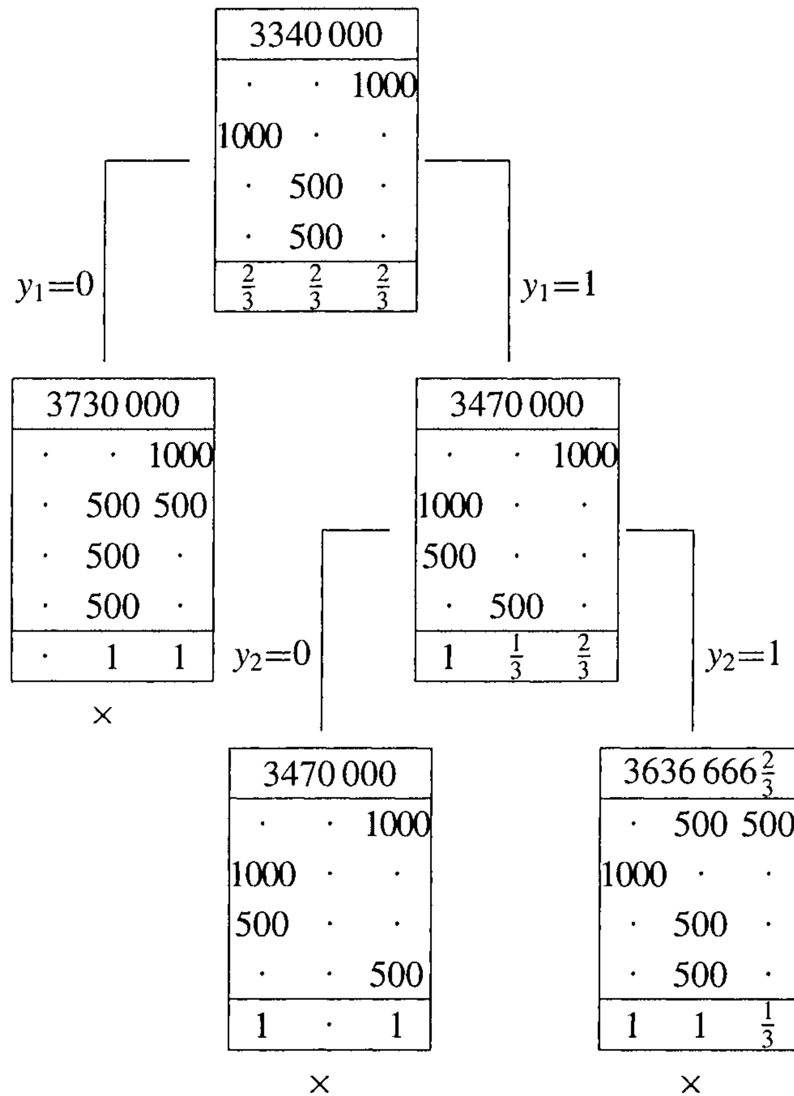


Figure 3.2. Final branch and bound tree.

Despite the possibility of extreme computation time, there are a number of techniques that have been developed to increase the likelihood of finding optimal solutions quickly. After we discuss creativity in formulations, we will discuss some of these techniques.

3.2 BE CREATIVE IN FORMULATIONS

At first, it may seem that integer programming does not offer much over linear programming: both require linear objectives and constraints, and both have numerical variables. Can requiring some of the variables to take on integer values significantly expand the capability of the models? Absolutely: integer programming models go far beyond the power of linear programming models. The key is the creative use of integrality to model a wide range of common

structures in models. Here we outline some of the major uses of integer variables.

3.2.1 Integer Quantities

The most obvious use of integer variables is when an integer quantity is required. For instance, in a production model involving television sets, an integral number of television sets might be required. Or, in a personnel assignment problem, an integer number of workers might be assigned to a shift.

This use of integer variables is the most obvious, and the most over-used. For many applications, the added "accuracy" in requiring integer variables is far outweighed by the greater difficulty in finding the optimal solution. For instance, in the production example, if the number of televisions produced is in the hundreds (say the fractional optimal solution is 202.7) then having a plan with the rounded off value (203 in this example) is likely to be appropriate in practice. The uncertainty of the data almost certainly means that no production plan is accurate to four figures! Similarly, if the personnel assignment problem is for a large enterprise over a year, and the linear programming model suggests 154.5 people are required, it is probably not worthwhile to invoke an integer programming model in order to handle the fractional parts.

However, there are times when integer quantities are required. A production system that can produce either two or three aircraft carriers and a personnel assignment problem for small teams of five or six people are examples. In these cases, the addition of the integrality constraint can mean the difference between useful models and irrelevant models.

3.2.2 Binary Decisions

Perhaps the most used type of integer variable is the *binary variable*: an integer variable restricted to take on the values 0 or 1. We will see a number of uses of these variables. Our first example is in modeling binary decisions.

Many practical decisions can be seen as "yes" or "no" decisions: Should we construct a chemical refining facility in site j (as in the introduction)? Should we invest in project B? Should we start producing new product Y? For many of these decisions, a binary integer programming model is appropriate. In such a model, each decision is modeled with a binary variable: setting the variable equal to 1 corresponds to making the "yes" decision, while setting it to 0 corresponds to going with the "no" decision. Constraints are then formed to correspond to the effects of the decision.

As an example, suppose we need to choose among projects A, B, C, and D. Each project has a capital requirement ($1 million, $2.5 million, $4 million, and $5 million respectively) and an expected return (say, $3 million, $6 million,

\$13 million, and \$16 million). If we have \$7 million to invest, which projects should we take on in order to maximize our expected return?

