CHALMERS |  GÖTEBORGS UNIVERSITET

Programvara för undervisning om Branch-and-Bound-metoden

Kandidatarbete inom civilingenjörsutbildningen vid Chalmers

Jonathan Ahlstedt Gyllbrandt
Anna Furberg
Agnes Ramle
Johan Villysson

Institutionen för matematiska vetenskaper
Chalmers tekniska högskola
Göteborgs universitet
Göteborg 2012

Programvara för undervisning om Branch-and-Bound-metoden

Kandidatarbete i matematik inom civilingenjörsprogrammet Kemiteknik med fysik vid Chalmers

Anna Furberg

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers

Jonathan Ahlstedt Gyllbrandt Agnes Ramle Johan Villysson

Handledare: Ann-Brith Strömberg
Examinator: Carl-Henrik Fant

Institutionen för matematiska vetenskaper
Chalmers tekniska högskola
Göteborgs universitet
Göteborg 2012

Sammanfattning

Branch-and-Bound är en viktig algoritm som används i matematisk optimering. Den bygger på tekniken *Divide and Conquer*, där ett problem förgrenas till delproblem som är lättare att lösa. Algoritmen används för att lösa generella heltalsoptimeringsproblem och speciella fall av sådana såsom handelsresandeproblemet.

Flertalet studenter som läser grundläggande kurser i optimering får lära sig grunderna i algoritmen, men utan en visuell representation av hur den fungerar är det förhållandevis svårt. Den här rapporten innehåller en detaljerad beskrivning av den programvara som skapats av författarna och som syftar till att underlätta inlärningen av Branch-and-Bound. Skapandet av ett pedagogiskt verktyg för att lära ut Branch-and-Bound involverar både implementering av algoritmen och att designa ett användarvänligt gränssnitt. Syftet med rapporten är att beskriva hur en programvara för användning i optimeringskurser vid Chalmers tekniska högskola och Göteborgs universitet skapats.

Det resulterande programmet, som löser både allmänna heltalsoptimeringsproblem och handelsresandeproblemet, testades för funktionalitet och användarvänlighet. Sammanfattningsvis har programvaran utvecklats som ursprungligen planerats i projektstarten och uppfyller i stort sätt våra förväntningar.

Abstract

Branch-and-Bound is an important algorithm used in mathematical optimization. It is based on the technique of *Divide and Conquer*, dividing a problem into subproblems that are easier to solve. The algorithm is used to solve general integer linear programming problems, ILP-problems, and special cases of such problems like the Traveling Salesman Problem.

Many students who study a basic course in optimization are required to learn the basics of the algorithm, but without a visual representation of how it works it is quite hard. This paper contains a detailed description of a program, created by the authors, designed to ease the difficulty of learning this particular algorithm. Creating a pedagogical tool for learning Branch-and-Bound involves both implementing the algorithm and designing a user friendly program interface. The purpose of this report is to describe how we made a standalone application for use in optimization courses at Chalmers University of Technology and the University of Gothenburg.

The resulting program, solving both general ILP problems and the Traveling Salesman Problem, was tested for functionality and usability. In conclusion, the software was designed as initially intended and fullfills most of our expectations.

Innehåll

Förord

Den här rapporten är en gemensam produkt av Anna Furberg, Jonathan Ahlstedt Gyllbrandt, Agnes Ramle och Johan Villysson. Tillika är den programvara som beskrivs i rapporten resultatet av det projekt som utförts av ovanstående medlemmar. Nedan listas vilka delar som ska tillskrivas respektive författare. De olika avsnitten har dock bearbetats av gruppens samtliga medlemmar och ska därför inte enbart ses som en enskild persons arbete. En mer omfattande förteckning än den som listas här finns i de loggböcker som respektive författare fört under arbetets gång samt i den gemensamma dagboken.

Rapportens olika delar

Under skrivandet av den här rapporten har varje författare kunnat gå in och redigera texten i alla kapitel. Dock ska det största arbetet med respektive avsnitt tillskrivas individuella personer. Härefter följer en förteckning över vem som är huvudförfattare för respektive del.

Sammanfattning - Anna Furberg, Agnes Ramle, Johan Villysson
Abstract - Jonathan Ahlstedt, Anna Furberg, Agnes Ramle, Johan Villysson
Förord - Jonathan Ahlstedt, Agnes Ramle
Inledning - Johan Villysson
- *Syfte* - Johan Villysson
- *Begränsingar* - Agnes Ramle
- *Arbetsgång* Johan Villysson
- *Rapportupplägg* - Agnes Ramle

Teori - Agnes Ramle
- *Simplexmetoden för lösning av linjärprogrammeringsproblem* - Jonathan Ahlstedt
- *Heltalsoptimering* - Johan Villysson
  - *Handelsresandeproblemet* - Agnes Ramle
  - *Kappsäcksproblemet* - Anna Furberg
- *Branch-and-Bound* - Agnes Ramle
  - *Branch-and-Bound-algoritmen för linjära heltalsoptimeringsproblem* - Anna Furberg
  - *Branch-and-bound-exempel för ett linjärt heltalsoptimeringsproblem* - Anna Furberg
  - *Branch-and-Bound-tekniker för handelsresandeproblemet* - Agnes Ramle

Implementering, programupplägg och testning - Agnes Ramle
- *Vår implementering av Branch-and-Bound för heltalsoptimeringsproblem* - Agnes Ramle
- *Vår implementering av Branch-and-Bound för handelsresandeproblemet* - Agnes Ramle
- *Upplägg av huvudprogramfönstret ur ett pedagogiskt perspektiv* - Agnes Ramle
- *Testing av Branch-and-Bound Illustrator* - Agnes Ramle
  - *Korrekthet* - Agnes Ramle
  - *Användarvänlighet* - Jonathan Ahlstedt

Diskussion och slutsats
- *Diskussion*
  - *Val vid implementering* - Jonathan Ahlstedt
  - *Begränsningar i programvaran* - Jonathan Ahlstedt, Agnes Ramle, Johan Villysson
  - *Pedagogiskt resonemang* - Johan Villysson, Anna Furberg
  - *Förbättrings- och utvecklingspotential* - Jonathan Ahlstedt, Johan Villysson
- *Slutsatser* - Johan Villysson

Appendix
- *Programstruktur* - Jonathan Ahlstedt
- *Tekniska aspekter kring implementeringen* - Johan Villysson
  - *Branch-and-Bound-trädet* - Johan Villysson
  - *Översiktskarta* - Johan Villysson
  - *Nodspecifik information - graf för heltalsoptimeringsproblem* - Johan Villysson
  - *Nodspecifik information - graf för handelsresandeproblemet* - Agnes Ramle
- *Att skapa en webbapplikation av vår programvara* - Johan Villysson

Programvarans olika delar

För klasserna[1] i programvaran finns, precis som för rapporten, en eller ett par personer som lagt grunden till just den delen. I vissa klasser har bara en medlem ur gruppen bidragit till koden medan det med andra klasser varit en mer dynamisk process där kod lagts till efter hand av olika personer. Den person som ska tillskrivas det största arbetet för respektive klass är fetmarkerad i listan nedan. Därefter följer övriga författare i fallande ordning efter arbetsinsats i den mån det finns flera skapare.

`SelectionFrameController` - **Agnes Ramle**, Jonathan Ahlstedt.
`WelcomeFrame` - **Agnes Ramle**.
`ILPFrame` - **Agnes Ramle**, Jonathan Ahlstedt.
`ILPProblemInputFrame` - **Agnes Ramle**.
`ILPOwnProblemFrame` - **Agnes Ramle**.
`TSPProblemInputFrame` - **Agnes Ramle**.
`TSPOwnProblemFrame` - **Agnes Ramle**.
`OpenFileFrame` - **Jonathan Ahlstedt**.
`SaveFileFrame` - **Jonathan Ahlstedt**.
`MainFrame` - **Johan Villysson**, Agnes Ramle, Jonathan Ahlstedt.
`MenuBarForMainFrame` - **Agnes Ramle**, Jonathan Ahlstedt.
`MainFrameController` - **Johan Villysson, Agnes Ramle**, Jonathan Ahlstedt.
`NodeButton` - **Jonathan Ahlstedt**, Agnes Ramle, Johan Villysson.
`NodeEdge` - **Agnes Ramle, Jonathan Ahlstedt**.
`TreePainter` - **Jonathan Ahlstedt, Agnes Ramle**, Johan Villysson.
`ScreenImage` - Klass hämtad från `http://tips4java.wordpress.com/2008/10/13/screen-image/`, skriven av Rob Camick [1].
`HowToUseTheProgramFrame` - **Agnes Ramle**.
`SearchMethodInformationFrame` - **Agnes Ramle**.
`Index` - **Agnes Ramle**.
`ILPExplanationFrame` - **Anna Furberg**, Agnes Ramle.
`TSPExplanationFrame` - **Anna Furberg**.
`ILPPlotPanel` - **Johan Villysson**, Jonathan Ahlstedt.
`BranchAndBoundILP` - **Agnes Ramle**, Jonathan Ahlstedt.
`BranchAndBoundNodeILP` - **Agnes Ramle**, Jonathan Ahlstedt.
`LPFormulation` - **Agnes Ramle, Anna Furberg**, Jonathan Ahlstedt.
`ILPSolver` - **Jonathan Ahlstedt**, Johan Villysson, Agnes Ramle.
`BranchAndBoundTSP` - **Agnes Ramle**.
`BranchAndBoundNodeTSP` - **Agnes Ramle**.
`Arc` - **Agnes Ramle**.
`TSPGraph` - **Agnes Ramle**, Jonathan Ahlstedt.
`BranchAndBoundInterface` - **Agnes Ramle**.
`BranchAndBoundNodeInterface`- **Agnes Ramle**.

[1] I appendix A finns en detaljerad lista över varje klass ansvarsområde. Här ska också nämnas att klasserna har tagit olika lång tid att programmera.

Inledning

År 1960 publicerade A. H. Land och A. G. Dakin en artikel i tidsskriften Econometrica om en automatisk metod för att lösa diskreta programmeringsproblem [2]. I arbetet formulerade de en algoritm för att numeriskt lösa linjära optimeringsproblem med heltalskrav på vissa variabler. Arbetet lade grunden för den algoritmtyp som idag kallas Branch-and-Bound. Idén går ut på att förenkla och förgrena ursprungsproblemet i delproblem som i sig är lättare att lösa med andra numeriska eller analytiska metoder. Denna förgrening fortgår till dess att ett optimum för ursprungsproblemet som uppfyller de ursprungliga villkoren har hittats och verifierats.

Ett exempel på ett problem som kan lösas med hjälp av Branch-and-Bound är kappsäcksproblemet, se avsnitt 6.2. Tanken bakom kappsäcksproblemet är att man har en kappsäck som ska fyllas med ett antal objekt av olika storlekar samt med olika sorters värde. Problemet går ut på att välja ut rätt objekt för att optimera värdet i kappsäcken. Strukturen kan hittas i flera vardagliga och industriella problem såsom investeringar i värdepapper eller paketering av råvarumaterial. Det gemensamma för dessa problem är att det finns ett binärt villkor på de variabler, det vill säga objekt, som representerar föremål som antingen packas ner i kappsäcken eller inte.

Branch-and-Bound är skräddarsydd för linjära optimeringsproblem med heltalsvillkor, även om den idag ofta används i kombination med andra lösningsmetoder och heuristiker. Den har även visat sig vara effektiv på problem som uppvisar samma karaktär som heltalsoptimeringsproblemen. Ett exempel på en speciell typ av heltalsoptimeringsproblem där Branch-and-Bound visar sig fungera bra är handelsresandeproblemet. I problemet ska en handelsresande utgå från sin hemstad och besöka ett antal städer i en ordning som minimerar restiden samtidigt som den handelsresande inte får besöka samma stad mer än en gång och dessutom måste återvända hem igen. Problemstrukturen uppkommer också ofta inom industrin, till exempel inom logistik[2] och vid tillverkningen av kretskort[3] och därför är Branch-and-Bound en intressant lösningsalgoritm att studera.

1 Syfte

Ur ett pedagogiskt perspektiv är Branch-and-Bound-algoritmen en metod som enklast förklaras illustrativt. Stegen i algoritmen kan visualiseras i ett träddiagram där varje steg i metoden kan följas tills en optimallösning hittas.

Syftet med det här projektet är att skapa ett enkelt och lärorikt program som underlättar i undervisningen av Branch-and-Bound-metoden vid Chalmers tekniska högskola och Göteborgs universitet. Målgruppen för programmet är studenter som läser en optimeringskurs innehållandes linjärprogrammering där vi förväntar oss att användaren tidigare har hört talas om Branch-and-Bound-metoden för att få ut maximalt av programmet. Målet med rapporten är att beskriva hur vi gått tillväga för att skapa ett interaktivt läroverktyg som stegvis förklarar och illusterar varje iteration i Branch-and-Bound-algoritmen.

2 Begränsningar

Vi kommer i den här rapporten presentera den matematiska formuleringen av ett antal olika optimeringsproblem. För två av dessa, heltalsoptimeringsproblem med linjär målfunktion och linjära bivillkor samt specialfallet av dessa handelsresandeproblemet, kommer vi också beskriva hur Branch-and-Bound-algoritmen kan användas på olika sätt för att hitta en optimallösning. För det tredje problemet, kappsäcksproblemet, presenteras dock ingen specifik Branch-and-Bound-teknik.

[2] DPS, Göteborg [3].
[3] Philips, Norrköping (1988) [4].

Av de presenterade Branch-and-Bound-teknikerna implementerades sedan bara två, mer rymdes inte inom ramen för det här kandidatarbetet. Hur implementeringen gjordes finns beskrivet i avsnitt 8–9.

3 Arbetsgång

Vid projektets början krävdes en del teoristudier om Branch-and-Bound. Eftersom gruppens medlemmar hade olika kunskapsbakgrund, där vissa hade läst optimeringskurser innehållandes Branch-and-Bound innan och vissa inte, blev det första steget att studera algoritmerna och teorin bakom dem innan vi satte igång med implementeringen. Dessa teoristudier fortsatte sedan kontinuerligt under arbetets gång vid sidan av programmeringen.

I början av programmeringsarbetet bestämde vi oss för att använda utvecklingsverktyget *Eclipse* [5]. Fördelen med att använda ett utvecklingsverktyg såsom Eclipse är möjligheten till hjälp med syntaxen och buggtestningen av programmet. Till exempel erbjuds en funktion, *auto-fill*, som automatiskt kan fylla i resterna av ett metodnamn eller ett ord för att göra dem fullständiga och korrekta. Detta är annars något som ofta resulterar i mindre fel vid kompilering av koden. Utöver många hjälpfunktioner för syntax finns även inbyggda rutiner för refaktorering av kod, exempelvis möjligheten att skapa ett `Interface` av en befintlig klass.

Programmeringen var också den del av arbetet som krävde mest timmar under projektets gång. Tillvägagångsättet under denna fas bestod mycket i ”Trial and Error”, alltså testades många olika idéer som ofta resulterade i små fel som fick korrigeras efter hand som de upptäcktes. Vi utnyttjade även den kunskapsbas som finns tillgänglig via internet. Oftast använde vi Google för att få fram forumtrådar där personer hade haft samma eller liknande problem som vi stött på och där erfarna programmerare erbjudit tips och svar på dessa frågor. Dessa tips fungerade sedan som en inspiration vid kodningen men även som uppfräschning av gamla kunskaper. Vi vill dock nämna att vi vid uppbyggnaden av Branch-and-Bound-algoritmerna inte utgick från några färdiga kodpaket eller andra implementationer.

Det krävdes även omfattande testning och felsökning under kodningsfasen. Testningen innefattade inte bara korrekthet av Branch-and-Bound-implementationen utan även testning av programvaran på studenter för att säkerställa tillräckligt hög användarvänlighet.

Under hela projektets gång har vi även samtidigt skrivit på denna rapport. För att effektivisera skrivproceduren delade vi till stor del upp arbetet bland gruppens medlemmar men varvade även detta med att skriva i grupp eller i par. All den kod och text som skrevs på olika håll krävde ett bra system för att sammanfoga arbetet. Till detta använde vi *SVN*, subversion, som är ett versionshanteringssystem som håller reda på vem som uppderat vilken fil och när detta skedde. Dessa filer behöver samlas på en server, i vårt fall blev det Fysikteknologsektionens, för att kunna ladda upp och ladda hem arbetet både hemifrån och från Chalmers datorer.

Till sist har vi även dokumenterat allt arbete i en gruppdagbok samt haft regelbundna möten med vår handledare Ann-Brith Strömberg, då vi har tillhandahållit rapporttexter och vårt program för att få kommentarer på det arbete som har gjorts.

4 Upplägg av rapporten

I den här rapporten kommer först den bakgrundsteori som krävs för att förstå och kunna implementera Branch-and-Bound att presenteras. Teoriavsnittet börjar med en beskrivning av *simplexalgoritmen*, en teknik för att hitta optimum till linjärprogrammeringsproblem, en vanlig metod för att hitta en lösning till delproblem som uppstår vid användning av Branch-and-Bound. Därefter följer en presentation av tre vanliga varianter av optimeringsproblem. Sist i avsnittet beskrivs olika Branch-and-Bound-tekniker för generella heltalsoptimeringsproblem och för specialfallet handelsresandeproblemet.

Efter teorikapitlet följer sedan en redogörelse för hur vi gått tillväga i implementationen av några av de algoritmer som beskrivits i kapitlet innan. Avsnittet innehåller också en beskrivning av vårt programs, *Branch-and-Bound Illustrators*, huvudprogramfönster ur det pedagogiska perspektivet samt slutligen ett avsnitt om programvarans testning.

Rapporten fortsätter därefter med en diskussion kring val och problem som uppstått under kandidatarbetets gång. Avslutningsvis bifogas ett appendix med bland annat programmets struktur, det vill säga vilka klasser som skrivits för att bygga upp programvaran, och tekniska aspekter kring implementeringen. Sist men inte minst följer i appendix ett avsnitt om hur en webbapplikation kan skapas av vår programvara.

Teori

Först i det här kapitlet kommer vi att introducera en lösningsmetod för linjärprogrammeringsproblem, *simplex*, som inte är direkt kopplad till Branch-and-Bound men som ändå används vid lösning av många optimeringsproblem som ett steg i Branch-and-Bound-algoritmen. Därefter kommer vi att gå in på matematiska formuleringar av optimeringsproblem som alla kan lösas med hjälp av Branch-and-Bound. De vi har valt ut är i tur och ordning, generella heltalsoptimeringsproblem, ofta förkortade med ILP, handelsresandeproblem och kappsäcksproblem.

Slutligen beskriver vi hur Branch-and-Bound-algoritmen ser ut mer i detalj för de olika problemtyperna heltalsoptimeringsproblem och handelsresandeproblem.

5 Simplexmetoden för lösning av linjärprogrammeringsproblem

I det här avsnittet kommer grunderna för hur simplexmetoden fungerar att presenteras. Simplexmetoden, eller simplexalgoritmen som den också kallas, är en effektiv metod för att snabbt hitta optimum till ett av de vanligaste problemen inom optimeringslära, linjärprogrammeringsproblem. I praktiken är det ofta simplexmetoden[4], eller någon mindre modifiering av den, som används när man löser linjärprogrammeringsproblem, eller LP-problem som det ofta förkortas. Ett LP-problem är ett minimerings- eller maximeringsproblem av en *linjär* funktion över ett område som definieras av *linjära* bivillkor.

Simplexalgoritmen fungerar bara för LP-problem på standardform [7, sid. 80 ff.], det vill säga uttryckta som att:

$$\begin{aligned} &\text{maximera } z = \sum_{j=1}^{n} c_j \; x_j, \\ &\text{då} \qquad \sum_{j=1}^{n} a_{ij} \; x_j = b_i, \quad i = 1, 2, 3, ..., m, \\ &\qquad\qquad x_j \geq 0, \quad j = 1, 2, 3, ..., n. \end{aligned}$$

I de flesta fall anges generella LP-problem dock inte på standardform från början. Det kan finnas variabler, x_k, som har andra begränsningar än icke-negativitet, såsom $x_k \leq 0$ eller att x_k tillåts anta vilket reellt värde som helst. Istället för likhet i andra raden ovan kan det för vissa i gälla att $\sum_{j=1}^{n} a_{ij}x_j \geq b_i$ eller $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$. Att inte starta med ett LP-problem uttryckt på standardform är dock inget större problem då det alltid går att, med olika tekniker, skriva om problemet till standardform.

Ett viktigt begrepp för att förstå simplexmetoden är baslösningar. En tillåten baslösning är en lösning, det vill säga en punkt $\mathbf{x}$, som uppfyller alla bivillkor för LP-problemet och som dessutom är en hörnpunkt i den polyeder som utgör området som ska optimeras över.

För att hitta optimum till ett LP-problem med simplexalgoritmen går man i varje steg från ett hörn i lösningsmängden till ett annat hörn med ett bättre värde på målfunktionen. I varje steg förbättras målfunktionsvärdet vilket gör att algoritmen konvergerar mot ett optimalt hörn i lösningsmängden. Minst en av optimallösningarna[5] till problemet kommer att återfinnas i detta hörn enligt *Linjärprogrammmeringens huvudsats*, se Optimeringslära ([7, sid. 80]).

Om det från början inte finns någon given tillåten baslösning kan simplexmetoden användas för att hitta en. För att finna den första tillåtna baslösningen löses ett nytt LP-problem som fås genom att skriva om startproblemet med hjälp av så kallade artificiella variabler, se

[4] Vanligt är också så kallade inre punkts-metoder, se vidare konferenspublikationen A new polynomial-time algorithm for linear programming [6]

[5] Det kan finnas flera punkter med samma optimalvärde.

Optimeringslära ([7, sid. 100–103]). Det nya problemet är utformat så att dess optimallösning är en tillåten baslösning till startproblemet. Med andra ord, om det saknas en tillåten baslösning löser vi först ett nytt LP-problem med simplexmetoden för att hitta en tillåten bas så att vi sedan kan lösa ursprungsproblemet med hjälp av simplexmetoden.