We can formulate this problem with binary variables x_A, x_B, x_C, and x_D representing the decision to take on the corresponding project. The effect of taking on a project is to use up some of the funds we have available to invest. Therefore, we have a constraint:

$$x_A + 2.5x_B + 4x_C + 5x_D \leq 7$$

Our objective is to maximize the expected profit:

$$\text{Maximize } 3x_1 + 6x_2 + 13x_3 + 15x_4$$

In this case, binary variables let us make the yes–no decision on whether to invest in each fund, with a constraint ensuring that our overall decisions are consistent with our budget. Without integer variables, the solution to our model would have fractional parts of projects, which may not be in keeping with the needs of the model.

3.2.3 Fixed Charge Requirements

In many production applications, the cost of producing x of an item is roughly linear except for the special case of producing no items. In that case, there are additional savings since no equipment or other items need be procured for the production. This leads to a *fixed charge* structure. The cost for producing x of an item is

- 0, if $x = 0$
- $c_1 + c_2x$, if $x > 0$ for constants c_1, c_2

This type of cost structure is impossible to embed in a linear program. With integer programming, however, we can introduce a new binary variable y. The value $y = 1$ is interpreted as having non-zero production, while $y = 0$ means no production. The objective function for these variables then becomes

$$c_1y + c_2x$$

which is appropriately linear in the variables. It is necessary, however, to add constraints that link the x and y variables. Otherwise, the solution might be $y = 0$ and $x = 10$, which we do not want. If there is an upper bound M on how large x can be (perhaps derived from other constraints), then the constraint

$$x \leq My$$

correctly links the two variables. If $y = 0$ then x must equal 0; if $y = 1$ then x can take on any value. Technically, it is possible to have the values $x = 0$ and

$y = 1$ with this formulation, but as long as this is modeling a fixed cost (rather than a fixed profit), this will not be an optimal (cost minimizing) solution.

This use of "M" values is common in integer programming, and the result is called a "Big-M model". Big-M models are often difficult to solve, for reasons we will see.

We saw this fixed-charge modeling approach in our initial facility location example. There, the y variables corresponded to opening a refining facility (incurring a fixed cost). The x variables correspond to assigning a factory to the refining facility, and there was an upper bound on the volume of raw material a refinery could handle.

3.2.4 Logical Constraints

Binary variables can also be used to model complicated logical constraints, a capability not available in linear programming. In a facility location problem with binary variables y_1, y_2, y_3, y_4, and y_5 corresponding to the decisions to open warehouses at locations 1, 2, 3, 4 and 5 respectively, complicated relationships between the warehouses can be modeled with linear functions of the y variables. Here are a few examples:

- At most one of locations 1 and 2 can be opened: $y_1 + y_2 \leq 1$.
- Location 3 can only be opened if location 1 is $y_3 \leq y_1$.
- Location 4 cannot be opened if locations 2 or 3 are such that $y_4 + y_2 \leq 1$ or $y_4 + y_3 \leq 1$.
- If location 1 is open, either locations 2 or 5 must be $y_2 + y_5 \geq y_1$.

Much more complicated logical constraints can be formulated with the addition of new binary variables. Consider a constraint of the form: $3x_1 + 4x_2 \leq 10$ OR $4x_1 + 2x_2 \geq 12$. As written, this is not a linear constraint. However, if we let M be the largest either $|3x_1 + 4x_2|$ or $|4x_1 + 2x_2|$ can be, then we can define a new binary variable z which is 1 only if the first constraint is satisfied and 0 only if the second constraint is satisfied. Then we get the constraints

$$3x_1 + 4x_2 \leq 10 + (M - 10)(1 - z)$$
$$4x_1 + 2x_2 \geq 12 - (M + 12)z$$

When $z = 1$, we obtain

$$3x_1 + 4x_2 \leq 10$$
$$4x_1 + 2x_2 \geq -M$$

When $z = 0$, we obtain

$$3x_1 + 4x_2 \leq M$$
$$4x_1 + 2x_2 \geq 12$$

This correctly models the original nonlinear constraint.

As we can see, logical requirements often lead to Big-M-type formulations.

3.2.5 Sequencing Problems

Many problems in sequencing and scheduling require the modeling of the order in which items appear in the sequence. For instance, suppose we have a model in which there are items, where each item i has a processing time on a machine p_i. If the machine can only handle one item at a time and we let t_i be a (continuous) variable representing the start time of item i on the machine, then we can ensure that items i and j are not on the machine at the same time with the constraints

$$t_j \geq t_i + p_i \text{ IF } t_j \geq t_i$$
$$t_i \geq t_j + p_j \text{ IF } t_j < t_i$$

This can be handled with a new binary variable y_{ij} which is 1 if $t_i \leq t_j$ and 0 otherwise. This gives the constraints

$$t_j \geq t_i + p_i - M(1 - y)$$
$$t_i \geq t_j + p_j - My$$

for sufficiently large M. If y is 1 then the second constraint is automatically satisfied (so only the first is relevant) while the reverse happens for $y = 0$.

3.3 FIND FORMULATIONS WITH STRONG RELAXATIONS

As the previous section made clear, integer programming formulations can be used for many problems of practical interest. In fact, for many problems, there are many alternative integer programming formulations. Finding a "good" formulation is key to the successful use of integer programming. The definition of a good formulation is primarily computational: a good formulation is one for which branch and bound (or another integer programming algorithm) will find and prove the optimal solution quickly. Despite this empirical aspect of the definition, there are some guidelines to help in the search for good formulations. The key to success is to find formulations whose linear relaxation is not too different from the underlying integer program.

We saw in our first example that solving linear relaxations was key to the basic integer programming algorithm. If the solution to the initial linear relaxation is integer, then no branching need be done and integer programming is no harder than linear programming. Unfortunately, finding formulations with this property is very hard to do. But some formulations can be better than other formulations in this regard.

Let us modify our facility location problem by requiring that every factory be assigned to exactly one refinery (incidentally, the optimal solution to our original formulation happened to meet this requirement). Now, instead of having x_{ij} be the tons sent from factory i to refinery j, we define x_{ij} to be 1 if factory i is serviced by refinery j. Our formulation becomes

$$
\begin{aligned}
\text{Minimize} \quad & 1000 \cdot 52 \cdot 25x_{11} + 1000 \cdot 52 \cdot 20x_{12} + 1000 \cdot 52 \cdot 15x_{13} \\
& + 1000 \cdot 52 \cdot 15x_{21} + 1000 \cdot 52 \cdot 25x_{22} + 1000 \cdot 52 \cdot 20x_{23} \\
& + 500 \cdot 52 \cdot 20x_{31} + 500 \cdot 52 \cdot 15x_{32} + 500 \cdot 52 \cdot 25x_{33} \\
& + 500 \cdot 52 \cdot 25x_{41} + 500 \cdot 52 \cdot 15x_{42} + 500 \cdot 52 \cdot 15x_{43} \\
& + 500\,000y_1 + 500\,000y_2 + 500\,000y_3 \\
\text{Subject to} \quad & x_{11} + x_{12} + x_{13} = 1 \\
& x_{21} + x_{22} + x_{23} = 1 \\
& x_{31} + x_{32} + x_{33} = 1 \\
& x_{41} + x_{42} + x_{43} = 1 \\
& \quad 1000x_{11} + 1000x_{21} + 500x_{31} + 500x_{41} \leq 1500y_1 \\
& \quad 1000x_{12} + 1000x_{22} + 500x_{32} + 500x_{42} \leq 1500y_2 \\
& \quad 1000x_{13} + 1000x_{23} + 500x_{33} + 500x_{43} \leq 1500y_3 \\
& \qquad x_{ij} \in \{0, 1\} \quad \text{for all } i \text{ and } j \\
& \qquad y_j \in \{0, 1\} \quad \text{for all } j.
\end{aligned}
$$

Let us call this formulation the *base formulation*. This is a correct formulation to our problem. There are alternative formulations, however. Suppose we add to the base formulation the set of constraints

$$x_{ij} \leq y_j \quad \text{for all } i \text{ and } j$$

Call the resulting formulation the *expanded formulation*. Note that it too is an appropriate formulation for our problem. At the simplest level, it appears that we have simply made the formulation larger: there are more constraints so the linear programs solved within branch-and-bound will likely take longer to solve. Is there any advantage to the expanded formulation?

The key is to look at non-integer solutions to linear relaxations of the two formulations: we know the two formulations have the same integer solutions (since they are formulations of the same problem), but they can differ in non-integer solutions. Consider the solution $x_{13} = 1, x_{21} = 1, x_{32} = 1, x_{42} = 1$, $y_1 = 2/3$, $y_2 = 2/3$, $y_3 = 2/3$. This solution is feasible to the linear relaxation of the base formulation but is not feasible to the linear relaxation of the expanded formulation. If the branch-and-bound algorithm works on the base formulation, it may have to consider this solution; with the expanded formulation, this solution can never be examined. If there are fewer fractional solutions to explore (technically, fractional extreme point solutions), branch and bound will typically terminate more quickly.

Since we have added constraints to get the expanded formulation, there is no non-integer solution to the linear relaxation of the expanded formulation that is not also feasible for the linear relaxation of the base formulation. We say that the expanded formulation is *tighter* than the base formulation.

In general, tighter formulations are to be preferred for integer programming formulations even if the resulting formulations are larger. Of course, there are exceptions: if the size of the formulation is much larger, the gain from the tighter formulation may not be sufficient to offset the increased linear programming times. Such cases are definitely the exception, however: almost invariably, tighter formulations are better formulations. For this particular instance, the Expanded Formulation happens to provide an integer solution without branching.

There has been a tremendous amount of work done on finding tighter formulations for different integer programming models. For many types of problems, classes of constraints (or *cuts*) to be added are known. These constraints can be added in one of two ways: they can be included in the original formulation or they can be added as needed to remove fractional values. The latter case leads to a *branch and cut* approach, which is the subject of Section 3.6.

A cut relative to a formulation has to satisfy two properties: first, every feasible integer solution must also satisfy the cut; second, some fractional solution that is feasible to the linear relaxation of the formulation must not satisfy the cut. For instance, consider the single constraint

$$3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16$$

where the x_i are binary variables. Then the constraint $x_3 + x_4 \leq 1$ is a cut (every integer solution satisfies it and, for instance $x = (0, 0, .5, 1)$ does not) but $x_1 + x_2 + x_3 + x_4 \leq 4$ is not a cut (no fractional solutions removed) nor is $x_1 + x_2 + x_3 \leq 2$ (which incorrectly removes $x = (1, 1, 1, 0)$.

Given a formulation, finding cuts to add to it to strengthen the formulation is not a routine task. It can take deep understanding, and a bit of luck, to find improving constraints.