5.1 Ett exempel med visuell beskrivning

För ett problem med mer än tre variabler blir en visuell förklaring av simplexmetoden ganska komplex även om lösningstekniken är densamma oavsett antal variabler. Det är till exempel svårt att grafiskt visa ett hörn på ett fyrdimensionellt objekt. Med två eller tre variabler är det däremot mycket lättare att utifrån en visuell beskrivning se hur algoritmen fungerar. Nedan följer en illustrering av hur algoritmen fungerar för ett exempel med två variabler.

Låt oss betrakta problemet att

$$\begin{aligned} \text{maximera } z = \quad & x_1 + 2x_2, \\ \text{då} \qquad\quad & -x_1 + x_2 \leq 2, \\ & -x_1 + 2x_2 \leq 5, \\ & 4x_1 + x_2 \leq 16, \\ & x_j \geq 0, \quad j = 1, 2, \end{aligned}$$

för vilket det är lätt att hitta en startlösning i origo eftersom problemet är ett maximeringsproblem som bara har mindre än eller lika med villkor där alla $x_j \geq 0$. Om vi skriver om problemet på standardform får vi,

$$\begin{aligned} \text{maximera } z = \quad & x_1 + 2x_2, \\ \text{då} \qquad\quad & -x_1 + x_2 + s_1 = 2, \\ & -x_1 + 2\,x_2 + s_2 = 5, \\ & 4x_1 + x_2 + s_3 = 16, \\ & x_j \geq 0, \quad j = 1, 2, \\ & s_k \geq 0, \quad k = 1, 2, 3. \end{aligned}$$

Det omformulerade problemet är ett problem med fem variabler där bara två av variablerna återfinns i målfunktionen. Vi har lagt till en variabel, s_k, per bivillkor. Dessa variabler kallas för *slackvariabler* och är hjälpvariabler för att omvandla ett mindre-än-eller-lika-med eller ett större-än-eller-lika-med-villkor till ett lika-med-villkor. För att beskriva hörnpunkten i origo med hjälp av denna nya bas så antar slackvariablerna värdena $s_1 = 2, s_2 = 5$ och $s_3 = 16$, det vill säga vi har lösningsvektorn [0, 0, 2, 5, 16].

Vi introducerar nu en så kallad *simplextablå*,

$\mathbf{x}_B$	z	x_1	x_2	s_1	s_2	s_3	$\bar{\mathbf{b}}$
z	1	-1	-2				0
s_1		-1	1	1			2
s_2		-1	2		1		5
s_3		4	1			1	16

.

Tablån används för att beskriva stegen i simplexalgoritmen på ett mer kompakt sätt, där kolumnen längst till vänster beskriver vilka variabler som ligger i basen. Kolumnen längst till höger beskriver hur lösningsvektorn ser ut samt vilket z-värde vi har i nuvarande steg där vi i det här fallet börjar med $z = 0$. Anledningen till att objektfunktionen har ett negativ värde är att vi skrivit om målfunktionen $z = x_1 + 2x_2$ som $z - x_1 - 2x_2 = 0$.

I andra steget av algoritmen ska vi först hitta den variabel som ska ingå i den nya basen och sedan vilken variabel som ska ta utgå ur den gamla. Vi betraktar andra raden i tablån, målfunktionens konstanter eller reducerade kostnader som det också kallas, och ser att vi har negativa konstanter framför både x_1 och x_2. För ett maximeringsproblem betyder det att för varje steg vi tar i x_1- eller x_2-riktning kommer vi att öka målfunktionens värde med 1 respektive 2. Om alla koefficienter däremot hade varit positiva konstanter hade den nuvarande

basen varit en optimallösning i och med att ingen av variablerna som skulle kunna inkluderas i en ny bas hade förbättrat lösningen. För minimeringsproblem letar vi istället efter positiva konstanter för att hitta bättre hörn och har en optimallösning då vi enbart har negativa konstanter.

För maximeringsproblem väljer vi den variabel vars koefficient är den minsta i målfunktionsraden som variabel att ta in i basen. Detta för att alltid försöka maximera målfunktionen z så mycket som möjligt i varje steg. För exempelproblemet innebär det att x_2 blir variabeln att ta in i basen då $-2 < -1$.

När ingående variabel är bestämd ska vi avgöra vilken av s_1, s_2, s_3 som ska ut ur basen. Vi beräknar därför vilket bivillkor som först kommer att begränsa ökningen av x_2 genom att dela varje villkors x_2-koefficient med värdet på dess högerled och därefter ta fram det minst icke-negativa värdet bland dessa. I det här fallet ger $\min\left\{\frac{2}{1}, \frac{5}{2}, \frac{16}{1}\right\} = 2$ att s_1 blir den variabel som ska ut ur basen. Med radreduktion räknar vi sedan ut tablån för nästa steg,

$\mathbf{x}_B$	z	x_1	x_2	s_1	s_2	s_3	$\bar{\mathbf{b}}$
z	1	-3		2			4
x_2		-1	1	1			2
s_2		1		-2	1		1
s_3		5		-1		1	14

.

Med hjälp av samma metod som innan ser vi att $\min\{-3, 2\} = -3$ ger att x_1 ska in i basen och att $\min\left\{\frac{1}{1}, \frac{14}{5}\right\} = 1$ ger att s_2 ska ut ur basen. Detta ger tablån,

$\mathbf{x}_B$	z	x_1	x_2	s_1	s_2	s_3	$\bar{\mathbf{b}}$
z	1			-4	3		7
x_2			1	-1	1		3
x_1		1		-2	1		1
s_3				9	-5	1	9

.

Med samma tillvägagångssätt som innan får vi av $\min\{-4, 3\} = -4$ att s_1 ska in basen och av $\min\left\{\frac{9}{9}\right\} = 1$ att s_3 ska ut. Tablån blir då,

$\mathbf{x}_B$	z	x_1	x_2	s_1	s_2	s_3	$\bar{\mathbf{b}}$
z	1				$\frac{7}{9}$	$\frac{4}{9}$	11
x_2			1		$\frac{4}{9}$	$\frac{1}{9}$	4
x_1		1			$\frac{-1}{9}$	$\frac{2}{9}$	3
s_1				1	$\frac{-5}{9}$	$\frac{1}{9}$	1

.

Nu kan vi se att alla värden i objektfunktionsraden är positiva, vilket innebär att vi har hittat optimum i punkten [3, 4, 1, 0, 0]. Det betyder att lösningsvektorn i de ursprungliga variablerna blir [3, 4] och optimalvärdet 11. Hur de fyra stegen ter sig rent grafiskt kan ses i Figur 1.

5.2 Algoritmbeskrivning

Följande är en stegvis algoritmbeskrivning av simplexmetoden för generella LP-Problem [7].

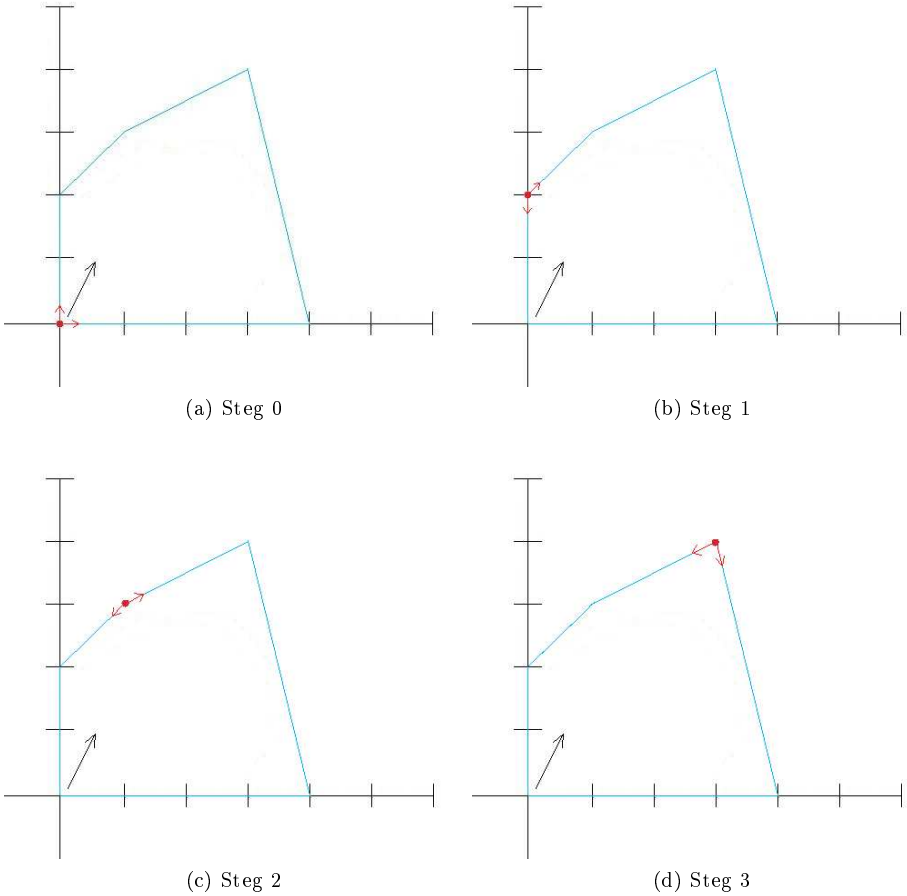
Steg 0 : Hitta en möjlig startlösning och sätt denna till $\mathbf{x}^{(0)}$. Sätt sedan stegnumret till k = 0.

Steg 1 : Beräkna de reducerade kostnaderna, $\bar{c}_j$, för basen $\mathbf{x}^k$.

Steg 2 : Kontrollera sedan om en optimallösning $\mathbf{x}^k$ hittats genom att se om

$$\begin{cases} \bar{c}_j \geq 0, & \forall j, \quad \text{om minimeringsproblem} \\ \bar{c}_j \leq 0, & \forall j, \quad \text{om maximeringsproblem.} \end{cases}$$

Om så är fallet kan vi avbryta itereringen här eftersom optimum är hittat. Optimalvärdet representeras då av vektorn $\mathbf{x}^k$.



(a) Steg 0 (b) Steg 1

(c) Steg 2 (d) Steg 3

Figur 1: Bilder av de fyra steg som krävs för att hitta optimum på det givna området, där den svarta pilen beskriver i vilken riktning målfunktionens värde ökar.

Steg 3 : Vi beräknar sedan vilken variabel som ska in i basen med hjälp av,

$$\begin{cases} \bar{c}_{in} = \min\limits_j \{\bar{c}_j \,|\bar{c}_j \leq 0\} & \text{om minimeringsproblem} \\ \bar{c}_{in} = \max\limits_j \{\bar{c}_j \,|\bar{c}_j \geq 0\} & \text{om maximeringsproblem} \end{cases}$$

Detta ger att x_p, variabeln med koefficient $\bar{c}_{in}$, ska in i basen och dessutom att vi får sökriktningen[6] $\mathbf{d}^k$ genom att uttrycka basvariablerna som funktioner av icke-basvariablerna och sedan ta ut vektorn framför den variabel som ska in i basen.

Steg 4 : Vi kan nu beräkna steglängd och vilken variabel som ska ut ur basen genom

$$t^k = \min_j \left\{ \tfrac{x_j^k}{-d_j^k} \mid d_j^k < 0 \right\}.$$

Formeln ovan ger att den variabel, x_j, som först begränsas är den variabel som går in i basen. Vidare har vi att problemet är obegränsat om alla d_j^k är icke-negativa.

Steg 5 : Sätt nu $\mathbf{x}^{k+1} = \mathbf{x}^k + t^k\mathbf{d}^k$ för att få basen i nästa steg och uppdatera $k = k + 1$. Gå tillbaka till steg 1.

[6] Exempelvis skulle sökriktning i första steget för exemplet innan bli [0, 1, -1, -2, -1].

5.3 Degeneration

Ett problem man måste ta hänsyn till vid användande av simplexalgoritmen är degeneration. Degeneration innebär att minst en av basvariablerna till en lösning antar värdet 0, vilket är samma sak som det finns en nolla i $\bar{\mathbf{b}}$-kolumnen i simplextablån. Om en sådan basvariabel väljs som utgående variabel i ett steg av algoritmen kommer målfunktionens värde inte att förändras. Risken är att algoritmen då går in i en cykel där man snurrar runt bland några lösningar med samma värde utan att komma vidare. Som ett exempel kan vi betrakta följande problem hämtat från powerpointpresentationen, Simplex Method : technique aspects [8]:

$$\begin{aligned}
\text{maximera } z = \quad & x_1 - 2x_2 + x_3, \\
\text{då} \quad & -2x_1 + x_2 - x_3 \geq 0, \\
& -3x_1 - x_2 - x_3 \geq 0, \\
& 5x_1 - 3x_2 + 2x_3 \geq 0, \\
& x_j \geq 0, \quad j = 1, 2, 3.
\end{aligned}$$

Om detta problem sätts upp med en simplextablå får vi att,

$\mathbf{x}_B$	z	x_1	x_2	x_3	x_4	x_5	x_6	$\bar{\mathbf{b}}$
z	1	-1	2	-1				0
x_4		-2	1	-1				0
x_5		-3	-1	-1				0
x_6		5	-3	2				0

.

De följande sex iterationerna blir då,

$\mathbf{x}_B$	z	x_1	x_2	x_3	x_4	x_5	x_6	$\bar{\mathbf{b}}$
z	1		$-\frac{7}{3}$	$\frac{2}{3}$		$-\frac{1}{3}$		0
x_4			$\frac{5}{3}$	$-\frac{1}{3}$		$\frac{2}{3}$		0
x_1			$-\frac{1}{3}$	$-\frac{1}{3}$		$-\frac{1}{3}$		0
x_6			$-\frac{14}{3}$	$\frac{1}{3}$		$-\frac{5}{3}$		0

,

$\mathbf{x}_B$	z	x_1	x_2	x_3	x_4	x_5	x_6	$\bar{\mathbf{b}}$
z	1		1		-2	1		0
x_3			5		-3	2		0
x_1			-2		1	-1		0
x_6			-3		-1	-1		0

,

$\mathbf{x}_B$	z	x_1	x_2	x_3	x_4	x_5	x_6	$\bar{\mathbf{b}}$
z	1				$-\frac{7}{3}$	$\frac{2}{3}$	$-\frac{1}{3}$	0
x_3					$-\frac{14}{3}$	$\frac{1}{3}$	$-\frac{5}{3}$	0
x_1					$\frac{5}{3}$	$-\frac{1}{3}$	$\frac{2}{3}$	0
x_2					$-\frac{1}{3}$	$-\frac{1}{3}$	$-\frac{1}{3}$	0

,

$\mathbf{x}_B$	z	x_1	x_2	x_3	x_4	x_5	x_6	$\bar{\mathbf{b}}$
z	1	-2			1		1	0
x_3		-1			-3		-1	0
x_5		-3			5		2	0
x_2		1			-2		-1	0

,

$\mathbf{x}_B$	z	x_1	x_2	x_3	x_4	x_5	x_6	$\bar{\mathbf{b}}$
z	1	$-\frac{7}{3}$		$-\frac{1}{3}$			$\frac{2}{3}$	0
x_4		$-\frac{1}{3}$		$-\frac{1}{3}$			$-\frac{1}{3}$	0
x_5		$-\frac{14}{3}$		$-\frac{5}{3}$			$\frac{1}{3}$	0
x_2		$\frac{5}{3}$		$\frac{2}{3}$			$-\frac{1}{3}$	0

,

och

$\mathbf{x}_B$	z	x_1	x_2	x_3	x_4	x_5	x_6	$\bar{\mathbf{b}}$
z	1	-1	2	-1				0
x_4		-2	1	-1				0
x_5		-3	-1	-1				0
x_6		5	-3	2				0

.

I detta exempel är varje steg degenererat, så det är viktigt att notera att tablån i den sista iterationen är densamma som tablån i den första. Vi har alltså kommit in i en oändlig loop. Det finns olika lösningar man kan implementera i algoritmen för att undvika degenerering.

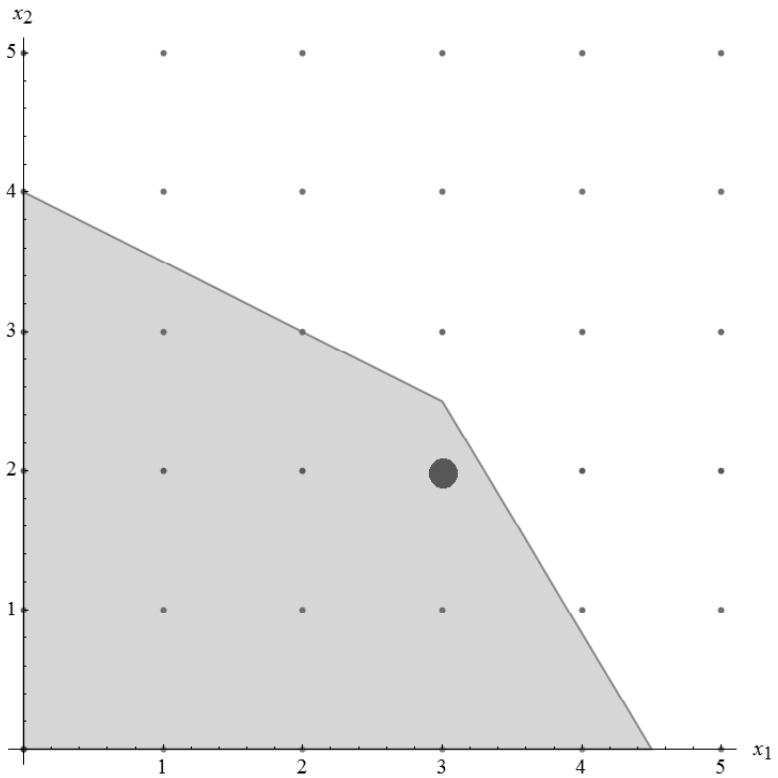
Ett exempel är Blands regel [9] för att välja ingående och utgående variabler vid beräkning med simplextablån. Vad regeln åstadkommer i vårt fall är att vid val av ingående variabel väljs istället en med positiv reducerad kostnad med så lågt index j, som möjligt. Det betyder att exempelvis x_1 väljs före x_2. När utgående variabel bestäms beräknar vi för varje rad i kvoten $\frac{\bar{\mathbf{b}}_i}{a_{ij}}$, där a_{ij} är index på elementen under målfunktionsraden i simplextablån, och väljer variabeln på raden med lägst kvot. Om flera rader har samma kvot väljer vi den med lägst index i.

6 Heltalsoptimering

Ett klassiskt skolboksexempel där Branch-and-Bound används är vid lösning av LP-problem, linjärprogrammeringsproblem, med heltalskrav på vissa eller alla variabler. Benämningen på sådana problem är ILP eller MILP som står för *Integer Linear Programming* respektive *Mixed Integer Linear Programming* där *Mixed* betecknar att inte alla variabler har heltalskrav på sig. I fallet med heltalsvillkor på vissa eller alla variabler får problemet en intressant karaktär som kräver en helt annan typ av lösningsmetod än för LP-problem. Nedan följer ett tvådimensionellt exempel på ett ILP-problem hämtat ur boken Optimeringslära ([7]):

$$\begin{aligned} &\text{minimera } z = -8x_1 - 6x_2, \\ &\text{då} \qquad\qquad x_1 + 2x_2 \leq 8, \\ &\qquad\qquad 10x_1 + 6x_2 \leq 45, \\ &\qquad\qquad x_1,\, x_2 \geq 0, \qquad \text{heltal.} \end{aligned} \tag{1}$$

I problemet ovan är det z, även kallad målfunktionen, som skall minimeras. Det som står nedanför målfunktionen är vilka villkor på x_1 och x_2 som funktionen ska optimeras under. För exemplet (1) är det möjligt att rita upp området som ska optimeras över i ett koordinatsystem, se Figur 2.



Figur 2: Området som beskrivs av bivillkoren för exempel (1), med den optimala punkten $(3, 2)$.

Problem som i exempel (1) kan generaliseras till en allmän problemformulering för ILP-problem,

$$
\begin{aligned}
&\text{minimera } z = \mathbf{c^T x}, \\
&\text{då} \qquad\quad A\mathbf{x} \leq \mathbf{b}, \\
&\qquad\quad C\mathbf{x} = \mathbf{d}, \\
&\qquad\quad x_i \in \mathbb{Z} \text{ för } i = 1, ..., n.
\end{aligned}
$$

På samma sätt kan också en allmän problemformulering för MILP-problem skrivas,

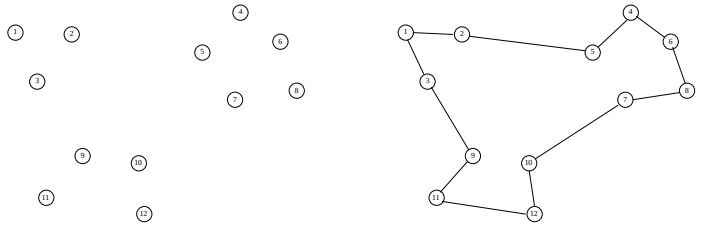
$$
\begin{aligned}
&\text{minimera } z = \mathbf{c^T x}, \\
&\text{då} \qquad\quad A\mathbf{x} \leq \mathbf{b}, \\
&\qquad\quad C\mathbf{x} = \mathbf{d}, \\
&\qquad\quad x_i \in \mathbb{Z} \text{ för } i = 1, ..., k, \\
&\qquad\quad x_j \in \mathbb{R} \text{ för } j = k+1, ..., n.
\end{aligned}
$$

För både ILP-problem och MILP-problem gäller att $1 \leq k \leq n-1,\ k \in \mathbb{N}$, $\mathbf{A} = \{a_{ij}\}_{i,j=1}^{n}$ respektive $\mathbf{C} = \{c_{ij}\}_{i,j=n}^{n}$ är villkorsmatriser, $\mathbf{c} = \{c_i\}_{i=1}^{n}$ en kostnadsvektor och $\mathbf{b} = \{b_i\}_{i=1}^{n}$ respektive $\mathbf{d} = \{d_i\}_{i=1}^{n}$ vektorer.

Vissa av dessa ILP-problem har en speciell struktur som går att utnyttja. I de två kommande avsnitten presenteras i tur och ordning specialfallen: handelsresandeproblem och kappsäcksproblem.

6.1 Handelsresandeproblemet

Enkelt uttryckt går handelsresandeproblemet ut på att besöka samtliga noder i en graf, se Figur 3, en gång och sedan ta sig tillbaka till startnoden med sammanlagt kortast möjliga väg.



Figur 3: I (a) syns noder som kan länkas samman till den optimala handelsresandetur som visas i (b).

För att modellera problemet matematiskt införs variabeln x_{ij} där

$$
x_{ij} = \begin{cases} 1, & \text{om det går en väg från nod } i \text{ till } j, \\ 0, & \text{annars.} \end{cases}
$$

För problemet i Figur 3 skulle de innebära att $x_{12} = x_{25} = x_{54} = x_{46} = x_{68} = x_{87} = x_{710} = x_{1012} = x_{1211} = x_{119} = x_{93} = x_{31} = 1$, medan övriga $x_{ij} = 0$.

Dessutom finns en kostnad för att ta sig från nod i till nod j. Denna beskrivs av c_{ij} och kan exempelvis representera tiden det tar att ta sig från i till j eller kostnaden för att flyga den delrutten om grafen representerar ett nät av flygturer. I exemplet motsvaras den kostnaden av avståndet mellan noderna. Oavsett vad för typ av kostnad c_{ij} står för blir målfunktionen, det vill säga den funktion som ska minimeras linjär, enligt:

$$
\text{minimera } z = \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij},
$$

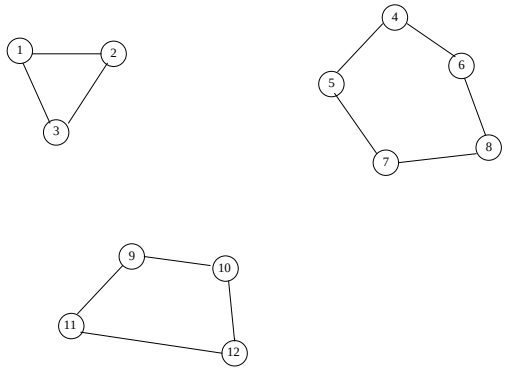
där vi har låtit mängden av alla noder betecknas med N. Dock behövs ett antal villkor på x_{ij} för att få en handelsresandetur. Först och främst måste x_{ij} anta heltalsvärdena 0 eller 1, i annat fall skulle möjligheten att åka en del av vägen från i till j finnas. Därtill måste det finnas villkor som ser till att varje nod besöks en gång och endast en gång. Det innebär att

$$\sum_{i \in N} x_{ij} = 1, \quad j \in N.$$

Dessutom måste varje nod lämnas en och endast en gång. Analogt med föregående bivillkor formuleras det som:

$$\sum_{j \in N} x_{ij} = 1, \quad i \in N.$$

Dessa krav räcker dock inte. Det som kan hända med enbart de tidigare nämnda villkoren är att så kallade subturer tillåts; se Figur 4.



Figur 4: Resultatet av att tillåta subturer.

Därför läggs också villkoren

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 \text{ där } S \subset N, \quad |S| \geq 2$$

till. Betydelsen är att det som mest kan finnas $|S| - 1$ bågar i varje strikt delmängd S av N med minst 2 och maximalt $|N| - 1$ noder.

Sammanfattningsvis beskrivs hela handelsresandeproblemet med samtliga bivillkor som:

$$\begin{aligned}
&\text{minimera } z = \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij}, \\
&\text{då} \qquad \sum_{i \in N} x_{ij} = 1, \quad j \in N, \\
&\qquad\quad \sum_{j \in N} x_{ij} = 1, \quad i \in N, \\
&\qquad\quad \sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1 \text{ där } S \subset N,\ |S| \geq 2 \\
&\qquad\quad x_{ij} \in \{0, 1\},\ i, j \in N.
\end{aligned}$$

6.2 Kappsäcksproblem

Många optimeringsproblem handlar om hur objekt ska väljas för att tillgängliga resurser ska användas på bästa sätt då resurserna i princip alltid är begränsade. Samtidigt är objekten som väljs ofta av sådan natur att variablerna i optimallösningarna måste vara på heltalsform. Dessa optimeringsproblem kan med fördel beskrivas som kappsäcksproblem.

Den matematiska formuleringen utgår ifrån boken Optimeringslära[7] och innehåller en målfunktion som optimeras för att till exempel minimera inköpskostnader samt bivillkor som

beskriver på vilket sätt resurserna, i detta fall pengar, är begränsade. Ett vanligt sätt att matematiskt formulera kappsäcksproblem är att;

$$\begin{aligned} \text{maximera } z = & \sum_{j=1}^{n} c_j x_j, \\ \text{då} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, \ldots, m, \\ & x_j \in \{0, 1\}, \quad j = 1, \ldots, n, \end{aligned}$$

där objektet x_j får värdet ett om det väljs och noll om det inte väljs. Enligt formuleringen ovan finns m olika resurstyper med begränsning b_i och n olika objekt. Variabeln z kan här till exempel motsvara vinst från ett projekt där varor produceras som bidrar till eller påverkar vinsten negativt genom koefficienten c_j. Samtidigt kan objekt bidra olika mycket till resursåtgången vilket beskrivs med hjälp av koefficienterna a_{ij}.

För att beskriva hur ett kappsäcksproblem kan se ut och hur problemet kan formuleras matematiskt följer ett exempel nedan.

Kappsäcksexempel - aktieinvestering

En aktieägare har bestämt sig för att investera maximalt 700 kr i ett eller flera olika företag. De fyra företag som aktieägaren ska välja mellan betecknas med x_1, x_2, x_3 och x_4. Aktierna i de olika företagen kostar olika mycket och ger olika stor utdelning enligt Tabell 1. Nu vill aktieägaren beräkna i vilket eller vilka företag som han ska investera sina pengar för att maximera vinsten.

Företag	Aktiekostnad [kr]	Vinst [kr]
1	400	700
2	200	300
3	200	200
4	300	200

Tabell 1: Tabellen visar kostnader och vinster för aktier i olika företag.

Problemet är ett heltalsproblem där variablerna endast kan anta vissa diskreta värden, det vill säga variablerna antar värdet ett om aktieägaren borde investera i företaget för att maximera sin vinst och värdet noll om han inte borde investera i ett visst företag. Begränsningen i detta fall är att aktieägaren inte har mer än 700 kr att investera och att en aktie endast kan köpas i ett exemplar. Omskrivet i matematiska termer blir problemet då att;

$$\begin{aligned} \text{maximera } z = & 700x_1 + 300x_2 + 200x_3 + 200x_4, \\ \text{då} \quad & 400x_1 + 200x_2 + 200x_3 + 300x_4 \leq 700, \\ & x_j \in \{0, 1\} \text{ för } j = 1, \ldots, 4. \end{aligned}$$

Kappsäcksproblemet kan lösas med till exempel Branch-and-Bound-metoden, som beskrivs i avsnitt 7.1, och då ges optimallösningen av $x_1 = x_2 = 1$ och $x_3 = x_4 = 0$ med den optimala vinsten $z^* = 1000$ kr.

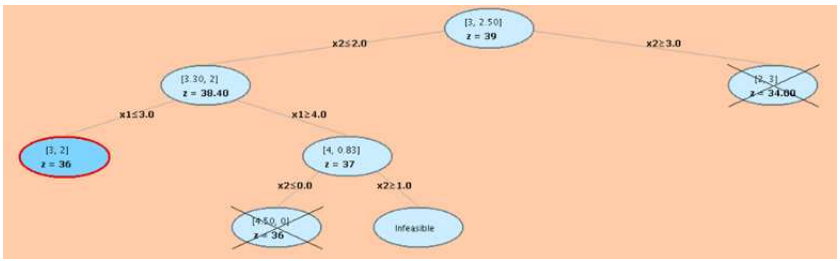
7 Branch-and-Bound

Branch-and-Bound-algoritmen är ett övergripande namn för en optimeringsalgoritm som i korta drag går ut på att dela upp ett optimeringsproblem, som är komplicerat att hitta ett optimum till direkt utifrån problemformuleringen, i delproblem som då är lättare att lösa än ursprungsproblemet. Algoritmen är en söndra-och-härska-teknik, mer känd under det engelska namnet *Divide and Conquer*, som kan användas för att lösa ett flertal olika typer av optimeringsproblem. I nästa avsnitt presenterar vi hur algoritmen ter sig för generella heltalsoptimeringsproblem (ILP). De olika stegen i algoritmen som beskrivs där uppkommer i princip alla varianter av Branch-and-Bound och kan därför också ses som en allmän beskrivning av vad som sker i varje iteration.

Därefter går vi vidare med ett optimeringsexempel som löses med Branch-and-Bound-metoden för generella heltalsoptimeringsproblem. Avslutningsvis beskrivs ett antal av de Branch-and-Bound-tekniker som finns för handelsresandeproblemet.

7.1 Branch-and-Bound-algoritmen för linjära heltalsproblem

Ett sätt att lösa linjära heltalsproblem är att använda sig av optimeringsmetoden Branch-and-Bound. I metoden förgrenas problemet i delproblem, ”Branching”, och en optimallösning beräknas för delproblemen. Dessa optimallösningar ger optimistiska uppskattningar av målfunktionsvärdet och begränsar det tillåtna lösningsområdet, ”Bounding” [10].



Figur 5: Illustration av Branch-and-Bound-metoden.

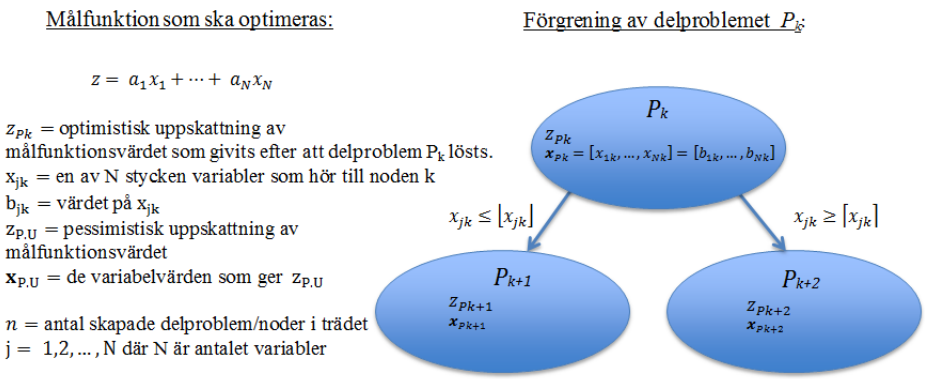
För att få en bra överblick över hur förgreningen sker beskrivs den ofta grafiskt som ett upp-och-nervänt träd, se Figur 5. Trädets rot symboliserar det första delproblemet och noderna, som fyller grenarna i trädet, står för de delproblem som genererats under lösningens gång. För att enklare kunna följa lösningsgången kan variablernas värden och målfunktionens värde för varje delproblem skrivas ut i noderna. I vilken ordning som delproblemen beräknas beror på valet av sökmetod. Tre vanliga sökmetoder och hur de skiljer sig åt beskrivs i avsnitt 7.1.2.

7.1.1 Matematisk beskrivning av algoritmen

Det finns många olika Branch-and-Bound-algoritmer som ofta är anpassade för en viss typ av problemstruktur. En sak som alla algoritmerna har gemensamt är att varje delproblem i trädet löses i en relaxerad form, det vill säga en förenklad form.

En vanlig förenkling är att ta bort heltalskravet på variablerna och endast lösa problemet i varje nod med hjälp av linjärprogrammering (LP). Denna typ av förenkling brukar kallas LP-relaxation. En Branch-and-Bound-algoritm som använder sig av LP-relaxation är Land-Doig-Dakins algoritm som publicerades av A. H. Land och A. G. Dakin år 1960 [2]. Algoritmens struktur och beteckningar utgår ifrån beskrivningen av Land-Doig-Dakins algoritm i boken Optimeringslära [7] och beteckningarna definieras i Figur 6.
Land-Doig-Dakins metod fungerar för alla linjära heltalsproblem och algoritmens olika steg beskrivs härefter.

Steg 1: Uppskattning av målfunktionsvärdet
För att kunna börja genomsökningen efter en optimallösning behövs en pessimistisk uppskattning, $z_{P,U}$, av det optimala målfunktionsvärdet. Denna uppskattning används för att



Figur 6: Definition av beteckningar som används i Land-Doig-Dakin-algoritmen.

kunna förkasta vissa delområden, det vill säga grenar i trädet. Den pessimistiska uppskattningen kan sedan uppdateras efter hand men till att börja med finns det två alternativ för att ta fram en första uppskattning:

- En känd tillåten lösning existerar och dess pessimistiska uppskattning av målfunktionsvärdet kan användas.
- Om ingen tillåten lösning är känd får man utgå ifrån uppskattningen av $z_{P.U}$ till $+\infty$ vid ett minimeringsproblem och till $-\infty$ vid ett maximeringsproblem.

Steg 2: Lösning av ett LP-relaxerat delproblem P_k
Det LP-relaxerade delproblemet löses med hjälp av till exempel simplexmetoden och lösningen ger värden på variablerna och målfunktionsvärdet z_{Pk}. Målfunktionsvärdet ger en optimistisk uppskattning av det optimala målfunktionsvärdet i delträdet.

Det finns några specialfall som kan uppkomma vid lösning av ett delproblem och om något av dessa inträffar sker ingen fortsatt sökning i den aktuella grenen utan algoritmen fortsätter med steg fyra nedan. De specialfall som kan uppkomma är:

- En lösning på det aktuella delproblemet saknas.
- En bättre lösning på målfunktionsvärdet går inte att hitta i det aktuella delområdet om:
  - Målfunktionsvärdet z_{Pk} är större än $z_{P.U}$ vid ett minimeringsproblem eller mindre än $z_{P.U}$ vid ett maximeringsproblem.
  - Delproblemet ger en tillåten lösning det vill säga en lösning där variablerna antar heltalsvärden. Skulle den tillåtna lösningens målfunktionsvärde z_{Pk} vara mindre än $z_{P.U}$ vid ett minimeringsproblem eller z_{Pk} vara större än $z_{P.U}$ vid ett maximeringsproblem uppdateras $z_{P.U}$ med z_{Pk}:s värde och $x_{P.U}$ med de till z_{Pk} tillhörande variabelvärdena x_{Pk}.

Steg 3: Förgrening till två nya delproblem
Vid förgreningen av ett delproblem P_k väljs något av problemets variabler x_j med värdet b_j, som inte är ett heltal, ut. Följande villkor sätts sedan på variabeln så att två nya delproblem P_{n+1} och P_{n+2} bildas:

- P_{n+1} ges av problemet P_k där villkoret att x_j ska vara mindre än eller lika med heltalsdelen av b_j:s värde har lagts till.
- P_{n+2} ges av problemet P_k där villkoret att x_j ska vara större än eller lika med heltalsdelen av b_j:s värde $+1$ har lagts till.

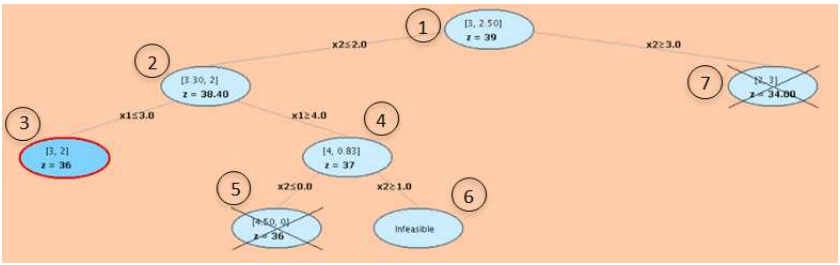
Antalet skapade delproblem efter detta steg är nu lika med $n + 2$ och noden P_k betecknas som genomsökt.

Steg 4: Fortsättning med en ny nod
Om det finns någon nod eller några noder som inte blivit genomsökta väljs en nod P_k av dessa ut och genomsökningen fortsätter från steg två. I vilken ordning som noderna ska genomsökas beror på vilken sökmetod som används, se *Sökmetoder* nedan.

Steg 5: Avslutning av genomsökningen
Om det inte finns några fler noder att söka igenom eller om något avbrottskriterium, se avsnitt 7.1.4, har uppfyllts avslutas sökningen. Optimallösningen är $z_{P.U}$ med tillhörande variabler $x_{P.U}$.

7.1.2 Sökmetoder

Valet av sökmetod besvarar frågan om i vilken ordning som delproblemen ska lösas. Vilken metod som används spelar roll för hur det slutgiltiga trädet kommer att se ut och kan beroende på problemets struktur göra så att lösningen av problemet tar olika lång tid att beräkna[7]. De tre kommande avsnitten beskriver kort de vanligaste sökmetoderna och utgår ifrån boken Optimeringslära [7].



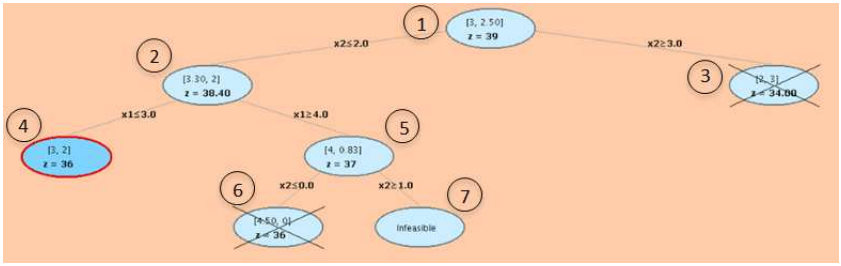
Figur 7: Träd som skapats vid djup-först-sökning.

Djup-först-sökning
När sökmetoden djup-först används evalueras den nod som för tillfället ligger djupast, eller med andra ord längst ner, i trädet [11]. Detta innebär att det är den nod vars område är mest begränsad som undersöks eftersom det vid varje förgrening till nya noder i trädet läggs till fler villkor. Djup-först-sökning kan användas som en första metod för att snabbt hitta en tillåten lösning till ett problem. På så sätt kan områden utan en bättre tillåten lösning förkastas fortare om man därefter löser det ursprungliga problemet med en annan sökmetod som till exempel bredd-först. Figur 7 visar ett träd som har skapats då sökmetoden djup-först använts[8].

Bredd-först-sökning
I en bredd-först-sökning beräknas delproblemen i en sådan ordning att trädet blir så brett som möjligt. Det betyder att de noder som ligger på samma nivå i trädet löses innan man börjar gå djupare i trädet. Bredd-först-sökning kan vara lämplig att använda om det finns anledning att tro att en bra lösning kan hittas utan att många begränsningar behöver läggas till på variablerna och samtidigt vill undvika att begränsa problemet för snabbt. Genom att inte gå djupt i trädet utan istället lösa noder i trädets bredd undviks att problemet begränsas för fort. Sökmetoden illustreras i Figur 8.

[7] Hur lång tid det tar att lösa ett problem beror främst på storleken hos problemet, det vill säga hur många noder som behöver evalueras innan en optimallösning till det ursprungliga problemet hittas. De problem som kan lösas i vårt program är inte tillräckligt stora för att lösningstiden ska variera på ett märkbart sätt vid användning av olika sökmetoder.

[8] I Figur 7 genomsökes alltid vänstergrenen först men i vilken ordning som grenar evalueras i ett träd kan variera. Vänstergrenen genomsökes även först i Figur 8.



Figur 8: Träd som skapats vid bredd-först-sökning och även i detta fall vid bäst-först-sökning.

Bäst-först-sökning
Under genomsökningen av trädet vid en bäst-först-sökning beräknas först den nod vars förälder har den bästa optimistiska uppskattningen av målfunktionsvärdet[9]. I Figur 8 visas ett träd som skapats då sökmetoden bäst-först använts. Bäst-först-sökning ger i detta fall samma träd som bredd-förstsökning men oftast brukar olika sökmetoder också generera olika träd.

7.1.3 Vikten av att ha en bra tillåten lösning till det ursprungliga problemet

Eftersom en effektiv begränsning av sökområdet gör att optimallösningen till det ursprungliga problemet kan hittas fortare är det viktigt att ha en beräknad tillåten lösning som är så bra som möjligt. En bättre tillåten lösning gör nämligen att större delar av sökområdet, fler grenar i trädet, kan kapas bort. Med en tillåten lösning menas att alla variablerna har ett heltalsvärde, eftersom det är linjära heltalsproblem som löses, och att problemets bivillkor är uppfyllda. Det är speciellt viktigt med en bra tillåten lösning för att förkorta beräkningstiden då problemet är stort medan lösningstiden för mindre problem inte påverkas så mycket av hur bra den ursprungliga tillåtna lösningen är.

Valet av sökmetoder och kombinationer av sådana kan också påskynda sökningen efter optimallösningen. Till exempel nämns kombinationen av sökmetoderna djup-först med bredd-först under avsnitt 7.1.2.

7.1.4 Optimistisk uppskattning

En optimistisk uppskattning av målfunktionsvärdet i en del av trädet ges, som tidigare nämnts, när en nod har evaluerats och givit en lösning där inte alla variabler är heltal. I avsnittet om den matematiska beskrivningen av Branch-and-Bound-algoritmen beskrivs ingen tillämpning av den bästa optimistiska uppskattningen för hela problemet. Anledningen till detta är att den främst brukar användas för att förkasta delområden när det inte är praktiskt möjligt att söka igenom alla noder som bildas i trädet. Problem med att alla noder i trädet inte kan evalueras, eftersom detta skulle ta alltför lång tid, uppkommer inte i vårt program på grund av att det inte är avsett för att lösa problem av den storleksordningen.