One generally useful approach is called the Chvatal (or Gomory–Chvatal) procedure. Here is how the procedure works for "$\leq$" constraints where all the variables are non-negative integers:

1 Take one or more constraints, multiply each by a non-negative constant (the constant can be different for different constraints). Add the resulting constraints into a single constraint.

2 Round down each coefficient on the left-hand side of the constraint.

3 Round down the right-hand side of the constraint.

The result is a constraint that does not cut off any feasible integer solutions. It may be a cut if the effect of rounding down the right-hand side of the constraint is more than the effect of rounding down the coefficients.

This is best seen through an example. Taking the constraint above, let us take the two constraints

$$3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16 \qquad x_3 \leq 1$$

If we multiply each constraint by 1/9 and add them we obtain

$$3/9x_1 + 5/9x_2 + 9/9x_3 + 10/9x_4 \leq 17/9$$

Now, round down the left-hand coefficients (this is valid since the x variables are non-negative and it is a "$\leq$" constraint):

$$x_3 + x_4 \leq 17/9$$

Finally, round down the right-hand side (this is valid since the x variables are integer) to obtain

$$x_3 + x_4 \leq 1$$

which turns out to be a cut. Notice that the three steps have differing effects on feasibility. The first step, since it is just taking a linear combination of constraints, neither adds nor removes feasible values; the second step weakens the constraint, and may add additional fractional values; the third step strengthens the constraint, ideally removing fractional values.

This approach is particularly useful when the constants are chosen so that no rounding down is done in the second step. For instance, consider the following set of constraints (where the x_i are binary variables):

$$x_1 + x_2 \leq 1 \qquad x_2 + x_3 \leq 1 \qquad x_1 + x_3 \leq 1$$

These types of constraints often appear in formulations where there are lists of mutually exclusive variables. Here, we can multiply each constraint by 1/2 and add them to obtain

$$x_1 + x_2 + x_3 \leq 3/2$$

There is no rounding down on the left-hand side, so we can move on to rounding down the right-hand side to obtain

$$x_1 + x_2 + x_3 \leq 1$$

which, for instance, cuts off the solution $x = (1/2, 1/2, 1/2)$.

In cases where no rounding down is needed on the left-hand side but there is rounding down on the right-hand side, the result has to be a cut (relative to the included constraints). Conversely, if no rounding down is done on the right-hand side, the result cannot be a cut.

In the formulation section, we mentioned that "Big-M" formulations often lead to poor formulations. This is because the linear relaxation of such a formulation often allows for many fractional values. For instance, consider the constraint (all variables are binary)

$$x_1 + x_2 + x_3 \leq 1000y$$

Such constraints often occur in facility location and related problems. This constraint correctly models a requirement that the x variables can be 1 only if y is also 1, but does so in a very weak way. Even if the x values of the linear relaxation are integer, y can take on a very small value (instead of the required 1). Here, even for $x = (1, 1, 1)$, y need only be 3/1000 to make the constraint feasible. This typically leads to very bad branch-and-bound trees: the linear relaxation gives little guidance as to the "true" values of the variables.

The following constraint would be better:

$$x_1 + x_2 + x_3 \leq 3y$$

which forces y to take on larger values. This is the concept of making the M in Big-M as small as possible. Better still would be the three constraints

$$x_1 \leq y \qquad x_2 \leq y \qquad x_3 \leq y$$

which force y to be integer as soon as the x values are.

Finding improved formulations is a key concept to the successful use of integer programming. Such formulations typically revolve around the strength of the linear relaxation: does the relaxation well-represent the underlying integer program? Finding classes of cuts can improve formulations. Finding such classes can be difficult, but without good formulations, integer programming models are unlikely to be successful except for very small instances.

3.4 AVOID SYMMETRY

Symmetry often causes integer programming models to fail. Branch-and-bound can become an extremely inefficient algorithm when the model being solved displays many symmetries.

Consider again our facility location model. Suppose instead of having just one refinery at a site, we were permitted to have up to three refineries at a site. We could modify our model by having variables y_j, z_j and w_j for each site (representing the three refineries). In this formulation, the cost and other coefficients for y_j are the same as for z_j and w_j. The formulation is straightforward, but branch and bound does very poorly on the result.

The reason for this is symmetry: for every solution in the branch-and-bound tree with a given y, z, and w, there is an equivalent solution with z taking on y's values, w taking on z's and y taking on w. This greatly increases the number

of solutions that the branch-and-bound algorithm must consider in order to find and prove the optimality of a solution.

It is very important to remove as many symmetries in a formulation as possible. Depending on the problem and the symmetry, this removal can be done by adding constraints, fixing variables, or modifying the formulation.

For our facility location problem, the easiest thing to do is to add the constraints

$$y_j \geq z_j \geq w_j \quad \text{for all } j$$

Now, at a refinery site, z_j can be non-zero only if y_j is non-zero, and w_j is non-zero only if both y_j and z_j are. This partially breaks the symmetry of this formulation, though other symmetries (particularly in the x variables) remain.

This formulation can be modified in another way by redefining the variables. Instead of using binary variables, let y_j be the number of refineries put in location j. This removes all of the symmetries at the cost of a weaker linear relaxation (since some of the strengthenings we have explored require binary variables).

Finally, to illustrate the use of variable fixing, consider the problem of coloring a graph with K colors: we are given a graph with node set V and edge set E and wish to determine if we can assign a value $v(i)$ to each node i such that $v(i) \in \{1, \ldots, K\}$ and $v(i) \neq v(j)$ for all $(i, j) \in E$.