För att förkorta lösningstiden för stora problem kan den för tillfället bästa optimistiska uppskattningen användas tillsammans med den pessimistiska i ett så kallat avbrottskriterium [7] för att inte alla delområden ska behöva genomsökas. Avbrottskriteriet,

$$\frac{\text{pessimistisk uppskattning - optimistisk uppskattning}}{\text{optimistisk uppskattning}} < \epsilon,$$

ser till så att sökningen inte fortsätter efter att ett tillräckligt bra målfunktionsvärde hittats. Den pessimistiska uppskattningen, se avsnitt 7.1.1, i avbrottskriteriet ges av den tillåtna lösning som finns för det ursprungliga problemet medan den optimistiska uppskattningen är den uppskattning som för tillfället är den bästa bland de noder som kan förgrenas vidare i trädet.

[9] Alla noder har en förälder som de härstammar från vilket visas i trädet genom att de förbinds med en linje. För att få en bättre förståelse för kopplingen mellan (nod-)barnet och dess förälder i trädet kan man se hela trädet som ett släktträd där roten, det första delproblemet, är stamfadern.

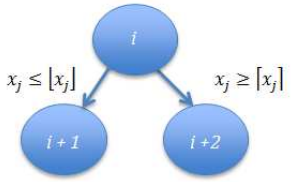
7.1.5 Bevis för att Branch-and-Bound-algoritmen konvergerar

Nedan följer ett bevis som visar att Branch-and-Bound-algoritmen konvergerar, vilket är ekvivalent med att trädet inte kan bli oändligt stort och noderna inte oändligt många. Beviset bygger på Proposition 2.1 i boken "Integer and Combinatorial Optimization"[12, sid. 357].

Sats. *Låt området för det relaxerade problemet av ett linjärt heltalsproblem beskrivas av:* $\mathbf{P} = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{x} \geq 0, \mathbf{Ax} \leq \mathbf{b}\}$

Om $\mathbf{P}$ *är begränsat så kommer det träd som utvecklas genom förgrening, med hjälp av variabeldelning vid lösning med Branch-and-Bound-algoritmen, att vara ändligt, förutsatt att det för varje nod i som behöver förgrenas över väljs en delning på formen* $x_j \leq \lfloor x_j^i \rfloor$, $x_j \geq \lceil x_j^i \rceil$ *där* x_j^i *inte är ett heltal, se Figur 9.*

Speciellt gäller att om w_j *är det största värdet som* x_j *antar på* $\mathbf{P}$*, avrundat uppåt till närmsta heltal, så kan inte höjden på trädet bli större än* $\sum_{j \in N} w_j$*.*



Figur 9: Nod i förgrenas över variablen x_{jk} som inte är ett heltal.

Bevis. När bivillkoret $x_j \leq d$ har lagts till för något $d \in \{0, \ldots, w_j\text{–}1\}$ är de enda ytterligare bivillkor som kan finnas på en väg från trädroten till ett löv; $x_j \leq d'$ där $d' \in \{0, \ldots, d\text{–}1\}$ och $x_j \geq d''$ där $d'' \in \{1, \ldots, d\}$.

Från detta följer att det största antalet bivillkor som involverar x_j kommer att uppkomma genom att addera villkoret:

- $x_j \leq d$ för alla $d \in \{0, \ldots, w_j\text{–}1\}$ eller
- $x_j \geq d$ för alla $d \in \{1, \ldots, w_j\}$ eller
- $x_j \geq d$ för alla $d \in \{1, \ldots, \alpha\}$ och $x_j \leq d$ för alla $d \in \{\alpha, \ldots, w_j\text{-}1\}$ där $\alpha \in \{\mathbb{Z} : 1 \leq \alpha \leq w_j - 1\}$

I var och ett av dessa fall krävs maximalt w_j antal villkor på x_j. Med andra ord är det största antalet villkor som kan läggas till problemet $\sum_{j \in N} w_j$, vilket i sin tur kan användas som övre gräns för höjden av trädet. Om trädet har en begränsad höjd så gäller även att trädet är ändligt och att algoritmen konvergerar. □

7.2 Branch-and-Bound-exempel för ett linjärt heltalsproblem

Nedan följer ett exempel på hur Branch-and-Bound-algoritmen kan användas för att lösa ett linjärt heltalsoptimeringsproblem i två dimensioner, det vill säga ett linjärt heltalsoptimeringsproblem med två variabler.

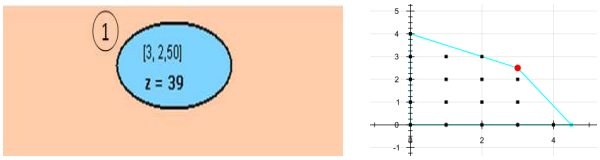
Ett industriföretag som tillverkar reaktorer har fått en beställning från ett forskningsföretag som studerar de reaktioner som sker i katalysatorer hos bilar. Företagen har skrivit ett avtal om tillverkning av reaktorer som forskningsföretaget ska kunna utföra ett antal tester på. Eftersom forskningsföretaget har begränsat med utrymme i sina lokaler behöver markutrymmet som reaktorerna tar upp vara mindre än 8 m^2. Två reaktorer, typ 1 och typ 2, som industriföretaget tillverkar är tillräckligt små för att kunna uppfylla det kravet. Reaktorn av typ 1 upptar 1 m^2 medan reaktorn av typ 2 tar upp 2 m^2.

Affärsavtalet som parterna har kommit överens om innebär att vinsten för industriföretaget kommer att vara 80 000 kr för reaktorn av typ 1 och 60 000 kr för reaktorn av typ

2. Industriföretaget har lång erfarenhet inom tillverkning av reaktorer och vet därför av erfarenhet att det tar tio timmar att tillverka reaktorn av typ 1 och sex timmar att tillverka reaktorn av typ 2. Forskningsföretaget behöver reaktorer att testa på omgående och därför har industriföretaget endast 45 timmar på sig att tillverka reaktorerna och de kan endast tillverka en reaktor i taget. För att veta vilka reaktorer som ska väljas att tillverka för att maximera vinsten bestämmer industriföretaget sig för att optimera problemet genom att;

$$\begin{aligned} \text{maximera } z = \ & 8x_1 + 6x_2, \\ \text{då} \qquad & x_1 + 2x_2 \leq 8, \\ & 10x_1 + 6x_2 \leq 45, \\ & x_1,\ x_2 \geq 0, \qquad \text{heltal.} \end{aligned}$$

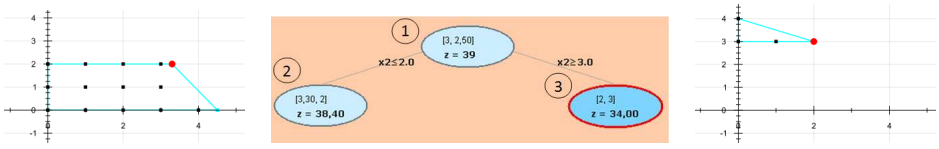
Lösning av problemet med Branch-and-Bound-metoden



(a) (b)

Figur 10: (a) visar noden som symboliserar trädets rot och (b) visar problemets lösningsområde.

Trädets rot: Den LP-relaxerade formen av problemet som beskrivs ovan löses med hjälp av simplexmetoden och ger den första noden i trädet, roten, se Figur 10.

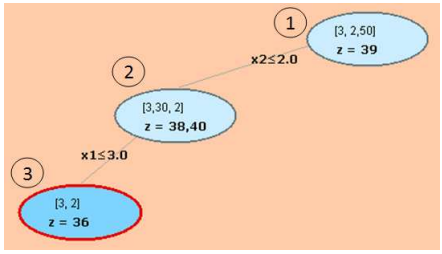


(a) (b) (c)

Figur 11: (a) visar området som definieras av nod 2 då trädrotsnoden 1 i (b) förgrenats över variabeln x_2 medan (c) visar området för nod 3.

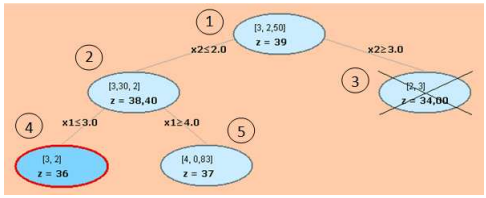
Förgrening av den första noden: I detta exempel används bredd-först-sökning och noden i Figur 10a kommer att förgrenas, över variabeln x_2 som inte är ett heltal, till två nya noder som betecknas med nummer två och tre enligt Figur 11b. Nod nummer tre är en tillåten lösning och därför uppdateras den pessimistiska uppskattningen, $z_{P.U}$, i detta steg till $z_{P.U}$ = 34. Om till exempel sökningsmetoden djup-först hade använts istället så skulle trädet efter det att tre delproblem lösts se ut som i Figur 12.

Det är viktigt att ingen lösning försvinner när en nod förgrenas. I Figur 11a och 11c visas de delområden som givits då området i Figur 10b förgrenats och från dessa figurer kan man se att ingen heltalslösning, markerade som svarta punkter i figurerna, försvinner när området för den första noden delas upp.
Fortsatt genomsökning: När sökningen nu ska fortsätta måste man välja om den ska genomföras först i den vänstra eller högra grenen i trädet. I programmet och i detta exempel söks den vänstra grenen igenom först och nod nummer två förgrenas därför till två nya noder med nummer fyra och nummer fem, se Figur 13. Nod nummer fyra är en tillåten lösning och ger en mindre pessimistisk uppskattning av målfunktionsvärdet än den som vi redan har vilket gör att $z_{P.U}$ uppdateras till $z_{P.U}$ = 36. Detta gör också att grenen med nod tre kan



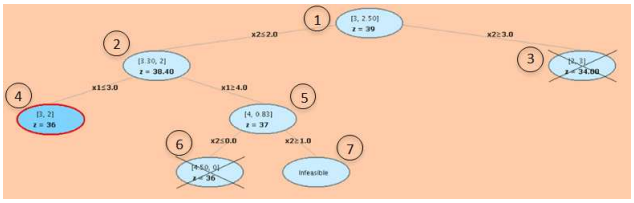
Figur 12: Trädets utseende vid djup-först-sökning istället för bredd-först-sökning.

bortses ifrån vilket i bilden symboliseras med ett kryss över den aktuella noden.



Figur 13: Fortsatt genomsökning i trädet.

Den enda gren som sökningen kan fortsätta i är grenen med nod nummer fem. Detta beror på att de andra två grenarna antingen har en nod med en tillåten lösning eller är en bortskuren gren. När nod fem förgrenas fås en nod, nummer sex se Figur 14, som kan skäras bort på grund av att dess optimistiska uppskattning av målfunktionsvärdet är lika med den pessimistiska uppskattningen och vi söker endast en optimallösning, inte flera. Förgreningen av nod nummer fem ger även en nod som betecknas med nummer sju, se Figur 14. Denna nod saknar lösning på det aktuella delproblemet och betecknas *Infeasible*.



Figur 14: Trädets utseende när alla noder är genomsökta.

Optimallösning: Eftersom det inte finns några fler noder att söka igenom har en optimallösning hittats med det optimala målfunktionsvärdet $z^* = 36$ och de tillhörande variabelvärdena $x_1 = 3$ och $x_2 = 2$, se nod fyra i Figur 14. Detta innebär att den maximala vinsten för industriföretaget ges om de tillverkar tre reaktorer av typ 1 och två reaktorer av typ 2. Vinsten blir då 360 000 kr.

7.3 Branch-and-Bound-tekniker för handelsresandeproblemet

Handelsresandeproblemet är, som kanske syns från den matematiska formuleringen i avsnitt 6.1, ett mycket komplext optimeringsproblem att lösa. Antalet möjliga turkombinationer av n stycken noder är $(n-1)!$ om det finns en väg mellan varje nodpar. Det gör att tiden det tar att lösa ett handelsresandeproblem genom att testa alla möjliga kombinationer blir omöjligt i praktiken även för den snabbaste dator.

Dock uppkommer ofta problem av handelsresandekaraktär inom industrin, vilket gör det viktigt att finna effektiva algoritmer. Idag finns en hel uppsjö av olika tekniker för att hitta bra, men inte alltid optimala, lösningar. Av dessa finns ett antal som bygger på Branch-and-Bound och några av dem kommer att tas upp nedan.

7.3.1 Heltalsrelaxering

En av dessa Branch-and-Bound-tekniker bygger på en relaxering av heltalskravet på variablerna x_{ij}. Algoritmen blir då densamma som för generella heltalsoptimeringsproblemen, det vill säga att varje x_{ij} tillåts ha ett värde mellan 0 och 1 för att sedan dela upp problemet så att en variabel i taget tvingas anta värdet 0 eller 1 beroende på gren i Branch-and-Bound-trädet.

Problemet med att välja den här Branch-and-Bound-tekniken för handelsresandeproblemet är att det i många fall kommer att ske alldeles för många förgreningar för att algoritmen ska vara effektiv i praktiken. Till stor del beror det på att antalet variabler ökar kvadratiskt med antalet städer.

7.3.2 Förgrening över möjliga vägar

Den här varianten av Branch-and-Bound för handelsresandeproblemet skiljer sig ganska markant från heltalsrelaxeringstekniken. Först och främst bygger den på att kostnaden för att ta sig från nod i till j beskrivs av en kostnadsmatris. I första steget i algoritmen väljs en nod ut som startnod. Teorin för algoritmen är hämtad från en powerpointpresentation av Busby med flera ([13]) och börjar, efter det att en startnod valts ut, med att en optimistisk nedre gräns för optimalvärdet beräknas genom att summera de minsta värdena i varje rad i kostnadsmatrisen. För att öka förståelsen för de kommande stegen i algoritmen tar vi ett exempel till hjälp, också det taget från Busbys presentation [13], med följande kostnadsmatris C:

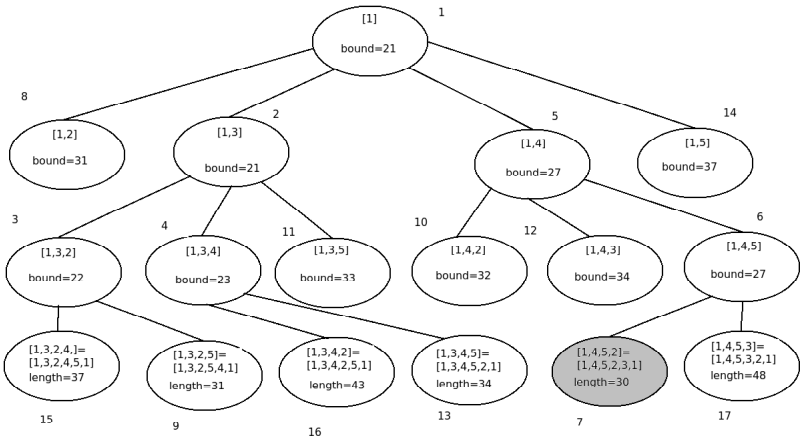
$$C = \begin{bmatrix} - & 14 & 4 & 10 & 20 \\ 14 & - & 7 & 8 & 7 \\ 4 & 5 & - & 7 & 16 \\ 11 & 7 & 8 & - & 2 \\ 18 & 7 & 17 & 4 & - \end{bmatrix}.$$

För problemet i fråga blir då den första optimistiska gränsen $4 + 7 + 4 + 2 + 4 = 21$ oavsett vilken nod som väljs som startnod, i det här fallet valdes dock den första. I nästa steg delas problemet upp i fyra grenar, en för varje väg ut från startnoden, se Figur 15. I vilken ordning dessa grenar sedan evalueras kan variera, men en variant är att alltid välja att beräkna den nod vars senast tillagda båge har lägst kostnad. Eftersom c_{13} är det minsta elementet i rad 1 räknas noden där väg $1 \rightarrow 3$ måste ingå ut först.

Därefter jämförs kostnaderna för att ta sig från nod 1 till någon av noderna $2, 4$ eller 5 med kostnaderna för att ta sig från nod 3 till nod $2, 4$ eller 5. Den nod som innehåller bågen med lägst kostnad blir sedan nästa nod att evaluera. I det här fallet visar det sig vara så att vi ska gå vidare med att undersöka gränsen för fallet då det är bestämt att handelsmannen ska gå från nod 1 till nod 3 och sedan vidare till nod 2. Den optimistiska gränsen för det här valet kan då beräknas till 22. Genom att fortsätta på samma sätt fås så småningom trädstrukturen i Figur 15.

Det här sättet att lösa handelsresandeproblemet är den av Branch-and-Bound-algoritmerna som mest påminner om uppräkning av samtliga möjliga turer. Även om beräkningen för varje nod går snabbt att utföra så kommer antalet noder som beräknas att öka i storleksordning $n!$ eftersom sannolikheten att noder kapas inte är så hög som kan önskas.

Figur 15: Trädstruktur efter förgrening över möjliga vägar. Figuren är hämtad från en powerpointpresentation av Busby med flera [13].



7.3.3 Relaxering genom att tillåta subturer

Ett vanligt sätt att lösa handelsresandeproblemet med Branch-and-Bound är att ta bort det tredje bivillkoret ur den matematiska formuleringen:

$$
\begin{aligned}
&\text{minimera } z = \sum_{i\in N}\sum_{j\in N} c_{ij}x_{ij}, \\
&\text{då} \qquad \sum_{i\in N} x_{ij} = 1, \quad j \in N, \\
&\qquad\quad \sum_{j\in N} x_{ij} = 1, \quad i \in N, \\
&\qquad\quad \sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1 \text{ där } S \subset N,\ |S| \geq 2 \\
&\qquad\quad x_{ij} \in \{0,1\},\ i,j \in N.
\end{aligned}
$$

Effekten blir då att subturer tillåts, se Figur 4 sida 12. Fördelen med att få problem på formen,

$$
\begin{aligned}
&\text{minimera } z = \sum_{i\in N}\sum_{j\in N} c_{ij}x_{ij}, \\
&\text{då} \qquad \sum_{i\in N} x_{ij} = 1, \quad j \in N, \\
&\qquad\quad \sum_{j\in N} x_{ij} = 1, \quad i \in N, \\
&\qquad\quad x_{ij} \in \{0,1\},\ i,j \in N.
\end{aligned}
$$

är att optimallösningen till dessa är av heltalstyp även om variablerna x_{ij} tillåts variera mellan 0 och 1, se sats 8.1 i Optimeringslära [7, sid. 216]. Således kan simplexalgoritmen användas för att lösa de relaxerade problemen på ett effektivt sätt.

Tanken är att, om inte lösningen är en tillåten handelsresandetur, i förgreningen förbjuda de subturer som uppstått. Det kan göras genom att hitta den deltur med lägst kardinalitet, det vill säga där minst antal noder ingår, och att sedan förgrena över denna. Om till exempel noderna x_{13}, x_{32} och x_{21} bildar en subtur så måste minst en av dessa anta värdet 0 i optimallösningen [14]. Om det är ett euklidiskt handelsresandeproblem, vilket betyder att kostnaden för att ta sig från nod a till nod b är lika stor som kostnaden från b till a, så måste även den motriktade vägen tas bort. Således fås en gren för varje väg, eller vägpar i det euklidiska fallet, som ska förbjudas.

Hur noder kapas och hur övre och undre gränser beräknas följer precis samma principer som för heltalsoptimeringsproblem och beskrevs i avsnitt 7.1.

7.3.4 Reduktion av kostnadsmatrisen för handelsresandeproblemet

Den fjärde Branch-and-Bound-tekniken vi ska gå igenom kan ses lite som en kombination av *Förgrening över möjliga vägar* och *Relaxering genom att tillåta subturer*. Även för den här metoden bygger algoritmen på att bågkostnaderna representeras med en kostnadsmatris. I det här fallet kommer matrisen dock att modifieras, eller reduceras som det kallas här, under lösningsgången. Algoritmen bygger dessutom på att alla bågkostnader är positiva.

Vi börjar först med att notera att om vi för varje rad i kostnadsmatrisen minskar respektive element med värdet på radens minsta element och sedan lägger till det här värdet till den totala kostnaden för handelsresandeturen så blir den slutliga kostnaden oförändrad jämfört med om vi inte gjort denna reduktion. Samma sak kan också göras för varje kolumn utan att totalkostnaden förändras. Det beror på att vi i den matematiska formuleringen i bivillkor ett och två kräver att ett element per rad respektive kolumn måste väljas ut för att få en handelsresandetur. Det är med denna reduktion som algoritmen, presenterad av John D. C. Little med flera i artikeln *An Algorithm for the Traveling Salesman Problem* år 1963 ([15]), börjar. Summan av dessa reducerade kostnader blir sedan en första undre gräns, eller optimistisk gräns om man så vill, för totalkostnaden för handelsresandeturen. Om vi inför beteckningarna, också de hämtade från Littles artikel,

- $z(t)$, kostnaden för en tur t innan reduktion av kostnadsmatrisen,
- h, summan av alla de reducerade kostnaderna,
- $z_1(t)$, kostnaden för turen t efter reduktion av kostnadsmatrisen,

så kan den optimistiska gränsen skrivas:

$$z(t) = h + z_1(t).$$

Om inte de bågar som väljs ut genom att för varje rad plocka ut bågen med lägst kostnad i den reducerade matrisen bildar en tillåten handelsresandetur görs förgreningen i nästa steg. För varje element c_{ij} i kostnadsmatrisen vars värde är noll beräknas därför $\Theta(i,j)$, där $\Theta(i,j)$ är summan av de minsta elementen i rad i i kostnadsmatrisen, borträknat den med index (i,j), plus det minsta elementet i kolumn j, också där borträknat element c_{ij}. Två grenar skapas sedan genom att välja ut den båge, b_{ij}, som har högst $\Theta(i,j)$. I den ena grenen, den vänstra, tillåts alla bågar förutom b_{ij} medan den högra grenen tvingas ha med b_{ij}. För att förklara stegen lite tydligare tar vi Littles exempel till hjälp. I exemplet har vi följande kostnadsmatris till att börja med:

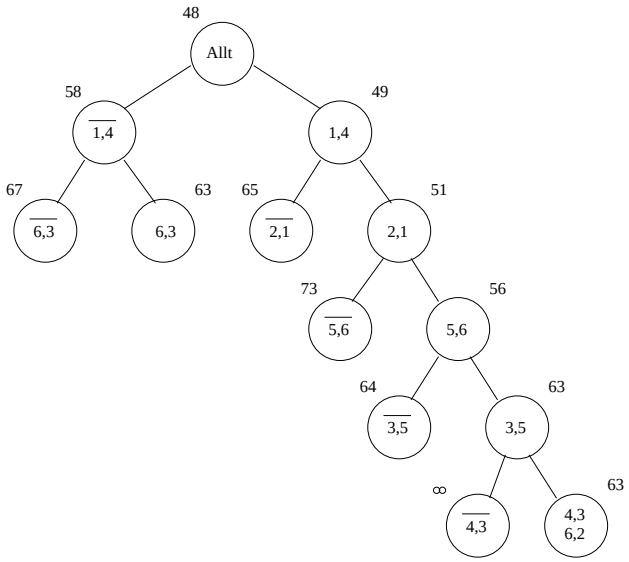
$$C = \begin{bmatrix} \infty & 27 & 43 & 16 & 30 & 26 \\ 7 & \infty & 16 & 1 & 30 & 25 \\ 20 & 13 & \infty & 35 & 5 & 0 \\ 21 & 16 & 25 & \infty & 18 & 18 \\ 12 & 46 & 27 & 48 & \infty & 5 \\ 23 & 5 & 5 & 9 & 5 & \infty \end{bmatrix}$$

som efter första stegets reduktion får följande form, med $\Theta(i,j)$ inom parentes:

$$C = \begin{bmatrix} \infty & 11 & 27 & 0\ (10) & 14 & 10 \\ 1 & \infty & 15 & 0\ (1) & 29 & 24 \\ 15 & 13 & \infty & 35 & 5 & 0\ (5) \\ 0\ (1) & 0\ (0) & 9 & \infty & 2 & 2 \\ 2 & 41 & 22 & 43 & \infty & 0\ (2) \\ 13 & 0\ (0) & 0\ (9) & 4 & 0\ (2) & \infty \end{bmatrix}.$$

Summan h av de reducerade kostnaderna blir då $16+1+0+16+5+5+5+0+0+0+0+0 = 48$. Från de beräknade $\Theta(i,j)$ får vi att vi ska förgrena över $(1,4)$-bågen.