We can formulate this problem as an integer programming by defining a binary variable x_{ik} to be 1 if i is given color k and 0 otherwise. This leads to the constraints

$$\sum_k x_{ik} = 1 \text{ for all } i \text{ (every node gets a color)}$$

$$x_{ik} + x_{jk} = 1 \text{ for all } k, (i, j) \in E \text{ (no adjacent get the same)}$$

$$x_{ik} \in \{0, 1\} \text{ for all } i, k$$

The graph coloring problem is equivalent to determining if the above set of constraints is feasible. This can be done by using branch-and-bound with an arbitrary objective value.

Unfortunately, this formulation is highly symmetric. For any coloring of graph, there is an equivalent coloring that arises by permuting the coloring (that is, permuting the set $\{1, \ldots, k\}$ in this formulation). This makes branch and bound very ineffective for this formulation. Note also that the formulation is very weak, since setting $x_{ik} = 1/k$ for all i, k is a feasible solution to the linear relaxation no matter what E is.

We can strengthen this formulation by breaking the symmetry through variable fixing. Consider a clique (set of mutually adjacent vertices) of the graph. Each member of the clique has to get a different color. We can break the symmetry by finding a large (ideally maximum sized) clique in the graph and

setting the colors of the clique arbitrarily, but fixed. So if the clique has size k_c, we would assign the colors $1, \ldots, k_c$ to members of the clique (adding in constraints forcing the corresponding x values to be 1). This greatly reduces the symmetry, since now only permutations among the colors $k_c + 1, \ldots, K$ are valid. This also removes the $x_{ik} = 1/k$ solution from consideration.

3.5 CONSIDER FORMULATIONS WITH MANY CONSTRAINTS

Given the importance of the strength of the linear relaxation, the search for improved formulations often leads to sets of constraints that are too large to include in the formulation. For example, consider a single constraint with non-negative coefficients:

$$a_1x_1 + a_2x_2 + a_3x_3 + \cdots + a_nx_n \leq b$$

where the x_i are binary variables. Consider a subset S of the variables such that $\sum_{i \in S} a_i > b$. The constraint

$$\sum_{i \in S} x_i \leq |S| - 1$$

is valid (it is not violated by any feasible integer solution) and cuts off fractional solutions as long as S is minimal. These constraints are called *cover constraints*. We would then like to include this set of constraints in our formulation.

Unfortunately, the number of such constraints can be very large. In general, it is exponential in n, making it impractical to include the constraints in the formulation. But the relaxation is much tighter with the constraints.

To handle this problem, we can choose to generate only those constraints that are needed. In our search for an optimal integer solution, many of the constraints are not needed. If we can generate the constraints as we need them, we can get the strength of the improved relaxation without the huge number of constraints.

Suppose our instance is

$$\begin{array}{ll} \text{Maximize} & 9x_1 + 14x_2 + 20x_3 + 32x_4 \\ \text{Subject to} & 3x_1 + 5x_2 + 8x_3 + 10x_4 \leq 16 \\ & x_i \in \{0, 1\} \end{array}$$

The optimal solution to the linear relaxation is $x^* = (1, 0.6, 0, 1)$ with objective 49.4. Now consider the set $S = (x_1, x_2, x_4)$. The constraint

$$x_1 + x_2 + x_4 \leq 2$$

is a cut that x^* violates. If we add that constraint to our problem, we get a tighter formulation. Solving this model gives solution $x = (1, 0, 0.375, 1)$ and objective 48.5. The constraint

$$x_3 + x_4 \leq 1$$

is a valid cover constraint that cuts off this solution. Adding this constraint and solving gives solution $x = (0, 1, 0, 1)$ with objective 46. This is the optimal solution to the original integer program, which we have found only by generating cover inequalities.

In this case, the cover inequalities were easy to see, but this process can be formalized. A reasonable heuristic for identifying violated cover inequalities would be to order the variables by decreasing $a_i x_i^*$ then add the variables to the cover S until $\sum_{i \in S} a_i > b$. This heuristic is not guaranteed to find violated cover inequalities (for that, a knapsack optimization problem can be formulated and solved) but even this simple heuristic can create much stronger formulations without adding too many constraints.

This idea is formalized in the *branch-and-cut* approach to integer programming. In this approach, a formulation has two parts: the *explicit constraints* (denoted $Ax \leq b$) and the *implicit constraints* ($A'x \leq b'$). Denote the objective function as Maximize cx. Here we will assume that all x are integral variables, but this can be easily generalized.

Step 1. Solve the linear program Maximize cx subject to $Ax \leq b$ to get optimal relaxation solution x^*.
Step 2. If x^* integer, then stop. x^* is optimal.
Step 3. Try to find a constraint $a'x \leq b'$ from the implicit constraints such that $a'x^* > b$. If found, add $a'x \leq b$ to the $Ax \leq b$ constraint set and go to step 1. Otherwise, do branch-and-bound on the current formulation.

In order to create a branch-and-cut model, there are two aspects: the definition of the implicit constraints, and the definition of the approach in Step 3 to find violated inequalities. The problem in Step 3 is referred to as the *separation problem* and is at the heart of the approach. For many sets of constraints, no good separation algorithm is known. Note, however, that the separation problem might be solved heuristically: it may miss opportunities for separation and therefore invoke branch-and-bound too often. Even in this case, it often happens that the improved formulations are sufficiently tight to greatly decrease the time needed for branch-and-bound.

This basic algorithm can be improved by carrying out cut generation within the branch and bound tree. It may be that by fixing variables, different constraints become violated and those can be added to the subproblems.