I fallet där $(1,4)$-bågen inte får vara med sätts $c_{14} = \infty$ och reduktionen kan göras om för den nya kostnadsmatrisen och en ny optimistisk gräns kan beräknas för den grenen. I det andra fallet, då $(1,4)$-bågen måste vara med, kan alla element i rad ett respektive kolumn fyra sättas till oändligheten eftersom dessa element då inte längre ska kunna bli utvalda till att ingå i den slutgiltiga handelsresandeturen. Dessutom måste kostnaden för bågar som skulle kunna innebära att subturer uppstår om de väljs ut sättas till oändligheten i varje iteration. Det betyder för den här iterationen att $c_{41} = \infty$.



Figur 16: Slutgiltigt träd för metoden, där värdet strax ovanför respektive nod är värdet på den optimistiska gränsen [15].

Vi kan se det som ganska sannolikt att $(1,4)$-bågen kommer att vara med i den slutgiltiga turen eftersom den har en låg kostnad och därför väljer man alltid att fortsätta med att förgrena över noden längst till höger till dess att en tillåten tur hittas. För den högra barnnoden till roten får vi då följande kostnadsmatris efter reduktion:

$$C = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ 0\ (16) & \infty & 14 & \infty & 28 & 23 \\ 15 & 13 & \infty & \infty & 5 & 0\ (5) \\ \infty & 0\ (2) & 9 & \infty & 2 & 2 \\ 2 & 41 & 22 & \infty & \infty & 0\ (2) \\ 13 & 0\ (0) & 0\ (9) & \infty & 0\ (2) & \infty \end{bmatrix},$$

vilket ger en undre gräns, $48+1 = 49$, för kostnaden för den här grenen. I den här iterationen sattes bara ett element till 0, nämligen c_{21}. Det hade också högst $\Theta(i,j)$ vilket innebär att den nya förgreningen nu ska göras över $2,1$-bågen. Kravet för att inga subturer ska uppstå inom den här grenen är nu, förutom det tidigare ställda kravet att $(4,1)$-bågen inte ska ingå, att varken $(2,1)$-bågen eller $(1,2)$-bågen ska inkluderas. Dessutom kan inte heller $(4,2)$-bågen räknas in eftersom subturen $1 \to 4 \to 2 \to 1$ då skulle vara tillåten. Därför sätts kostnaden

på dessa till oändligheten. Genom att fortsätta på samma sätt fås så småningom ett träd enligt Figur 16.

Som synes i Figur 16 så har en del noder kapats eftersom en tillåten tur med lägre totalkostnad hittats. Vad som också kan nämnas är hur den optimistiska gränsen för nod nummer 11, det vill säga vänsterbarnet till rotnoden, beräknats. För noden i fråga fick vi gränsen $48 + 10 = 58$ ur följande kostnadsmatris:

$$C = \begin{bmatrix} \infty & 1 & 7 & \infty & 4 & 0\ (1) \\ 1 & \infty & 15 & 0\ (1) & 29 & 24 \\ 15 & 13 & \infty & 35 & 5 & 0\ (5) \\ 0\ (1) & 0\ (1) & 9 & \infty & 2 & 2 \\ 2 & 41 & 22 & 43 & \infty & 0\ (2) \\ 13 & 0\ (0) & 0\ (9) & 4 & 0\ (2) & \infty \end{bmatrix}.$$

Implementering, programupplägg och testning

Som tidigare nämnts så handlar det här kandidatprojeketet om att bygga en programvara för att illustrera olika Branch-and-Bound-algoritmer. Den matematiska teorin bakom algoritmen beskrevs i föregående kapitel men hur kan teorin översättas till javakod? I de två kommande avsnitten beskrivs hur vi implementerade de olika Branch-and-Bound-algoritmerna.

Därefter följer ett avsnitt om hur vi tänkt kring programfönstrets design ur ett pedagogiskt perspektiv. Där kommer varje del av fönstret gås igenom och utseende och innehåll motiveras. De tekniska detaljerna för hur exempelvis uppritningen av Branch-and-Bound-trädet fungerar eller hur området som ska optimeras över ritas upp tas inte upp här utan finns att läsa om i appendix B. Kapitlet avslutas sedan med ett stycke om hur programvaran testats både ur användarens synvinkel och kring optimeringslösarens korrekthet rent matematiskt sett.

8 Vår implementering av Branch-and-Bound för heltalsoptimeringsproblem

Vår implementering av Branch-and-Bound-algoritmen för generella heltalsoptimeringsproblem i Java bygger naturligtvis på den matematiska teorin bakom algoritmen, se avsnitt 7.1, men vi har också försökt utnyttja att Java är ett objektorienterat programmeringsspråk[10].

8.1 Initiering av data

Tanken är att när ett optimeringsproblem av heltalstyp ska lösas så skapas ett `BranchAndBoundILP`-objekt som innehåller information om den relaxerade LP-formuleringen, om det är ett maximerings- eller minimeringsproblem samt undre och övre gräns för målfunktionsvärdet. Därefter skapas tre olika datastrukturer för att hålla reda på vilken nod som ska evalueras i nästa steg i Branch-and-Bound-algoritmen, beroende på vilken sökmetod som används. Den första är en länkad lista som används vid bredd-förstsökning, den andra är likaså en länkad lista men i det här fallet för djup-förstsökning medan den tredje datastrukturen som används för bäst-förstsökning är av typen `TreeSet`, en trädstruktur som hela tiden hålls sorterad, i det här fallet efter optimum hos föräldranoden.

8.2 Lösningsgång

För att lösa optimeringsproblemet anropas därefter metoden `oneStepSolve`, vilken innehåller själva algoritmen.

Val av sökmetod
Det första som sker i metoden `oneStepSolve` är att undersöka vilken sökmetod användaren har valt:

```
1  //Breadth first
2  if(methodChoice==0) {
3   currentEvaluatedNode = nodeQueueBreadthFirst.removeFirst();
4   nodeQueueBestFirst.remove(currentEvaluatedNode);
5   nodeQueueDepthFirst.remove(currentEvaluatedNode);
6  }
7  //Depth first
8  else if (methodChoice==1) {
9   currentEvaluatedNode = nodeQueueDepthFirst.removeFirst();
```

[10] Se Diskussionskapitlet.

```
10    nodeQueueBestFirst.remove(currentEvaluatedNode);
11    nodeQueueBreadthFirst.remove(currentEvaluatedNode);
12   }
13   //Best first
14   else if(methodChoice==2) {
15    currentEvaluatedNode = nodeQueueBestFirst.last();
16    nodeQueueBestFirst.remove(currentEvaluatedNode);
17    nodeQueueBreadthFirst.remove(currentEvaluatedNode);
18    nodeQueueDepthFirst.remove(currentEvaluatedNode);
19   }
```

För att användaren ska kunna växla mellan sökningsmetoderna, bredd-först, djup-först och bäst-först under lösningens gång, plockas den valda noden också bort i de övriga datastrukturerna (rad 4-5, 10-11 samt 17-18 i koden ovan). Hur nya noder läggs till i de olika listorna återkoms till senare i kapitlet.

Lösning av det relaxerade problemet
När så en nod är vald löses LP-problemet för den här noden med hjälp av den externa LP-lösaren *QSopt*[16]. Den kommer att returnera en variabel av typen `int` som talar om lösningens status, det vill säga om problemet gick att lösa, om lösning saknas eller om området som skulle optimeras över var obegränsat så att optimallösningen går mot oändligheten. Lösaren har dock vissa begränsningar men mer om det i *Diskussionen* sida 42. Om lösning saknas eller om problemet är obegränsat händer inte så mycket mer än att ett par kommandon skickas till den klass som ritar upp trädet samt att `BranchAndBoundNodeILP`-klassen[11] får reda på att den innehåller ett problem utan tillåten lösning. Naturligtvis skärs den noden också av. Om det däremot finns en lösning händer desto mer, men mer om det i nästa avsnitt.

Test av nodspecifika avbrottskriterier
Först och främst ser vi till att få tag på alla variablers värde i optimum samt givetvis också optimalvärdet. I enlighet med den matematiska teorin undersöks sedan ett par av avbrottskriterierna:

```
1   if(minProblem){
2    if(lpOptimum+TOLERANCE>=upperBound || (Math.ceil(lpOptimum)>=
         upperBound && currentEvaluatedNode.
         isIntegerObjectiveFunctionCoefficients())) {
3     currentEvaluatedNode.setCut(true);
4     updateOptimisticBounds();
5     checkIfUpperBoundIsEqualToLowerBound();
6     return;
7    }
8   }
9   else {
10   if(lpOptimum-TOLERANCE<=lowerBound || (Math.floor(lpOptimum)<=
         lowerBound && currentEvaluatedNode.
         isIntegerObjectiveFunctionCoefficients())) {
11    currentEvaluatedNode.setCut(true);
12    updateOptimisticBounds();
13    checkIfUpperBoundIsEqualToLowerBound();
14    return;
15   }
16  }
```

För ett minimeringsproblem betyder det att inga nya noder ska läggas till om optimalvärdet är större än den övre gräns som finns för tillfället eller, vid heltalsvärden på koefficienterna i målfunktionen, att optimalvärdet avrundat uppåt till närmsta heltal är större än den övre gränsen. Dessutom behöver inga nya noder läggas till om nuvarande övre gräns är lika med den undre gränsen. Dock måste vi ta hänsyn till att lösaren ger oss numeriska värden så att det variabelvärde som exempelvis egentligen skulle vara 2 blev 1,9999 istället. Därför finns

[11] Klass som samlar nodspecifika data, se appendix A.

variabeln `TOLERANCE` som anger hur stort numeriskt fel vi tolererar utan att det ska påverka lösningsgången allt för mycket. På samma sätt fungerar det också för ett maximeringsproblem.

Det finns dock ytterligare ett kriterium för att inte lägga till några nya noder i kön till att bli evaluerade; om alla variabelvärden är heltal. Detta testas genom:

```
1  boolean moreVariables=true;
2  int i=1;
3  /*
4  *Checks for fractional values of the variables and adds
5  * a new constraint for the first one found
6  */
7  for(Double d: currentSolution) {
8   if(Math.abs(d-Math.round(d)) > TOLERANCE && moreVariables) {
9    lowerConstraintFormulation=currentEvaluatedNode.
         copyBranchAndBoundNode(true);
10   upperConstraintFormulation=currentEvaluatedNode.
         copyBranchAndBoundNode(false);
11   ceil=Math.ceil(d);
12   floor=Math.floor(d);
13   lowerConstraintFormulation.setNewConstraintString(i, floor, true);
14   upperConstraintFormulation.setNewConstraintString(i, ceil,false);
15   newLessThanConstraint.addLast(1.0);
16   newGreaterThanConstraint.addLast(1.0);
17   moreVariables=false;
18  }
19  else {
20   newLessThanConstraint.addLast(0.0);
21   newGreaterThanConstraint.addLast(0.0);
22  }
23  i++;
24 }
```

För varje variabelvärde i optimum kontrolleras alltså om det är ett heltal genom att titta på om beloppet på skillnaden mellan talet och talet avrundat till närmsta heltal är tillräckligt litet, se rad 8 i koden ovan. Återigen tillåter vi en viss felmarginal här på grund av numeriska fel. Dock bryr vi oss bara om första gången som någon variabel har en fraktionell del, därav den booleska variabeln `moreVariables`. Den sätts till att vara falsk om någon variabel inte är ett heltal eftersom vi inte vill förgrena över mer än en variabel i taget. Därtill använder vi oss av två listor, `newLessThanConstraint` och `newGreaterThanConstraint`, för att lägga till de nya bivillkor som uppstår vid förgrening. Koefficienten framför den variabel som ska förgrenas över sätts till 1 medan övriga koefficienter sätts till 0. I ett problem med 4 variabler betyder det att ett villkor som skulle skrivas

$$x_3 \leq 4$$

matematiskt, i datorn kommer att skrivas som:

```
[0.0 0.0 1.0 0.0]  floor,
```

där `floor` beskriver värdet på den övre gränsen för variabel 3. Förutom stegen som är direkt kopplade till Branch-and-Bound-algoritmen finns ett par rader kod som hör ihop med hur Branch-and-Bound-trädet kommer att ritas upp grafiskt, anropen av metoden `setNewConstraintString`. `setNewConstraintString` har till uppgift att se till att det nya bivillkoret som läggs till bevaras i tillhörande `BranchAndBoundNodeILP` som en sträng.

Det som sker härnäst är det sista steget i algoritmen, nämligen att lägga till de nya noderna om lösningen inte redan visade sig vara en heltalslösning:

```
1  if(moreVariables) {
2   if(minProblem) {
3    upperBound=lpOptimum;
4    /*
```

```
5       *Checks if any of the nodes in the queues waiting for being
6       * evaluated has a parent optimum that is greater than upper bound.
7       * If so those nodes will be removed.
8       */
9      ArrayList<BranchAndBoundNode> temp=new ArrayList<BranchAndBoundNode
           >(nodeQueueDepthFirst);
10     for(BranchAndBoundNodeInterface bab: temp) {
11      if(bab.getParentOptimum()>upperBound)  {
12       bab.getParentNode().setCut(true);
13       nodeQueueBreadthFirst.remove(bab);
14       nodeQueueBestFirst.remove(bab);
15       nodeQueueDepthFirst.remove(bab);
16      }
17     }
18    }
19    else {
20     lowerBound=lpOptimum;
21     /*
22      * Checks if any of the nodes in the queues waiting for being
23      * evaluated has a parent optimum that is lesser than upper bound.
24      * If so those nodes will be removed.
25      */
26     ArrayList<BranchAndBoundNode> temp=new ArrayList<BranchAndBoundNode
           >(nodeQueueDepthFirst);
27     for(BranchAndBoundNodeInterface bab:temp) {
28      if(bab.getParentOptimum()<lowerBound || ((Math.floor(bab.
           getParentOptimum()))<lowerBound && currentEvaluatedNode.
           isIntegerObjectiveFunctionCoefficients())))  {
29       bab.getParentNode().setCut(true);
30       nodeQueueBreadthFirst.remove(bab);
31       nodeQueueBestFirst.remove(bab);
32       nodeQueueDepthFirst.remove(bab);
33      }
34     }
35    }
36    bestSolution = currentSolution;
37    if(currentOptimumNode!=null) {
38     currentOptimumNode.setCurrentOptimum(false);
39     currentOptimumNode.setCut(true);
40     listOfCutNodes.add(currentOptimumNode);
41    }
42    currentOptimumNode=currentEvaluatedNode;
43    currentOptimumNode.setCurrentOptimum(true);
44    }
45   else {
46     newLessThanConstraint.addLast(floor);
47     newGreaterThanConstraint.addLast(ceil);
48     lowerConstraintFormulation.addLessThanInequalityConstraint(
           newLessThanConstraint);
49     nodeQueueBreadthFirst.addLast(lowerConstraintFormulation);
50     nodeQueueDepthFirst.addFirst(upperConstraintFormulation);
51     nodeQueueDepthFirst.addFirst(lowerConstraintFormulation);
52     nodeQueueBestFirst.add(lowerConstraintFormulation);
53     upperConstraintFormulation.addGreaterThanInequalityConstraint(
           newGreaterThanConstraint);
54     nodeQueueBreadthFirst.addLast(upperConstraintFormulation);
55     nodeQueueBestFirst.add(upperConstraintFormulation);
56   }
57   updateOptimisticBounds();
58   checkIfUpperBoundIsEqualToLowerBound();
```

Som tidigare antytts kommer boolesken `moreVariables` anta värdet `true` om alla variabel-

värden är heltal. I sådana fall kommer övre respektive undre gräns samt den nuvarande bästa lösningen att uppdateras beroende på om det är ett maximerings- eller minimeringsproblem som ska lösas. Dessutom kommer, i så fall, alla noder i köerna gås igenom för att se till att noder där nodens förälder har ett optimum som är sämre än nuvarande övre gräns för minimeringsproblem och undre gräns för maximeringsproblem tas bort enligt ett av bortskärningskriterierna. I annat fall, det vill säga om `moreVariables` är falsk, läggs de nya bivillkoren till som fås via förgreningen till i barnnoderna, `lowerConstraintFormulation` och `upperConstraintFormulation`, som innan enbart var kopior av föräldranoden. Dessa `BranchAndBoundNode`:s läggs sedan till i datastrukturerna för respektive sökningsmetod. Avslutningsvis kontrolleras om de övre och undre gränserna kan uppdateras och i så fall om den övre gränsen är lika med den undre.

Tillägg av nya noder i respektive sökmetods datastruktur
I fallet bredd-förstsökning har vi valt att alltid gå från vänster till höger i trädet. Det betyder att vänsterbarnet till en nod alltid ska ligga före högerbarnet i den länkade listan där noderna sparas. Det har vi löst genom att alltid först lägga till vänsterbarnet, det vill säga mindre-än-eller-lika-med-villkoret, sist i listan och sedan högerbarnet efter det, också sist i listan. För det konkreta trädet i Figur 17 skulle noderna alltså ligga i följande ordning för bredd-förstsökning:

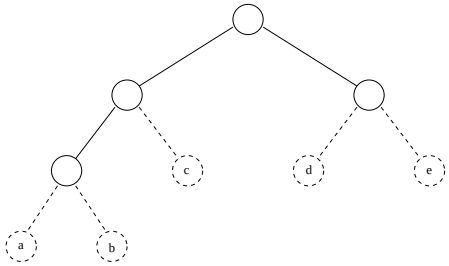
c	d	e	a	b

,

och så här för djupförstsökning:

a	b	c	d	e

.



Figur 17: Ett exempelträd för att illustrera köordning vid bredd-först- respektive djupförstsökning.

Vid djupförstsökning betyder det att de nya noderna alltid läggs till först i listan, därav kommandot `addFirst`.

Sist men inte minst har vi datastrukturen `TreeSet` där vi inte behöver göra något mer än att bara lägga till nodobjekten eftersom `TreeSet`-klassen [17] sköter sorteringen åt oss.

8.3 Uppdatering av optimistiska gränser

Eftersom ett relaxerat problem alltid har ett målfunktionsvärde som är bättre eller lika bra som för det ursprungliga problemet kan vi allt eftersom noder i Branch-and-Bound-trädet evalueras göra bättre och bättre uppskattningar på hur bra optimallösning som kan fås. Algoritmen för att göra detta bygger på att för varje nod som evalueras lägga den i en lista över möjliga noder för den optimistiska gränsen. Dessa noder tas sedan bort när båda dess barn är evaluerade och på så sätt hålls listan uppdaterad med enbart de noder som kan innehålla den optimistiska gränsen. För minimeringsproblem plockas den nod med minst optimalvärde ut ur listan medan det för maximeringsproblem är den med störst optimalvärde som gäller.

8.4 Lösning av hela optimeringsproblemet

I sin helhet löses sedan optimeringsproblemet genom att anropa `oneStepSolve()` till dess att alla nodköerna är tomma (vilket sker samtidigt för alla tre). Optimalvärdet ligger sedan sparat i variabeln `lpOptimum` och kan hämtas med metoden `getOptimum()`.

9 Vår implementering av Branch-and-Bound för handelsresandeproblemet

Implementeringen av den fjärde Branch-and-Bound-tekniken för handelsresandeproblemet, *Reduktion av kostnadsmatrisen för handelsresandeproblemet*, se också avsnitt 7.3.4, liknar i mångt och mycket Branch-and-Bound-algoritmen för heltalsoptimeringsproblem. Precis som att algoritmen för heltalsoptimeringsproblemen börjar med initiering av data och val av sökmetod börjar även den här implementeringen på samma sätt och med liknande kodinnehåll.