3.6 CONSIDER FORMULATIONS WITH MANY VARIABLES

Just as improved formulations can result from adding many constraints, adding many variables can lead to very good formulations. Let us begin with our graph coloring example. Recall that we are given a graph with vertices V and edges E and want to assign a value $v(i)$ to each node i such that $v(i) \neq v(j)$ for all $(i, j) \in E$. Our objective is to use the minimum number of different values (before, we had a fixed number of colors to use: in this section we will use the optimization version rather than the feasibility version of this problem).

Previously, we described a model using binary variables x_{ik} denoting whether node i gets color k or note. As an alternative model, let us concentrate on the set of nodes that gets the same color. Such a set must be an *independent set* (a set of mutually non-adjacent nodes) of the graph. Suppose we listed all independent sets of the graph: $S_1, S_2, \ldots, S_m$. Then we can define binary variables $y_1, y_2, \ldots, y_m$ with the interpretation that $y_j = 1$ means that independent set S_j is part of the coloring, and $y_j = 0$ means that independent set S_j is not part of the coloring. Now our formulation becomes

$$\begin{aligned} \text{Minimize } & \sum_j y_j \\ \text{Subject to } & \sum_{j: i \in S_j} y_j = 1 \text{ for all } i \in V \\ & y_j \in \{0, 1\} \text{ for all } j \in \{1, \ldots, m\} \end{aligned}$$

The constraint states that every node must be in some independent set of the coloring.

This formulation is a much better formulation that our x_{ik} formulation. This formulation does not have the symmetry problems of the previous formulation and results in a much tighter linear relaxation. Unfortunately, the formulation is impractical for most graphs because the number of independent sets is exponential in the number of nodes, leading to an impossibly large formulation.

Just as we could handle an exponential number of constraints by generating them as needed, we can also handle an exponential number of variables by *variable generation*: the creation of variables only as they are needed. In order to understand how to do this, we will have to understand some key concepts from linear programming.

Consider a linear program, where the variables are indexed by j and the constraints indexed by i:

$$\text{Maximize } \sum_j c_j x_j$$

$$\text{Subject to } \sum_j a_{ij} x_{ij} \leq b_i \text{ for all } i$$

$$x_j \geq 0 \text{ for all } j$$

When this linear program is solved, the result is the optimal solution x^*. In addition, however, there is a value called the *dual value*, denoted π_i, associated with each constraint. This value gives the marginal change in the objective value as the right-hand side for the corresponding constraint is changed. So if the right-hand side of constraint i changes to $b_i + \Delta$, then the objective will change by $\pi_i \Delta$ (there are some technical details ignored here involving how large Δ can be for this to be a valid calculation: since we are only concerned with marginal calculations, we can ignore these details).

Now, suppose there is a new variable x_{n+1}, not included in the original formulation. Suppose it could be added to the formulation with corresponding objective coefficient c_{n+1} and coefficients $a_{i,n+1}$. Would adding the variable to the formulation result in an improved formulation? The answer is certainly "no" in the case when

$$c_{n+1} < \sum_i a_{i,n+1} \pi_i$$

In this case, the value gained from the objective is insufficient to offset the cost charged marginally by the effect on the constraints. We need $c_{n+1} - \sum_i a_{i,n+1} \pi_i > 0$ in order to possibly improve on our solution.

This leads to the idea of variable generation. Suppose you have a formulation with a huge number of variables. Rather than solve this huge formulation, begin with a smaller number of variables. Solve the linear relaxation and get dual values π. Using π, determine if there is one (or more) variables whose inclusion might improve the solution. If not, then the linear relaxation is solved. Otherwise, add one or more such variables to the formulation and repeat.

Once the linear relaxation is solved, if the solution is integer, then it is optimal. Otherwise, branch and bound is invoked, with the variable generation continuing in the subproblems.

Key to this approach is the algorithm for generating the variables. For a huge number of variables it is not enough to check all of them: that would be too time consuming. Instead, some sort of optimization problem must be defined whose solution is an improving variable. We illustrate this for our graph coloring problem.

Suppose we begin with a limited set of independent sets and solve our relaxation over them. This leads to a dual value π_i for each node. For any other independent set S, if $\sum_{i \in S} \pi_i > 1$, then S corresponds to an improving variable. We can write this problem using binary variables z_i corresponding to whether i is in S or not:

$$\text{Maximize } \sum_i \pi_i z_i$$

$$\text{Subject to } z_i + z_j \leq 1 \text{ for all } (i, j) \in E$$

$$z_i \in \{0, 1\} \text{ for all } i$$

This problem is called the *maximum weighted independent set* (MWIS) problem, and, while the problem is formally hard, effective methods have been found for solving it for problems of reasonable size.

This gives a variable generation approach to graph coloring: begin with a small number of independent sets, then solve the MWIS problem, adding in independent sets until no independent set improves the current solution. If the variables are integer, then we have the optimal coloring. Otherwise we need to branch.

Branching in this approach needs special care. We need to branch in such a way that our subproblem is not affected by our branching. Here, if we simply branch on the y_j variables (so have one branch with $y_j = 1$ and another with $y_j = 0$), we end up not being able to use the MWIS model as a subproblem. In the case where $y_j = 0$ we need to find an improving set, except that S_j does not count as improving. This means we need to find the second most improving set. As more branching goes on, we may need to find the third most improving, the fourth most improving, and so on. To handle this, specialized branching routines are needed (involving identifying nodes that, on one side of the branch, must be the same color and, on the other side of the branch, cannot be the same color).

Variable generation together with appropriate branching rules and variable generation at the subproblems is a method known as *branch and price*. This approach has been very successful in attacking a variety of very difficult problems over the last few years.