9.1 Reduktion av kostnadsmatrisen

När en nod är vald reduceras dess kostnadsmatris med hjälp av metoden `reduceMatrix`. Metoden tar en `BranchAndBoundNodeTSP`[12] som parameter och ändrar dess element enligt koden som följer:

```
1  private void reduceMatrix(BranchAndBoundNodeTSP matrix) {
2   int numberOfCities=matrix.numberOfCities();
3   for(int i=0;i<numberOfCities;i++) {
4    List<Double> list=matrix.getRow(i);
5    double d=reduceRowOrColumn(list);
6    for(int k=0;k<numberOfCities;k++) {
7     matrix.setElementAtIndex(i, k, list.get(k));
8    }
9    if(d!=Double.MAX_VALUE) {
10    matrix.addToReducedCost(d);
11   }
12  }
13  for(int i=0;i<numberOfCities;i++) {
14   List<Double> list=matrix.getColumn(i);
15   double d=reduceRowOrColumn(list);
16   for(int k=0;k<numberOfCities;k++) {
17    matrix.setElementAtIndex(k,i, list.get(k));
18   }
19   if(d!=Double.MAX_VALUE) {
20    matrix.addToReducedCost(d);
21   }
22  }
23 }
```

I `reduceMatrix` hittas det minsta elementet först för varje rad och sedan för varje kolonn. Om detta element då är nollskilt och inte heller har värdet `Double.MAX_VALUE`, som här får representera oändligheten, så dras kostnaden ifrån värdet på alla andra element i raden respektive kolonnen. Metoden för att hitta den reducerade kostnaden, det vill säga elementets kostnad, och för att uppdatera alla värden i `list` ovan heter `reduceRowOrColumn`. Den innehåller följande kod:

```
1  private double reduceRowOrColumn(List<Double> list) {
2   double d=findMin(list);
3   if(d>0 && d!=Double.MAX_VALUE) {
4    for(int i=0;i<list.size();i++) {
5     double f=list.get(i);
6     if(f!=Double.MAX_VALUE) {
7      list.set(i, f-d);
```

[12] Klass för att samla nodspecifika data för handelsresandeproblemet, se appendix A.

```
8       }
9      }
10    }
11    return d;
12   }
```

Alla de reducerade kostnaderna läggs sedan till i `BranchAndBoundNodeTSP`:s `sumOfReducedCost`, en variabel för att hålla reda på den totala reducerade kostnaden, för att kunna fungera som en optimistisk gräns för målfunktionsvärdet.

9.2 Test för att se om noden innehåller en tillåten lösning

För att se om de bågar som plockats ut med metoden `getArcsOfSolution` bildar en tillåten handelsresandetur anropas metoden `checkIfAllowedSolution`. I metoden hämtas först den första bågen, vars startnod läggs till i en lista över besökta städer, ifrån listan över bågar som `currentEvaluatedNode`:s, det vill säga den för närvarande senast evaluerade nodens, lösning innehåller. Därefter letas listan igenom efter en annan nod som har samma startnod som den första bågens slutnod. Om ingen sådan hittas returneras ”falskt”, det vill säga det är ingen tillåten tur. Om däremot en sådan båge hittas så blir den bågen som en ny första båge såvida dess slutnod inte finns i listan över besökta städer. I så fall returneras också ”falskt” om inte alla städer faktiskt är besökta en gång för då returneras givetvis ”sant”. I javakod ser det ut som följer:

```
1  private boolean checkIfAllowedSolution(BranchAndBoundNodeTSP matrix)
       {
2   List<Arc> arcs=getArcsOfSolution(matrix);
3   ArrayList<Integer> citiesVisited=new ArrayList<Integer>();
4   Arc currentArc=arcs.get(0);
5   int i=0;
6   while(i<matrix.numberOfCities()) {
7    Integer city=currentArc.from;
8    citiesVisited.add(city);
9    i++;
10   for(Arc a:arcs) {
11    if(a.from==currentArc.to) {
12     if(citiesVisited.size()==matrix.numberOfCities()-1 && a.to==
           citiesVisited.get(0)) {
13      return true;
14     }
15     currentArc=a;
16     if(citiesVisited.size()!=matrix.numberOfCities() && citiesVisited
           .contains(a.to)) {
17      return false;
18     }
19     break;
20    }
21   }
22  }
23  return true;
24 }
```

Om lösningen visade sig vara tillåten testas också om den optimistiska gränsen, `optimisticBound`, är mindre än den nuvarande övre gränsen, `upperBound`:

```
1  if(optimisticBound<upperBound) {
2   upperBound=optimisticBound;
3   bestSolution=currentEvaluatedNode.getListOfArcs();
4   if(currentOptimumNode!=null) {
5    currentOptimumNode.setCurrentOptimum(false);
6   }
7   currentOptimumNode=currentEvaluatedNode;
8   currentOptimumNode.setCurrentOptimum(true);
```

```
9    ArrayList<BranchAndBoundNodeTSP> temp=new ArrayList<
         BranchAndBoundNodeTSP>(nodeQueueDepthFirst);
10   for(BranchAndBoundNodeTSP bab: temp) {
11    BranchAndBoundNodeTSP m=bab.getParent();
12    if(m.getOptimisticBound()>=upperBound )   {
13     bab.getParentNode().setCut(true);
14     listOfCutNodes.add(m);
15     nodeQueueBreadthFirst.remove(bab);
16     nodeQueueBestFirst.remove(bab);
17     nodeQueueDepthFirst.remove(bab);
18    }
19   }
20  }
```

I så fall uppdateras `upperBound` och `bestSolution`. Dessutom tas noder vars förälder har en optimistisk gräns som är sämre än den övre gränsen bort från nodköerna `nodeQueueBreadthFirst`, `nodeQueueBestFirst` respektive `NodeQueueDepthFirst`. Detta eftersom föräldranoderna inte ska fortsätta att förgrenas över då en fortsatt förgrening inte kommer leda till ett bättre målfunktionsvärde än värdet på den optimistiska gränsen.

9.3 Test av avbrottskriterium

Om det visar sig att `currentEvaluatedNode` inte innehåller en tillåten lösning, men att dess optimistiska gräns samtidigt är större än `upperBound` sker ingen förgrening. Det går inte att hitta ett bättre målfunktionsvärde i den grenen. Kodmässigt ser testet ut så här:

```
1  else if(optimisticBound>=upperBound) {
2   updateOptimisticBounds();
3   currentEvaluatedNode.setCut(true);
4   listOfCutNodes.add(currentEvaluatedNode);
5   treePainter.addNodeButton(currentEvaluatedNode);
6   return;
7  }
```

Den tredje till femte raden ovan finns enbart där av grafiska skäl och har inget med algoritmen för att hitta optimum att göra.

9.4 Förgrening

Då noden inte innehåller en tillåten lösning eller då dess optimistiska gräns är mindre än den övre gränsen återstår bara vidare förgrening. Det första som sker är följaktligen att hitta vilken båge som ska förgrenas över. Det görs med ett anrop av metoden `branchArc`:

```
1  Arc branchArc=branchArc(currentEvaluatedNode);
```

där koden för `branchArc` ser ut som:

```
1  private Arc branchArc(BranchAndBoundNodeTSP matrix) {
2   Arc a=null;
3   double cost=-Double.MAX_VALUE;
4   for(int i=0;i<matrix.numberOfCities();i++) {
5    List<Double> rowi=matrix.getRow(i);
6    int k=0;
7    for(int j=0; j<rowi.size();j++) {
8     double d=rowi.get(j);
9     if(d==0) {
10     ArrayList<Double> tempRow=new ArrayList<Double>(rowi);
11     ArrayList<Double> tempColumn=new ArrayList<Double>(matrix.
           getColon(j));
12     tempRow.remove(j);
13     tempColumn.remove(i);
14     if(findMin(tempRow)!=Double.MAX_VALUE && findMin(tempColumn)!=
           Double.MAX_VALUE) {
```

```
15          double f=findMin(tempRow)+findMin(tempColumn);
16          if(f>cost) {
17           cost=f;
18           a=new Arc(i,j);
19          }
20         }
21        }
22        k++;
23       }
24      }
25      return a;
26     }
```

I metoden ovan beräknas för varje element c_{ij}, som har värdet 0, i `matrix`:s kostnadsmatris summan av det minsta värdet på rad i, borträknat element c_{ij}, och det minsta elementet på rad j, borträknat element c_{ij}. I avsnitt 7.3 betecknades denna summa $\Theta(i, j)$. Metoden returnerar sedan bågen med den största summan.

Därefter skapas höger- respektive vänsterbarn till `currentEvaluatedNode`. För vänsterbarnet sätts sedan elementet med samma index som `branchArc`[13] till `Double.MAX_VALUE` och `branchArc` läggs till i listan över noder som inte ska inkluderas. För högerbarnet sker desto mer:

```
1  right.addNewArc(branchArc);
2  right.setNewConstraintString(branchArc.toString());
3  for(int i=0;i<currentEvaluatedNode.numberOfCities();i++) {
4   right.setElementAtIndex(branchArc.from, i, Double.MAX_VALUE);
5   right.setElementAtIndex(i, branchArc.to, Double.MAX_VALUE);
6  }
7  right.setElementAtIndex(branchArc.to, branchArc.from, Double.
       MAX_VALUE);
8  removeSubTours(right);
```

Först och främst läggs förgreningsbågen, låt oss kalla den b_{ij}, till i listan över de bågar som ska tvingas till att vara med i den slutliga turen. Därtill sätts värdet på alla element i rad i och i kolonn j till `Double.MAX_VALUE`. Därefter ändras också värdet på element c_{ji} till `Double.MAX_VALUE` och dessutom ändras alla värden på bågar som skulle kunna göra att subturer uppstår genom anropet av `removeSubTours`, se *Borttagning av subturer*, avsnitt 9.5, nedan. Slutligen läggs dessa barnnoder in i sina respektive datastrukturer på samma sätt som för heltalsoptimeringsproblem, se avsnitt 7.1.

9.5 Borttagning av subturer

Metoden `removeSubTours`, som används för att ta bort subturer, börjar med att alla bågar sorteras så att de som bildar en sammanhängande väg kommer efter varandra i en lista:

```
1  List<Arc> arc=matrix.getListOfArcs();
2  List<Arc> sortedArc=new ArrayList<Arc>();
3  Arc temp=arc.get(0);
4  sortedArc.add(temp);
5  for(int k=0;k<arc.size();k++) {
6   boolean status=false;
7   for(Arc a:arc) {
8    if(temp.to==a.from && !sortedArc.contains(a)) {
9     temp=a;
10    sortedArc.add(temp);
11    status=true;
12    break;
13   }
14  }
15  if(!status) {
```

[13] Klassen har två instansvariabler som representerar start- och slutnod för en båge. Nodnumret för dessa två bildar sedan ett index kopplat till kostnadsmatrisen.

```
16      for(Arc b:arc) {
17       if(!sortedArc.contains(b)) {
18        temp=b;
19        sortedArc.add(b);
20        break;
21       }
22      }
23     }
24    }
```

Utan att gå in närmare i detalj på exakt vad som sker i ovanstående kod så skulle en ingående lista av bågar som från början sorterade enligt

1-2	4-3	2-5	3-2	6-1

,

hamna i följande ordning:

1-2	2-5	4-3	3-2	6-1

.

För varje båge, b_{ij}, i den nya sorterade listan, `sortedArc`, undersöks sedan ett antal kriterier:

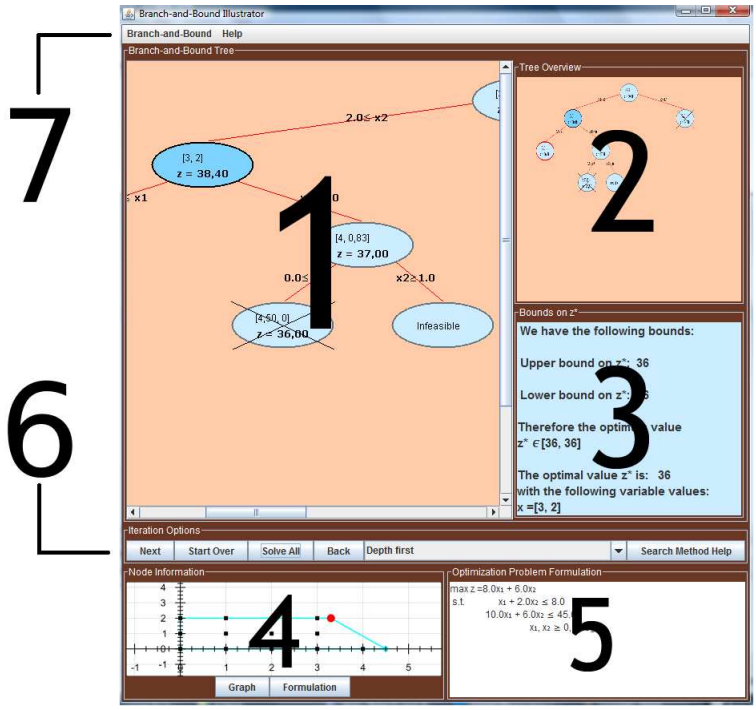
```
1   double max=Double.MAX_VALUE;
2   List<Arc> prev=new ArrayList<Arc>();
3   List<List<Arc>> listOfprev=new ArrayList<List<Arc>>();
4   prev.add(sortedArc.get(0));
5   matrix.setElementAtIndex(sortedArc.get(0).to, sortedArc.get(0).from,
        max);
6   for(int i=1;i<sortedArc.size();i++) {
7    Arc a=sortedArc.get(i);
8    matrix.setElementAtIndex(a.to, a.from, max);
9    if( !prev.isEmpty() && prev.get(prev.size()-1).to==a.from) {
10    for(Arc b:prev) {
11     matrix.setElementAtIndex(a.to, b.from, max);
12    }
13    prev.add(a);
14   }
15   else {
16    listOfprev.add(prev);
17    prev=new ArrayList<Arc>();
18    prev.add(a);
19   }
20   for(List<Arc> al:listOfprev) {
21    if(al.get(0).from==a.to) {
22     matrix.setElementAtIndex(al.get(al.size()-1).to,a.from,max);
23    }
24   }
25  }
```

Först sätts c_{ji} till `Double.MAX_VALUE`. Därefter undersöks om slutnoden för sista bågen i listan över sammanlänkade bågar, `prev`, är samma som startnoden för b_{ij}. Om så är fallet tas alla bågar som har en startnod lika med j och en slutnod lika med startnod för bågarna i `prev` bort. I annat fall läggs `prev` till i listan över sammanlänkade vägar, `listOfprev`, och `prev` startas om med b_{ij} som första nod i listan.

Det sista som undersöks är om det går att lägga till en nod mellan två vägar och på sätt få en subtur, se rad 20-23 i koden ovan. I så fall tas den bågen också bort från mängden av valbara bågar.

10 Upplägg av programfönstret ur ett pedagogiskt perspektiv

I grova drag är fönstret där trädet ritas upp och där användaren kan se stegen i Branch-and-Bound-algoritmen uppbyggd som i Figur 18.



Figur 18: Programfönstret. (1) Branch-and-Bound-trädet, (2) Översiktskarta, (3) Panel för övre och undre gränser på målfunktionsvärdet, (4) Nodinformationsrutan, (5) Originalproblemsrutan, (6) Knappar för att lösa problemet och (7) Menyn.

De olika delarna i Figur 18 kommer att beskrivas mer i detalj i kommande avsnitt där vi börjar med trädet från område 1.

10.1 Branch-and-Bound-trädet och Översiktskartan

Ett Branch-And-Bound-träd, se område 1 i Figur 18, kan bli både mycket djupt och mycket brett. Detta ställer krav på att vi kan rita upp trädet på ett smidigt sätt så att det fortfarande är möjligt att få en överblick av förgreningen. Samtidigt ska det också gå att få information om vad som händer i varje nod. Vår tanke är därför att trädet i sin helhet ritas upp i en panel som är scrollbar vilket gör att det bara går att se en del av trädet i taget men att informationen i de noder användaren väljer att titta på syns tydligt. [14] Därtill finns det en liten karta med trädet i miniatyr, se område 2 i Figur 18, som användaren samtidigt kan använda för att få en översikt över hela trädet.

Varje nod innehåller direkt synlig information om variabelvärdena i dess optimum samt optimalvärdet, texten *infeasible* om tillåten lösning saknas eller *unbounded* om området som ska optimeras över är obegränsat och optimallösningen går mot oändligheten. Dessutom innehåller den dold information, se *Nodinformationsrutan*, som användaren kan få tag på genom att klicka på noden. Därtill har den nod som just evaluerats en mörkare ton än övriga för att göra det extra tydligt var i algoritmen man för tillfället befinner sig. Därutöver för att markera att en nod inte kommer att förgrenas över ytterligare sätts ett stort kryss över den. Den för tillfället bästa tillåtna lösningen, om det finns en sådan, märks ut extra genom att en röd ring ritas kring den noden.

[14] De tekniska detaljerna för hur det här är implementerat finns i appendix B.

Anledningen till att inte all tillgänglig information finns synlig i noden är först och främst att noderna då skulle bli oerhört stora men också att underlätta för en nybörjare på området. Som nybörjare kan det ibland vara svårt att ta till sig allt för mycket fakta på en gång. Det är lätt att bli förvirrad, till exempel av alla olika bivillkor som kan uppkomma i nodinformationsrutan vid ILP-problem. Vi vill dock ge användaren tillgång till en grafisk representation av området som ska optimeras (om det är ett ILP-problem i två dimensioner) och möjlighet att se att fler och fler bivillkor tillkommer ju längre ner i trädet man kommer.

10.2 Panel för övre och undre gränser på optimalvärdet

Branch-and-Bound-algoritmen bygger på att noder i ett träd skärs bort enligt olika kriterier, bland annat genom jämförelse med undre gräns, vid maximeringsproblem, respektive övre gräns, för minimeringsproblem, på optimalvärdet. Att snabbt hitta dessa gränser i trädet kan vara svårt för en nybörjare. Trädet kan ju efter ett tag innehålla en stor mängd noder som det kan bli svårt att navigera bland och därför vill vi tydliggöra vilka gränserna är och vad det innebär för i vilket intervall det optimala målfunktionsvärdet kan ligga. Dessa gränser visas i område 3 i Figur 18.

I panelen visas också vad som blev optimallösningen när Branch-and-Bound-algoritmen gått igenom alla steg för det aktuella problemet.

10.3 Nodinformationsrutan

Nodinformationsrutan, se område 4 i Figur 18, innehåller information om den nod som precis klickats på eller just evaluerats. Då användaren valt ett ILP-problem finns två knappar, `Graph` och `Formulation`, kopplade till ruta 4 i Figur 18. Om användaren klickar på `Graph`-knappen ritas det tillåtna området upp i ett koordinatsystem, under förutsättning att problemet innehåller två variabler. På så sätt kan användaren jämföra tillåtna områden mellan förälder- och barnnod och se hur ett område som ändå inte innehåller några heltalspunkter skärs bort. För att se hur området beskrivs matematiskt kan användaren istället välja att klicka på knappen `Formulation`.

För handelsresandeproblem finns det istället information om summan av de reducerade kostnaderna samt den reducerade kostnadsmatrisen respektive en graf över de valda bågarna för den noden.

10.4 Orginalproblemsrutan

Område 5 i Figur 18 innehåller den matematiska formuleringen av optimeringsproblemet. Den finns med för att göra användaren påmind om vad det egentligen är för ursprungsproblem som ska lösas och för att användaren lätt ska kunna jämföra med problemen i respektive nod.

10.5 Knappar för att lösa problemet

I den knappanel som finns i område 6 i Figur 18 ställs användaren inför valet av vilken sökmetod han eller hon vill använda. Dessa metoder finns beskrivna i kapitel 7.1. För att kunna stega sig fram i algoritmen i egen takt finns knappen `Next`. Om användaren klickar på den så utförs ett steg i Branch-and-Bound-algoritmen med den valda sökmetoden. Vid stora problem kan det dock bli tröttsamt att stega sig fram i längden. Därför har vi infört möjligheten att lösa resterande del av problemet på en gång med knappen `Solve All`. Därutöver går det att starta om hela lösningsgången, vilket gör det möjligt att exempelvis jämföra hur trädet ser ut för bredd-först- respektive bäst-förstsökning. En ytterligare funktionalitet för att göra programmet mer pedagogiskt är möjligheten att backa ett steg i algoritmen och på så sätt kunna se skillnad på vilken nod som evalueras i de olika sökmetoderna.

10.6 Menyn

I menyn, område 7 i Figur 18, finns knapparna `Explanation of Branch-and-Bound for ILP` och `Explanation of Branch-and-Bound for TSP`, under `Help`, som ger en kortare förkla-

ring till hur Branch-and-Bound fungerar vilket kan ge ytterligare förståelse för algoritmen i stort. Det finns även en förklaring, också det under `Help`, till hur programmet kan användas samt möjlighet att välja ett nytt problem, spara Branch-and-Bound-trädet till en bildfil med mera.

11 Testning av Branch-and-Bound Illustrator

Oavsett vad för typ av programvara som ska konstrueras krävs en utförlig testning och utprovning. I de allra flesta fall ska både korrektheten av programmet och användarvänligheten testas, så även för vår programvara.

11.1 Korrekthet

Begreppet *korrekthet* syftar, i det här fallet, främst på att våra Branch-and-Bound-algoritmer ger optimala värden i enlighet med den matematiska teorin. Initialt, det vill säga i utvecklingsfasen av programvaran, har vi använt oss av ett par exempelproblem ur läroboken Optimeringslära ([7]) för att kontrollera Branch-and-Bound-algoritmen för heltalsoptimeringsproblemen. I dessa exempel finns tillgång till hela trädstrukturen för ett visst val av sökmetod och dessutom optimalvärdet. Att programmet fungerar som det ska för exempelproblemen ger dock inga garantier för att det ska fungera för alla andra heltalsoptimeringsproblem. För vidare kontroll av korrektheten har vi egentligen två utgångspunkter; kvantitativ testning och testning av specialfall.

Den kvantitativa testningen handlar om att låta många olika problem lösas av våra Branch-and-Bound-klasser och jämföra med det optimala värdet. Eftersom det handlar om stora mängder data som ska genereras och testas så räcker det inte att ta några få exempel från litteraturen. Istället har vi valt att lita på korrektheten hos den kommersiella optimeringslösaren *Cplex* [18], och utdata från denna för de generella heltalsoptimeringsproblemen. För att testa korrektheten av vår Branch-and-Bound-algoritm för handelsresandeproblemet bestämde vi oss för att beräkna kostnaden för samtliga möjliga turer genom att räkna upp dem.