To summarize, models with a huge number of variables can provide very tight formulations. To handle such models, it is necessary to have a variable generation routine to find improving variables, and it may be necessary to modify the branching method in order to keep the subproblems consistent with that routine. Unlike constraint generation approaches, heuristic variable generation routines are not enough to ensure optimality: at some point it is necessary to prove conclusively that the right variables are included. Furthermore, these variable generation routines must be applied at each node in the branch-and-bound tree if that node is to be crossed out from further analysis.

3.7 MODIFY BRANCH-AND-BOUND PARAMETERS

Integer programs are solved with computer programs. There are a number of computer programs available to solve integer programs. These range from basic spreadsheet-oriented systems to open-source research codes to sophisticated commercial applications. To a greater or lesser extent, each of these codes offers parameters and choices that can have a significant affect on the solvability of integer programming models. For most of these parameters, the only way to determine the best choice for a particular model is experimentation: any choice that is uniformly dominated by another choice would not be included in the software.

Here are some common, key choices and parameters, along with some comments on each.

3.7.1 Description of Problem

The first issue to be handled is to determine how to describe the integer program to the optimization routine(s). Integer programs can be described as spreadsheets, computer programs, matrix descriptors, and higher-level languages. Each has advantages and disadvantages with regards to such issues as ease-of-use, solution power, flexibility and so on. For instance, implementing a branch-and-price approach is difficult if the underlying solver is a spreadsheet program. Using "callable libraries" that give access to the underlying optimization routines can be very powerful, but can be time-consuming to develop.

Overall, the interface to the software will be defined by the software. It is generally useful to be able to access the software in multiple ways (callable libraries, high level languages, command line interfaces) in order to have full flexibility in solving.

3.7.2 Linear Programming Solver

Integer programming relies heavily on the underlying linear programming solver. Thousands or tens of thousands of linear programs might be solved in the course of branch-and-bound. Clearly a faster linear programming code can result in faster integer programming solutions. Some possibilities that might be offered are primal simplex, dual simplex, or various interior point methods. The choice of solver depends on the problem size and structure (for instance, interior point methods are often best for very large, block-structured models) and can differ for the initial linear relaxation (when the solution must be found "from scratch") and subproblem linear relaxations (when the algorithm can use previous solutions as a starting basis). The choice of algorithm can also be affected by whether constraint and/or variable generation are being used.

3.7.3 Choice of Branching Variable

In our description of branch-and-bound, we allowed branching on any fractional variable. When there are multiple fractional variables, the choice of variable can have a big effect on the computation time. As a general guideline, more "important" variables should be branched on first. In a facility location problem, the decisions on opening a facility are generally more important than the assignment of a customer to that facility, so those would be better choices for branching when a choice must be made.

3.7.4 Choice of Subproblem to Solve

Once multiple subproblems have been generated, it is necessary to choose which subproblem to solve next. Typical choices are depth-first search, breadth-first search, or best-bound search. Depth-first search continues fixing variables for a single problem until integrality or infeasibility results. This can lead quickly to an integer solution, but the solution might not be very good. Best-bound search works with subproblems whose linear relaxation is as large (for maximization) as possible, with the idea that subproblems with good linear relaxations may have good integer solutions.

3.7.5 Direction of Branching

When a subproblem and a branching variable have been chosen, there are multiple subproblems created corresponding to the values the variable can take on. The ordering of the values can affect how quickly good solutions can be found. Some choices here are a fixed ordering or the use of estimates of the resulting linear relaxation value. With fixed ordering, it is generally good to first try the more restrictive of the choices (if there is a difference).

3.7.6 Tolerances

It is important to note that while integer programming problems are primarily combinatorial, the branch-and-bound approach uses numerical linear programming algorithms. These methods require a number of parameters giving allowable tolerances. For instance, if $x_j = 0.998$ should x_j be treated as the value 1 or should the algorithm branch on x_j? While it is tempting to give overly big values (to allow for faster convergence) or small values (to be "more accurate"), either extreme can lead to problems. While for many problems, the default values from a quality code are sufficient, these values can be the source of difficulties for some problems.

3.8 TRICKS OF THE TRADE

After reading this tutorial, all of which is about "tricks of the trade", it is easy to throw one's hands up and give up on integer programming! There are so many choices, so many pitfalls, and so much chance that the combinatorial explosion will make solving problems impossible. Despite this complexity, integer programming is used routinely to solve problems of practical interest. There are a few key steps to make your integer programming implementation go well.

- Use state-of-the-art software. It is tempting to use software because it is easy, or available, or cheap. For integer programming, however, not having the most current software embedding the latest techniques can doom your project to failure. Not all such software is commercial. The COIN-OR project is an open-source effort to create high-quality optimization codes.

- Use a modeling language. A modeling language, such as OPL, Mosel, AMPL, or other language can greatly reduce development time, and allows for easy experimentation of alternatives. Callable libraries can give more power to the user, but should be reserved for "final implementations", once the model and solution approached are known.

- If an integer programming model does not solve in a reasonable amount of time, look at the formulation first, not the solution parameters. The default settings of current software are generally pretty good. The problem with most integer programming formulations is the formulation, not the choice of branching rule, for example.

- Solve some small instances and look at the solutions to the linear relaxations. Often constraints to add to improve a formulation are quite obvious from a few small examples.

- Decide whether you need "optimal" solutions. If you are consistently getting within 0.1% of optimal, without proving optimality, perhaps you should declare success and go with the solutions you have, rather than trying to hunt down that final gap.