11.1.1 Testning med hjälp av Cplex

För enkelhetens skull valde vi att enbart använda oss av heltalsoptimeringsproblem med två variabler. Motiveringen till detta är dels att tiden per problem som ska genereras och lösas minskar men också att de flesta svårigheter som skulle kunna inträffa sker redan med enbart två variabler. Dessutom var det mycket enklare implementeringsmässigt sätt att använda sig av ett fast antal variabler och valet föll då på två stycken.

Det vi gjorde var att slumpmässigt generera en målfunktion som ska maximeras[15], där målfunktionskoefficienterna fick variera mellan 0 och 1000, eftersom det inte blir någon skillnad algoritmmässigt om dessa koefficienter tillåts vara större. Vi lät däremot antal bivillkor variera slumpmässigt. Dock krävde vi att det alltid skulle finnas ett mindre än eller lika med villkor eftersom det annars skulle uppstå en allt för stor andel problem där området skulle vara obegränsat. De maximeringsproblem som genererades var alltså på formen att:

$$\begin{aligned} &\text{maximera } z = \sum_{i=1}^{2} c_i x_i, \\ &\text{då} \qquad\qquad A\mathbf{x} \leq \mathbf{b}, \\ &\qquad\qquad\quad D\mathbf{x} \geq \mathbf{e}, \\ &\qquad\qquad\quad F\mathbf{x} = \mathbf{g}, \end{aligned}$$

där A har storleken $m \times 2$, där m kan variera mellan ett och tre, D har storleken $n \times 2$, där n kan variera mellan noll och tre, F är av storlek $j \times j$ med j varierande mellan noll och ett, och $\mathbf{b}$, $\mathbf{e}$, och $\mathbf{g}$ motsvarande högerled. Elementen i matriserna A, D och F är slumpmässigt genererade heltal mellan -100 och 100. I motsvarande högerled, det vill säga i vektorerna $\mathbf{b}$, $\mathbf{e}$ och $\mathbf{g}$, kan heltalen variera mellan -1000 och 1000 istället. Anledningen är att det ger ett större spann för målfunktionskoefficientsvariablerna. Dessutom hade varje variabel en övre

[15] Vårt program är byggt helt analogt för minimeringsproblem varför vi valde att bara testa för maximeringsproblem.

och undre gräns mellan -100 och 100 eller så har de varit obegränsade uppåt och/eller nedåt, i datorn representerat med det största värde som får plats i en `double`.

Sammanlagt slumpades över 80 000 testfall fram utan att vår algoritm gav ett annat svar än vad Cplex gjorde mer än i något enstaka fall. Dessa fel berodde på vår lösares, det vill säga QSopts [16], noggrannhet och är inget vi kan styra över. Mer kommentarer om varför vi valde att använda denna i alla fall finns i diskussionsavsnittet, avsnitt 12. Det ska dock tilläggas att i cirka 85% av fallen saknade problemet tillåten lösning vilket inte testar speciellt många utfall.

11.1.2 Autogenererad testning för handelsresandeproblemet

Testningen av vår implementering av den Branch-and-Bound-algoritm för handelsresandeproblemet som bygger på reducering av kostnadsmatrisen gick ut på att slumpmässigt generera en kostnadsmatris av storlek 6×6 där varje element, förutom elementen på diagonalen i matrisen, var ett slumptal mellan 0 och 1000. Optimalturen för varje matris beräknades sedan genom att räkna upp alla möjliga turer och plocka ut turen med lägst kostnad.

Vi lät därefter vår algoritm räkna ut kostnaden för den optimala handelsresandeturen och jämförde med värdet som fåtts genom uppräkning av samtliga turer. Testet upprepades över tre miljoner gånger och varje gång gav vår algoritm ut det optimala värdet.

11.1.3 Testning av specialfall

Vid slumpmässig generering av problem kan det hända att vissa typer av heltalsoptimeringsproblem, som vårt program kan komma att användas för att lösa och illustrera, inte testas tillräckligt utförligt om de ens testas överhuvudtaget. Därför krävs det att dessa specialfall undersöks separat.

För handelsresandeproblemet är ett av dessa fall att samtliga bågkostnader är lika stora, vilket testades genom att generera en matris där samtliga bågkostnader, förutom de på diagonalen i kostnadsmatrisen, fick samma värde. Inte heller undersöktes om programmet klarar av handelsresandeproblem som saknar lösning med den autogenererade testningen. Detta specialfall testades genom att mata in problem där en hel rad eller kolonn i kostnadsmatrisen sattes till oändligheten[16].

Ett exempel på något som eventuellt kunde ha orsakat problem för vår Branch-and-Bound-algoritm för ILP-problem är om problemet som matades in var degenererat. Vi har inte gjort någon omfattande undersökning för att testa så att det inte finns några undantagsfall som vår lösare inte klarar av, men vi har prövat att mata in ett par mer eller mindre degenererade[17] problem. För dessa har vi fått ut korrekta värden. Värt att nämna är att det egentligen inte är vår Branch-and-Bound-implementation det hänger på i det här fallet utan på att vår externa lösare, QSopt [16], klarar av att finna optimallösning även för degenererade problem.

11.2 Användarvänlighet

Att utforma och testa ett programs användarvänlighet är något helt annat än att kontrollera dess korrekthet. Här handlar det om att undersöka huruvida programmet är lätt att förstå och se hur det kan användas. Den här testningen är inget vi har kunnat genomföra själva utan vi har tagit hjälp av personer utifrån. Ett stort tack ska riktas till våra klasskamrater på Teknisk matematik som varit till stor hjälp.

Testningen gick ut på att våra försökspersoner, cirka tio till antalet, först helt utan hjälp fick försöka navigera i programmet. När användaren förstått programmets grundmoment gav vi mindre instruktioner, till exempel att starta ett nytt heltalsoptimeringsproblem med två variabler.

Det generella intrycket försökspersonerna gav över programmet var positivt. De tyckte ofta att det såg bra ut och att programmet hade de funktioner som de förväntade sig. Dock uppdagades det att det fanns problem med att hela huvudprogramfönstret inte syntes på

[16] Egentligen `Double.MAX_VALUE`.
[17] Se avsnitt 5.3 om degeneration.

mindre skärmar, men att mata in problem och starta upp exempelproblem gick lätt för alla som testade oavsett förkunskaper inom optimeringslära. Dock var tröskeln för att förstå och hitta i huvudfönstret förhållandevis hög. Nästan alla testare var tvungna att trycka på de olika panelerna i ett par minuter för att förstå. En tredjedel hade problem med att hitta hur man kan stega sig fram i algoritmen. Alla kom dock fram till hur de skulle bära sig åt efter en liten stund.

De försökspersoner som inte sysslat så mycket med optimering innan kunde inte förstå hur stegen fungerade, speciellt inte för handelsresandeproblemet, men de hittade till den hjälptext där informationen fanns. Även de som hade grundläggande kunskaper i optimering kunde ha problem att förstå vad som händer i varje steg i algoritmen för handelsresandeproblemet. Däremot hade de inga problem att se när en giltig handelsresanderutt hittats. Främst bidrog uppritningsverktyget för handelsresandeproblemet till denna förståelse.

Det användare förstod i särklass bäst var heltalsoptimeringsproblem med två variabler. Förgreningen blir då inte så svår att förstå när testaren kan se hur området som optimeras över krymper grafiskt. Positiv respons fick vi också på enkelheten i inmatning av egna problem. Det var lätt förstå vad som förväntades av dig som användare. Dessutom upplevdes översiktskartan som lättförståelig och användarna tyckte sig få en bra överblick av trädet genom den.

En sak vi fick lära oss av att studera testarna var att många av programmets funktioner inte var uppenbara. Till exempel förstod ingen av användarna att noder kan klickas på eller att det går att zooma med mushjulet i plotverktyget för heltalsoptimeringsproblemen. Dessutom var det tydligt att nästan alla hade problem med att upptäcka när algoritmen terminerar och optimalvärdet hittas.

Diskussion och slutsats

12 Diskussion

12.1 Val vid implementering

Redan i början av projektet var vi tvungna att göra val kring hur vi skulle gå till väga för att på bästa sätt skapa en lättanvänd, lättillgänglig och tilltalande programvara. Det första implementeringsvalet handlade om att bestämma programmeringsspråk: *Java* eller *Matlab*. I Matlab finns många funktioner inbyggda, det går därför snabbt att skapa program och det är inte heller speciellt svårt. Ett problem är dock att det är svårt att specialdesigna grafik i Matlab. Det går att göra mycket i Matlab men över lag är programmeraren någorlunda låst till de metoder som redan finns inbyggda. I det språk vi tillslut valde, Java, tar det längre tid att skapa ett körbart grundprogram. Fördelen är att man med Java kan skapa vilka program eller metoder som helst om tillräckligt med tid läggs ner. Möjligheten att specialdesigna egna knappar och att göra olika typer av uppritare är bra mycket större i Java. Slutligen är Java ett objektorienterat programmeringsspråk vilket lämpar sig väl vid skapandet av stora program som kräver god struktur. En av de största anledningarna till varför valet föll på att skriva i Java var att programmet då kommer att vara körbart överallt. Den programvara som krävs för att köra Java-applikationer finns på nästan alla datorer världen över. Om programmet hade varit skrivet i Matlab så skulle antalet potentiella användare minskat markant då utnyttjande av Matlab kräver en licens.

Det andra implementationsvalet vi ställdes inför gällde lösning av linjärprogrammeringsproblem. Vilken lösare ska användas för att hitta optimum till LP-problem med hjälp av simplexmetoden? Efter att ha utvärderat ett antal alternativ begränsade vi oss till två olika lösare: *Cplex* och *QSopt*. För den kommersiella lösaren Cplex finns ett användargränssnitt för Java och det råder inget tvivel om att Cplex har en högre noggrannhet än QSopt, men det krävs att användaren har en licens. För att kunna publicera projektet på internet kan vi således inte använda Cplex, utan måste välja en lösare såsom QSopt, som är tillåten att använda i utbildningssyfte.

Det finns många olika typer av Branch-and-Bound-algoritmer som hade gått att implementera på samma sätt som de två typer av problem som är med i programmet. Till exempel hade vi kunnat implementera speciella Branch-and-Bound-tekniker för kappsäcksproblemet eller olika typer av schemaläggningsproblem. Dessutom finns det många olika intressanta relaxeringar till handelsresandeproblemet som skulle kunnat implementerats. Istället valde vi att först implementera algoritmen som löser generella heltalsoptimeringsproblem, som då också kan användas för att hitta optimum för många av specialfallen[18]. Dessutom är den grafiska lösningsgången förhållandevis lätt att följa så att användare kan förstå principen bakom Branch-and-Bound. Som vårt andra problem bestämde vi oss för att implementera handelsresandeproblemet där relaxeringen sker med reducerade kostnadsmatriser, se avsnitt 7.3.4. Skälet till att vi valde denna lösningsmetod var för att relaxeringen skapar binära träd så att vår nuvarande implementation av trädgrafiken i programmet fortfarande kan användas utan större modifikation.

12.2 Begränsningar i programvaran

Även om programvaran innehåller en hel del funktioner finns också vissa begränsningar. Till exempel får generella heltalsoptimeringsproblem maximalt ha sju variabler och nio bivillkor av vardera typ, vilket resulterar i att programmet inte kan lösa allt för stora problem. På samma sätt kan man vid inmatning av en egen kostnadsmatris för handelsresandeproblem maximalt välja sju städer. Detta är ett medvetet val som tvingar ner storleken på problemen. Stora problem är delvis svårare att förstå och blir på så sätt opedagogiska och om träden blir

[18] Kappsäcksproblem, handelsresandeproblem och schemaläggningsproblem

för djupa följer ett minnesproblem i Java. Problemet bottnar i hur översiktskartan uppdateras i varje iteration av Branch-and-Bound-algoritmen. I avsnitt B.2 beskrivs hur vi skapade översiktskartan med hjälp utav objektet `BufferedImage` och hur det ledde fram till minnesläckan. Det visade sig att denna minnesläcka var ett känt problem i Java och är något som antingen måste kringgås eller en begränsning som får accepteras. I vårt fall fanns ingen bra alternativ lösning, istället lade vi till en dialogruta som varnar användaren när ett trädet börjar bli för stort för Java att hantera.

En funktionalitet i programmet som underlättar inlärningen av hur Branch-and-Bound fungerar, men som också är begränsad, är att låta användaren backa i algoritmen. Bara att implementera möjligheten att backa ett steg är tillräckligt komplicerat, då det krävs att programmets tillstånd sparas för varje gång en ny nod evalueras i algoritmen på ett effektivt sätt. Därför finns det inte möjlighet att gå bak mer än ett steg.

Under testningen av vår Branch-and-Bound-algoritm upptäckte vi att QSopt, det vill säga den externa lösare vi använder för att lösa linjärprogrammeringsproblemen utan heltalskrav, har en förhållandevis låg noggrannhet vad gäller optimallösningens värde. Det innebär att den toleransnivå vi har i felmarginal måste vara förhållandevis stor vilket i sin tur leder till att vissa problem inte löses helt korrekt. Tyvärr kan vi inte styra över QSopt:s toleransnivå utan det är något vi måste acceptera då vi valt att använda denna lösare.

Dock är problemet kanske inte riktigt så stort som det låter. De optimeringsproblem som gör att QSopt:s toleransnivå inte räcker till är av sådan natur att en nybörjare inom Branch-and-Bound med stor sannolikhet ändå inte skulle ha någon större nytta av dess illustrering. Det beror på att trädet för den här typen av problem blir väldigt djupt. Ett djupt träd tillför i princip ingenting teoretiskt sätt mer än att illustrera att heltalsoptimeringsproblem ibland kan vara mycket komplexa att lösa.

12.3 Pedagogiskt resonemang

Eftersom målet med programmet är att det ska kunna fungera som ett verktyg för att lära ut Branch-and-Bound-metoden har en hel del arbete lagts ner för att göra programmet så pedagogiskt som möjligt. För att nå detta mål krävdes ett genomtänkt programupplägg, med fokus på inlärningsvärdet. En del i upplägget var att förse användaren med en rimlig mängd information samtidigt som möjligheten ska finnas för användaren att få mer information om teorin bakom Branch-and-Bound. Vi valde därför att lägga in förklaringstexter i programmet som finns tillgängliga via menyraden. En utgångspunkt vid skapandet av användargränsnittet var att knappar och fönster skulle vara självförklarande men för att ytterligare göra programmet mer användarvänligt skapade vi även en hjälp-ruta som också är tillgänglig via menyraden.

Vid positioneringen och uppbygganden av de olika panelerna i huvudfönstret var idéen att trädet, som Branch-and-Bound-algoritmen resulterar i, skulle ligga centralt och ha den största delen av huvudfönstret. Det var ett medvetet val precis som att vi valde att visa variabelvärdena och målfunktionsvärdet i varje nod. Detta efter att noggrant ha övervägt vilken information som är mest relevant för en nybörjare. Bland annat diskuterades huruvida variabelvärdena skulle vara med eller inte. För handelsresandeproblemet fanns det en liknande diskussion kring om de utvalda bågarna eller de borttagna skulle synas i noden. Ytterligare ett diskussionsämne var en övervägning mellan att ge användaren möjlighet att backa och att numrera nodernas evalueringsordning.

12.4 Förbättrings- och utvecklingspotential

I början av projektet hade vi förslag på vilka problemtyper som skulle kunna finnas med i programmet om det vore tidsmässigt möjligt. Efter att ha kommit igång med arbetet beslutade vi oss för att lägga fokus på det viktigaste, Branch-and-Bound för generella heltalsoptimeringsproblem. Under processens gång kom vi även på nya funktioner som vi inte tänkt på från början men som ändå lyfter helhetsintrycket av programmet, till exempel att backa ett steg i algoritmen.

Ett av de största områden för utveckling är antalet olika problem som går att lösa med Branch-and-Bound och vår programvara. Om grafiken utvecklas, så att träduppritaren kan

rita andra träd än binära, blir svårigheten med att lägga till nya problemtyper till stor del att översätta den matematiska algoritmen till javakod.

En annan del som skulle kunna förbättras är att göra det grafiska tydligare på ytterligare ett antal punkter:

- Till exempel skulle knapparna för att välja problemtyp kunna specialdesignas med en liten bild på problemstrukturen. Strukturen på hur de olika delarna ligger i huvudprogramfönstret skulle kanske även den kunna vara bättre utformad, med nya positioner för de olika komponenterna eller med mer tilltalande knappar i algoritmpanelen.
- En annan grafisk förbättring hade varit att alltid skala huvudprogramfönstret utefter storleken på skärmen programmet körs i.
- En funktionalitet som saknas men som vore tillsynes användbar att ha med är navigation genom att klicka på en position i översiktskartan och på så sätt få den delen centrerad i trädpanelen. Nästan alla användare som testade programmet försökte klicka på översiktskartan för att styra över vilken del av trädet som visas.
- För inmatning av handelsresandeproblem skulle en utveckling kunna ske genom att låta användaren mata in euklidiska handelsresandeproblem via en klickbar karta.
- För heltalsoptimeringsproblem skulle man kunna ha en uppritare för problem med tre variabler genom att göra samma steg som i uppritaren för två dimensioner men att göra uppritningen i openGL för att kunna rita områden och ytor i 3D.

Slutligen skulle vi även kunna implementera en egen simplexalgoritm där noggrannheten vore möjlig att styra. Anledningen är att det numeriska värde som QSopt:s lösare returnerar inte är tillräckligt exakt, se också avsnitt 12.2.

13 Slutsats

I slutändan blev programvaran som planerat. De första utkasten på programstruktur och framförallt huvudfönsterstruktur stämmer överens med resultatet. Programmet går, också det som planerat, att köras som en applikation via ett webbfönster och kräver inga specifika programvaror eller licenser, förutom Java, för att köras.

Referenser

[1] Camick R. Screen Image; 2008. Javaklass, hämtad mars 2012. http://tips4java.wordpress.com/2008/10/13/screen-image/.

[2] Land AH, Doig AG. An Automatic Method of Solving Discrete Programming Problems. Econometrica (före-1986). 1960;28(3):sid. 497–521.

[3] DPS International. Tour Optimiser; 2012. Programvara för fordonsschemaläggning, hämtat maj 2012. http://www.dps-int.com/products/web/route-optimising-software.php?ref=t%opNav.

[4] Patriksson M. Lecture 1, The travelling salesman problem, II; 2011. Powerpointpresentation, hämtad april 2012. http://www.math.chalmers.se/Math/Grundutb/CTH/tma947/1011/lecture1.pdf.

[5] Eclipse Foundation. The Eclipse project; 2001. Utvecklingsverktyg för Java, hämtat januari 2012. http://www.eclipse.org/org/.

[6] Karmarkar N. A new polynomial-time algorithm for linear programming. In: Proceedings of the sixteenth annual ACM symposium on Theory of computing. STOC '84. New York, NY, USA: ACM; 1984. p. 302–311. Available from: http://doi.acm.org/10.1145/800057.808695.

[7] Lundgren J, Rönnqvist M, Värbrand P. Optimeringslära. Tredje upplagan ed. Lund: Studentlitteratur; 2008.

[8] Dai Y. Simplex Method : technique aspects; 2012. Föreläsningsanteckningar, hämtat maj 2012. http://array.bioengr.uic.edu/~yangdai/teach/bioe494-fall02/note3.pdf.

[9] Bland RG. New finite pivoting rules for the simplex method. Mathematics of Operations Reaserch. 1977;2(2):sid. 103–107.

[10] Sierksma G. Linear and Integer Programming – Theory and Practice. Andra upplagan ed. Monographs and Textbooks in Pure and Applied Mathematics: 245. Marcel Dekker, Inc.; 1988.

[11] Rardin RL. Optimization in Operations Research. Prentice Hall, Inc.; 1998.

[12] Nemhauser GL, Wolsey LA. Integer and Combinatorial Optimization. Wiley Interscience Series In Discrete Mathematics And Optimization. John Wiley & Sons; 1988.

[13] Busby, Dodge, Fleming, Negrusa. Backtracking & Branch and Bound; 2012. Powerpointpresentation, hämtad april 2012. http://www.academic.marist.edu/~jzbv/algorithms/StudentProjects/Backtracking%andBandB.ppt.

[14] Balas E, Toth P. Branch and Bound Methods for the Traveling Salesman problem. Carnegie-Mellan University Pittsburgh PA Management Science Reserah Group; 1983. ADA126957.

[15] Little JDC, Murty KG, Sweeney DW, Karel C. An Algorithm for the Traveling Salesman Problem. Operations Research. 1963;11(6):sid. 972–989. Available from: http://www.jstor.org/stable/167836.

[16] Applegate D, Cook W, Dash S, Mevenkamp M. QSopt library; 2012. Javabibliotek, hämtat februari 2012. http://www2.isye.gatech.edu/~wcook/qsopt/index.html.

[17] Oracle. Java Programming Language Application Programming Interfaces; 1993. Klassmanual för Java, hämtad februari 2012. http://docs.oracle.com/javase/6/docs/api/.

[18] IBM ILOG CPLEX Optimizer. Cplex Optimizer; 2012. Optimeringslösare, hämtat april 2012. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer%/.

[19] Naveh B. JGraphT; 2012. Javabibliotek, hämtat februari 2012. http://jgrapht.org/.

[20] Seifert E. GRAphing Library; 2012. Javabibliotek, hämtat februari 2012. http://trac.erichseifert.de/gral/wiki.

[21] Free Software Foundation; 2007. GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl-3.0.html.

[22] Råde L, Westergren B. Mathematics Handbook for science and engineering. Studentlitteratur; 1988.

[23] Oracle. Java Web Start; 2001. Ramverk för webbapplikationer, hämtat mars 2012. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136112.htm%l.

[24] Oracle. Java Applet; 1995. Ramverk för webbapplikationer, hämtat mars 2012. http://java.sun.com/applets/.

[25] Oracle. Java Network Launching Protocol; 2001. Protokoll för JavaWS, hämtat mars 2012. http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/jnl%p.html.