- Try radically different formulations. Often, there is another formulation with completely different variables, objective, and constraints that will have a much different computational experience.

3.9 CONCLUSIONS

Integer programming models represent a powerful approach to solving hard problems. The bounds generated from linear relaxations are often sufficient

to greatly cut down on the search tree for these problems. Key to successful integer programming is the creation of good formulations. A good formulation is one where the linear relaxation closely resembles the underlying integer program. Improved formulations can be developed in a number of ways, including finding formulations with tight relaxations, avoiding symmetry, and creating and solving formulations that have an exponential number of variables or constraints. It is through the judicious combination of these approaches, combined with fast integer programming computer codes that the practical use of integer programming has greatly expanded in the last 20 years.

SOURCES OF ADDITIONAL INFORMATION

Integer programming has existed for more than 50 years and has developed a huge literature. This bibliography therefore makes no effort to be comprehensive, but rather provides initial pointers for further investigation.

General Integer Programming There are a number of excellent recent monographs on integer programming. The classic is Nemhauser and Wolsey (1988). A book updating much of the material is Wolsey (1998). Schrijver (1998) is an outstanding reference book, covering the theoretical underpinnings of integer programming.

Integer Programming Formulations There are relatively few books on formulating problems. An exception is Williams (1999). In addition, most operations research textbooks offer examples and exercises on formulations, though many of the examples are not of realistic size. Some choices are Winston (1997), Taha (2002), and Hillier and Lieberman (2002).

Branch and Bound Branch and bound traces back to the 1960s and the work of Land and Doig (1960). Most basic textbooks (see above) give an outline of the method (at the level given in this tutorial).

Branch and Cut The cutting plane approach dates back to the late 1950s and the work of Gomory (1958), whose cutting planes are applicable to any integer program. Juenger et al. (1995) provides a survey of the use of cutting plane algorithms for specialized problem classes.

As a computational technique, the work of Crowder et al. (1983) showed how cuts could greatly improve basic branch-and-bound.

For an example of the success of such approaches for solving extremely large optimization problems, see Applegate et al. (1998).

Branch and Price Barnhart et al. (1998) is an excellent survey of this approach.

Implementations There are a number of very good implementations that allow the optimization of realistic integer programs. Some of these are commercial, like the CPLEX implementation of ILOG, Inc. (CPLEX, 2004). Bixby et al. (1999) gives a detailed description of the advances that this software has made.

Another commercial product is Xpress-MP from Dash, with the textbook by Gueret et al. (2002) providing a very nice set of examples and applications.

COIN-OR (2004) provides an open-source initiative for optimization. Other approaches are described by Ralphs and Ladanyi (1999) and by Cordier et al. (1999).

References

Applegate, D., Bixby, R., Chvatal, V. and Cook, W., 1998, On the solution of traveling salesman problems, in: *Proc. Int. Congress of Mathematicians, Doc. Math. J. DMV*, Vol. 645.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P. and Vance, P. H., 1998, Branch-and-price: column generation for huge integer programs, *Oper. Res.* **46**:316.

Bixby, R. E., Fenelon, M., Gu, Z., Rothberg, E. and Wunderling, R., 1999, *MIP: Theory and Practice—Closing the Gap, Proc. 19th IFIP TC7 Conf. on System Modelling*, Kluwer, Dordrecht, pp. 19–50.

Common Optimization INterface for Operations Research (COIN), 2004, at http://www.coin-or.org

Cordier, C., Marchand, H., Laundy, R. and Wolsey, L. A., 1999, bc-opt: a branch-and-cut code for mixed integer programs, *Math. Program.* **86**:335.

Crowder, H., Johnson, E. L. and Padberg, M. W., 1983, Solving large scale zero-one linear programming problems, *Oper. Res.* **31**:803–834.

Gomory, R. E., 1958, Outline of an algorithm for integer solutions to linear programs, *Bulletin AMS* **64**:275–278.

Gueret, C., Prins, C. and Sevaux, M., 2002, *Applications of Optimization with Xpress-MP*, S. Heipcke, transl., Dash Optimization, Blisworth, UK.

Hillier, F. S. and Lieberman, G. J., 2002, *Introduction to Operations Research*, McGraw-Hill, New York.

ILOG CPLEX 9.0 Reference Manual, 2004, ILOG.

Juenger, M., Reinelt, G. and Thienel, S., 1995, *Practical Problem Solving with Cutting Plane Algorithms in Combinatorial Optimization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 111, American Mathematical Society, Providence, RI.

Land, A. H. and Doig, A. G., 1960, An Automatic Method for Solving Discrete Programming Problems, *Econometrica* **28**:83–97.

Nemhauser, G. L. and Wolsey, L. A., 1998, *Integer and Combinatorial Optimization,* Wiley, New York.

Ralphs, T. K. and Ladanyi, L., 1999, *SYMPHONY: A Parallel Framework for Branch and Cut,* White paper, Rice University.

Schrijver, A., 1998, *Theory of Linear and Integer Programming,* Wiley, New York.

Taha, H. A., 2002, *Operations Research: An Introduction*, Prentice-Hall, New York.

Williams, H. P., 1999, *Model Building in Mathematical Programming,* Wiley, New York.

Winston, W., 1997, *Operations Research: Applications and Algorithms,* Thomson, New York.

Wolsey, L. A., 1998, *Integer Programming,* Wiley, New York.

XPRESS-MP Extended Modeling and Optimisation Subroutine Library, Reference Manual, 2004, Dash Optimization, Blisworth, UK.