[26] Oracle. Java Runtime Environment; 1995. Kompileringsmiljö, hämtat mars 2012. http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Appendix

A Programstruktur

Startmeny
Nedan listas alla klasser som används, i det skeende av programmet, där användaren ännu inte har matat in vilket specifikt problem som ska lösas.

`SelectionFrameController` - Klass som kontrollerar vilket fönster som ska visas innan ett specifikt problem är valt. `SelectionFrameController` lyssnar på alla knappar från fönster den öppnar och hanterar vad som händer när man trycker på dem.

`WelcomeFrame` - Första fönstret som visas när programmet startas upp. Låter användaren välja mellan att lösa ett heltalsoptimeringsproblem och ett handelsresandeproblem.

`ILPFrame` - Fönster som visas om användaren väljer att lösa ett heltalsoptimeringsproblem. Låter användaren välja mellan två färdiga exempel: att öppna ett problem från en fil eller att mata in ett eget problem.

`ILPOwnProblemFrame` - Fönster som visas om användaren väljer att mata in ett eget heltalsoptimeringsproblem. I denna ruta får användaren välja hur många variabler och bivillkor problemet ska ha.

`ILPProblemInputFrame` - Fönster som visas efter det att användaren har matat in hur stort heltalsoptimeringsproblemet ska vara. Här matas vilka värden de olika bivillkoren ska ha, vad målfunktionen ska vara samt vilka begränsningar de olika variablerna ska ha in. När inmatningen är klar kan användaren antingen spara problemet till en `.ilp-fil` eller starta programmet för det givna problemet.

`TSPFrame` - Fönster som visas om användaren väljer att lösa ett handelsresandeproblem. Användaren får välja mellan att lösa ett exempelproblem eller ett eget problem. I valet av eget problem kan användaren välja hur många städer det egeninmatade problemet ska ha.

`TSPProblemInputFrame` - Om användaren väljer att mata in ett eget handelsresandeproblem visas detta fönster. Användaren får nu mata in värden i en färdig matris som beskriver bågkostnaden mellan olika städer. När inmatningen är klar kan användaren antingen spara problemet till en `.tsp-fil` eller starta programmet för det givna problemet.

`OpenFileFrame` - Fönster som visas om användaren väljer att öppna ett problem från en fil.

`SaveFileFrame` - Fönster som visas om användaren väljer att spara ett problem till fil.

Grafik
Här listas alla klasser som används för att grafiskt visa hur algoritmen fungerar. Det är klasser som sköter uppritning av trädet och dess noder, översiktskartan, hjälptexter och annat som visas efter det att användaren har matat in vilket problem som ska lösas.

`MainFrame` - Huvudfönstret som innehåller alla andra små fönster och knappar. Vissa funktioner är implementerade rakt i `MainFrame`-klassen, såsom översiktskartan och fönstret som visar övre och undre gränser för målfunktionen, medan andra ligger i separata klasser. I översiktskartan kopieras och skalas innehållet i `TreePainter`, som ritar upp trädstrukturen, till en bild.

`MenuBarForMainFrame` - Klass för huvudfönstrets meny.

`MainFrameController` - Klass som lyssnar och reagerar på de olika knapparna i `MainFrame`. Från `MainFrameController` uppdateras sedan `MainFrame`.

`TreePainter` - Fönster som innehåller alla noder och alla kanter. `Treepainter` räknar var i fönstret olika noder och bågar ska ligga för att det ska se snyggt ut och håller koll på om fönstret behöver bli bredare eller högre.

`NodeButton` - Klass för att skapa en nod som också fungerar som knapp. I klassen fås nodspecifik information från instanser av `BranchAndBoundNodeInterface` .

`NodeEdge` - Klass för att rita bågarna som sammankopplar noder.

`ScreenImage` - Klass som används för att kopiera bilder av uppritade javaobjekt av typen `JComponent`[17]. Används i `MainFrame` för att skapa översiktskartan och är skriven av R Camick [1].

`HowToUseTheProgramFrame` - Om användaren från menyn i `MainFrame` väljer först ”Help” och sedan "How to use the program", kommer detta fönster visas. I detta fönster finns en replika av knapparna i `MainFrame` med förklarande texter som beskriver vad var och en av knapparna gör.

`SearchMethodInformation` - Fönster som beskriver hur varje sökmetod fungerar.

`Index` - Klass som används för att kunna skriva ut nersänkta siffror, det vill säga index, i Java.

`ILPExplanationFrame` - Om användaren, körandes ett heltalsoptimeringsproblem, från menyn i `MainFrame` väljer först ”Help” och därefter ”Explanation of the Branch-and-Bound algorithm for ILP”, kommer detta fönster visas. I fönstret finns en bild som beskriver hur de olika stegen av Branch-and-Bound-algoritmen för heltalsoptimeringsproblem fungerar.

`TSPExplanationFrame` - Om användaren, körandes ett handelsresandeproblem, från menyn i `MainFrame` väljer först ”Help” och därefter ”Explanation of the Branch-and-Bound algorithm for TSP”, kommer detta fönster visas. I fönstret finns en bild som beskriver hur de olika stegen av Branch-and-Bound-algoritmen för handelsresandeproblemet fungerar.

`ILPPlotPanel` - Klass som ritar upp områden för heltalsoptimeringsproblem med två variabler.

Algoritm
Här ingår alla klasser som har med implementeringen av Branch-and-Bound-algoritmen att göra.

`BranchAndBoundILP` - Klass som utför Branch-and-Bound för heltalsoptimeringsproblem.

`BranchAndBoundNodeILP` - Klass för att samla all nodspecifik information i Branch-and-Bound-algoritmen för heltalsoptimeringsproblem.

`LPFormulation` - Klass som innehåller samlade data för ett linjärprogrammeringsproblem.

`ILPSolver` - Klass som löser ett linjärprogrammeringsproblem med simplexmetoden. Använder den externa lösaren `QSopt`[16] för detta ändamål.

`BranchAndBoundTSP` - Klass som utför Branch-and-Bound-stegen för handelsresandeproblemet.

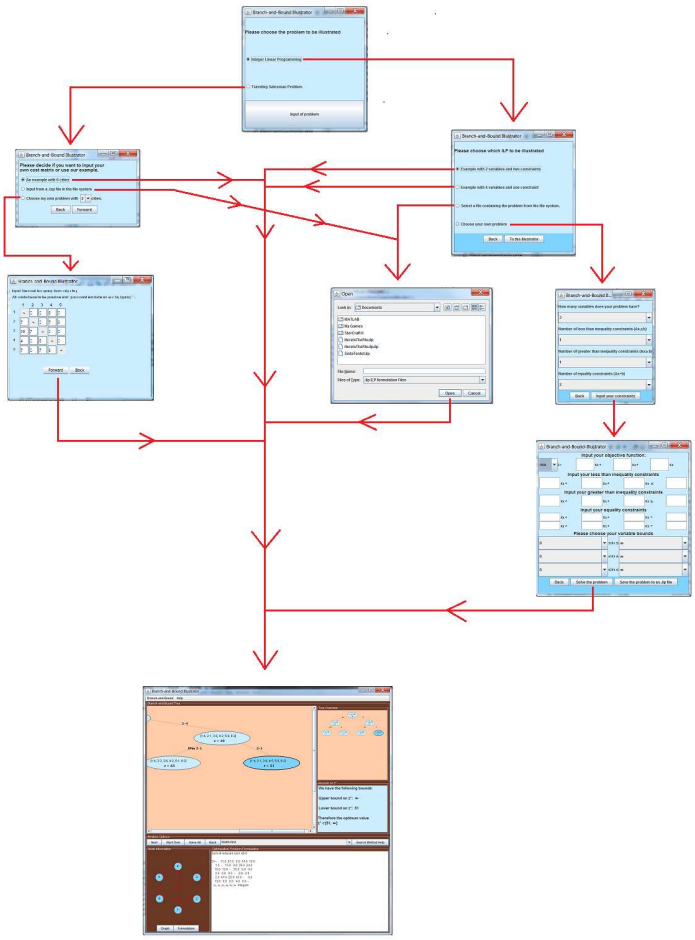
`BranchAndBoundNodeTSP` - Klass för att spara all specifik noddata i Branch-and-Bound-algoritmen för handelsresandeproblemet.

`Arc` - Klass som beskriver bågar i en graf. Används bland annat i grafrepresentationen av handelsresandeproblem.

`TSPGraph` - Fönster som ritar upp de optimala turerna för relaxerade handelsresandeproblem.

`BranchAndBoundInterface` - Interface för Branch-and-Bound-algoritmer.

`BranchAndBoundNodeInterface` - Interface för Branch-and-Bound-noder.

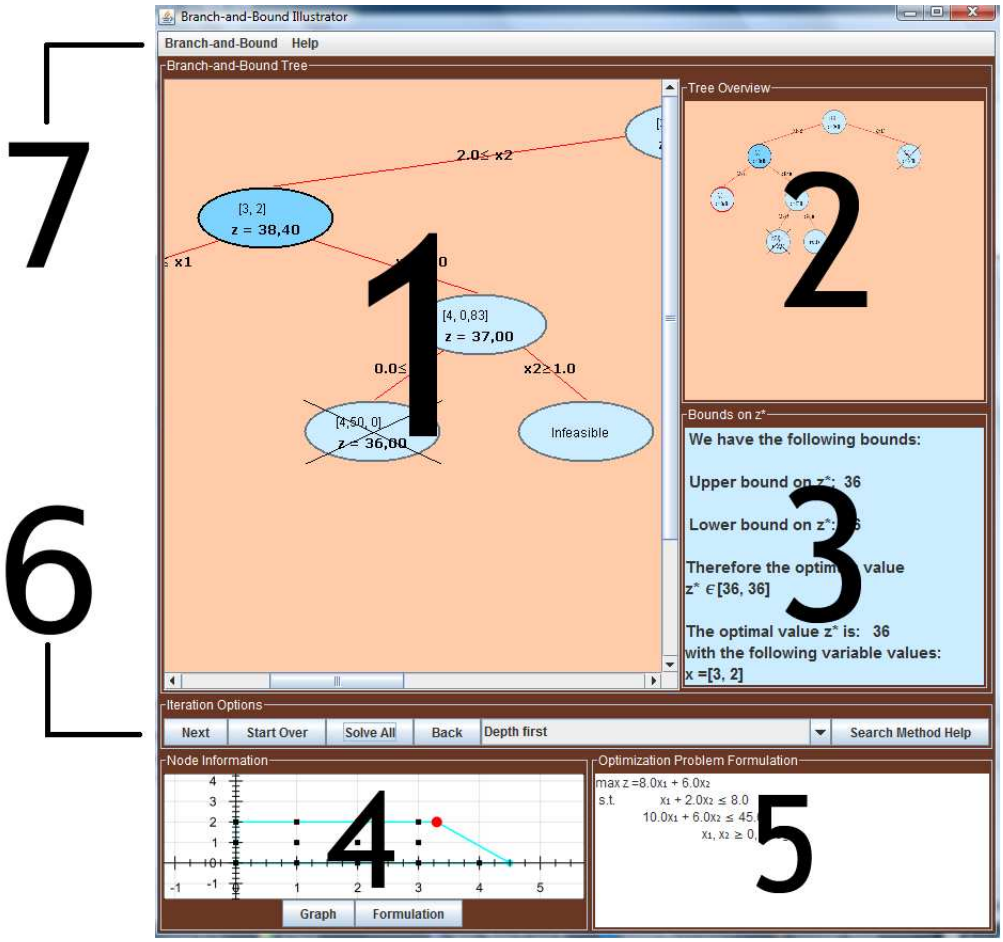


Figur 19: Beskrivning av fönsterna som används för att navigera fram till `MainFrame`.

B Tekniska aspekter kring implementeringen

I följande bilaga förklaras en del av de tekniska aspekterna kring implementeringen av vårt program *Branch-and-Bound Illustrator*. Eftersom programvaran innehåller en stor mängd kod som i sig inte är så relevant för varken Branch-and-Bound-algoritmen eller det pedagogiska perspektivet men som är av stor betydelse för helhetsintrycket av programmet kommer vi här ta upp de viktigaste byggstenarna för att få huvudfönstret att fungera. Vissa av dessa delar innehåller en del intressant matematik och en stor del av arbetstiden lades på att lösa de problem som uppstod i varje stadie av uppbyggnaden. En del av dessa diskuteras i korthet nedan.

För en fullständig förståelse för programmets olika delar hänvisar vi till koden samt dess kommentarer.



Figur 20: Programfönstret. (1) Branch-and-Bound-trädet, (2) Översiktskarta, (3) Panel för övre och undre gränser på målfunktionsvärdet, (4) Nodinformationsrutan, (5) Originalproblemsrutan, (6) Knappar för att lösa problemet och (7) Menyn.

B.1 Branch-and-Bound-trädet

Vid valet av hur vi skulle rita upp det träd som representeras i Branch-and-Bound-algoritmen hade vi två alternativ. Det första var att använda ett färdigskrivet paket för uppritning av grafer, i detta fallet studerade vi paketet JGraphT [19], eller så kunde vi skapa en egen uppritare. Valet föll på det senare då vi upptäckte att en del funktioner som vi hade behövt saknades i JGraphT. Ett exempel är en algoritm för att rita binära träd. Dessutom ville

vi ha möjligheten att konfigurera uppritningen för binära träd på ett sätt som passade in i huvudfönstret.

Idéen bakom uppbyggnaden av trädet gick ut på att låta varje nod vara ett objekt av interfacet `BranchAndBoundNodeInterface`. På så vis kan vi styra vilket innehåll varje enskild nod ska ha, till exempel fick varje nod ett eget målfunktionsvärde kopplat till lösningen av det relevanta relaxerade problemet.

Förutom vad som nämndes ovan får varje nod också en höjdparameter samt ett index i det horisontella ledet. Med hjälp utav dessa två parametrar kan vi skapa en formel för positionen för nya noder i trädet i förhållande till fönstrets storlek och sedan beräkna den enligt nedan:

```
1  private void setNodeButtonPosition(BranchAndBoundNodeInterface node){
2   int numberOfVariables = node.getNumberOfVariables();
3   NodeButton nodeButton=new NodeButton(pcl,node,width);
4   int x=node.getHorizontalIndex();
5   int y=node.getHeight();
6   int ypos;
7   int xpos=(int) ((2*x-1)*width/(Math.pow(2, y+1)))-(80+25*
        numberOfVariables)/2;
8   if(y == 0){
9    ypos=20;
10  }
11  else{
12   ypos=y*100;
13   NodeEdge edge=new NodeEdge(node,width);
14   int edgeLocation;
15   if(node.getHorizontalIndex()%2==1) {
16    edgeLocation=xpos+(80 + (25*numberOfVariables))/2;
17   }
18   else {
19    edgeLocation=xpos+(80 + (25*numberOfVariables))/2-edge.getBreadth
          ();
20   }
21   edge.setLocation(edgeLocation,ypos-50);
22   panel.add(edge);
23  }
24  nodeButton.setLocation(xpos, ypos);
25  setScrollPaneView(xpos,ypos);
26  panel.add(nodeButton);
27  panel.repaint();
28 }
```

För att hela trädet skulle få plats gällde det även att panelen som alla noder skulle adderas till var stor nog. I många fall blir träden väldigt breda efter bara några få steg i algoritmen och därför behövs en rutin som undersöker platsutrymmet i både sid- och höjdled:

```
1  public void addNodeButton(BranchAndBoundNodeInterface node) {
2   if(node.getHeight()>currentDepth) {
3    currentDepth=node.getHeight();
4    //Checks if the size of the scrollpane should be changed.
5    if((Math.pow(2, currentDepth))*(82 + (25*node.getNumberOfVariables
         ())) > width ) {
6     width=width*2;
7     createScrollPane();
8     sizeChanged();
9    }
10   if(20 + currentDepth*102> height) {
11    height=height+105;
12    createScrollPane();
13    sizeChanged();
14   }
15  }
```

```
16    setNodeButtonPosition(node);
17    pcs.firePropertyChange("nodeAdded", node, null);
18    listOfNodes.add(node);
19
20  }
```

Med dessa två rutiner samt nod-klassen kunde vi således rita upp trädet som representerar Branch-and-Bound-algoritmens steg på ett sätt som uppfyller våra krav.

B.2 Översiktskarta

För att skapa möjligheten att se en översiktskarta över hela trädet som ritas upp i trädfönstret krävdes en del kreativitet. Det fanns ingen färdig funktion i Javas API [17] som löste problemet och det saknades även färdiga paket för ändamålet, men via en klass, `ScreenImage`[1], var det möjligt att skapa en bild av en `JComponent`[17]. I vårt fall var det klassen `"treePainter"`, en instans av `JComponent`. Genom att först spara och skala ner storleken på en bild av typen `BufferedImage` kunde vi därefter spara om den till en ny `BufferedImage` för att sedan rita upp den i översiktsfönstret.

Skalningen av bilden var inte helt rättfram den heller. För att översiktskartan alltid ska ha samma mått vid varje uppdatering av trädfönstret behövs en funktion som håller koll på om trädfönstret skalas om både i höjdled och i sidled. När så sker skalas den mindre översiktskartan mer i höjdled respektive i sidled. Nedan följer koden för skalningen av bilden:

```
1   /**
2    * A method that scales the image for the minimap.
3    * @param img
4    * @return a downscaled version for the minimap.
5    */
6   public BufferedImage scale(BufferedImage img){
7    int newHeight;
8    int newWidth;
9    if(img.getHeight()>prevHeight){
10    currentHeightScale=currentHeightScale*0.5;
11    newHeight = new Double(img.getHeight() * currentHeightScale).
          intValue();
12   }else{
13    newHeight = new Double(img.getHeight() * 0.4).intValue();
14   }
15   if(img.getWidth()>prevWidth){
16    currentWidthScale=currentWidthScale*0.9;
17    newWidth = new Double(img.getWidth() * currentWidthScale*0.9).
          intValue();
18   }else{
19    newWidth = new Double(img.getHeight() * 0.4).intValue();
20   }
21   if(resized!=null){
22    resized.flush();
23   }
24   resized = new BufferedImage(newWidth, newHeight, img.getType());
25   Graphics2D g = resized.createGraphics();
26   g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
27       RenderingHints.VALUE_INTERPOLATION_BILINEAR);
28   g.drawImage(img, 0, 0, newWidth, newHeight, 0, 0,
29       img.getWidth(), img.getHeight(), null);
30   g.dispose();
31   return resized;
32
33  }
```

I och med ovanstående lösning uppdagades ett nytt problem. Eftersom trädet i vår uppritare blir bredare och bredare ju fler noder som läggs till och på grund av att noderna

inte ska krocka med varandra, resulterade det i att den bild som skapades av `ScreenImage` och som sparades som en `BufferedImage` tog upp mycket minne. Det är ett problem då Java-applikationer bara tilldelas en viss mängd minne som kan utnyttjas. Med bilder som är uppemot 10 000 pixlar i bredd försvinner minnesutrymmet snabbt och det är inte bara de stora bilderna som orsakar dessa minnesproblem. Vid varje uppdatering av trädfönstret måste översiktskartan uppdateras, vilket leder till att många `BufferedImage`-objekt skapas. Dessa bilder ligger sedan kvar i den stack som applikationen har i sin minnestilldeling. Även om man försöker tömma minnet manuellt och ta bort gamla bilder så är det bara referenserna till bilderna som försvinner. Alltså kommer minnet att sakta men säkert ätas upp av applikationen vilket kommer att resultera i ett `OutOfMemory`-exception. När detta skrevs fanns ingen bra lösning på problemet och var en sedan tidigare känd bugg i Java, istället begränsade vi storleken på problem som kan lösas med vår applikation, något som inte kommer att påverka användarvänligheten eftersom större träd inte tillför så mycket ur en pedagogisk synvinkel.

En positiv följd av att skapa översiktsbilden med hjälp av `BufferedImage` är att vid varje uppdatering av bilden får vi även med vilken nod som är markerad, den mörkblå, vilket gör det enklare för användaren att navigera i stora träd.

B.3 Nodspecifik information - graf för heltalsoptimeringsproblem

För att kunna illustera de områden som optimeras över i varje iteration av Branch-and-Bound-algoritmen behövs ett plotverktyg för Java. Valet föll på det färdiga paketet GRAL [20]. GRAL är ett gratisbibliotek till Java för att rita upp grafer, diagram och tabeller. Det är registrerat under LGPL[19] v3 [21], vilket betyder att det kan användas i utbildningssyfte samt i kommersiellt syfte så länge man inte ändrar i källkoden.

Biblioteket är utformat så att man som användare kan använda paketet i Java Swing [17], verktygskittet för grafiska användargränssnitt i Java. Problemet vi stötte på med GRAL var att det inte fanns någon inbyggd metod för att rita upp ett område utifrån givna olikhetsvillkor i två dimensioner. Den funktionen ville vi ha i det färdiga programmet för att snabbt och enkelt kunna rita upp de områden som ska optimeras över i varje iteration av Branch-and-Bound-algoritmen.

Hörnpunkter i en polyeder

Det som vi kunde utnyttja i GRAL var möjligheten att mata in data för hörnpunkterna i den polyeder som skapas av de linjära olikhetsvillkoren. Om dessa punkter matas in i rätt ordning kan således GRAL dra linjer mellan punkterna och på så sätt bygga upp kanterna för det område som de linjära villkoren spänner upp. Idéen går alltså ut på att hitta de hörnpunkter i polyedern som spänns upp av villkoren och för att vi skulle kunna göra detta behövs ett till villkor, området får inte vara obegrä. Med ovanstående förutsättningar kan vi se alla olikhetsvillkor som funktioner av en variabel ($x_2 = f(x_1)$), alltså eliminerar vi alla olikhetstecken i villkoren och sätternsat likhetstecken istället. Genom att göra om alla villkor till funktioner av en variabel kan vi hitta alla skärningspunkter mellan de olika villkoren. Det sista steget blir då att lösa ett två-dimensionellt ekvationssystem av typen:

$$\begin{cases} ax_1 + bx_2 = e \\ cx_1 + dx_2 = f \\ a, b, c, d, e, f \in \mathbb{R} \end{cases} \tag{2}$$

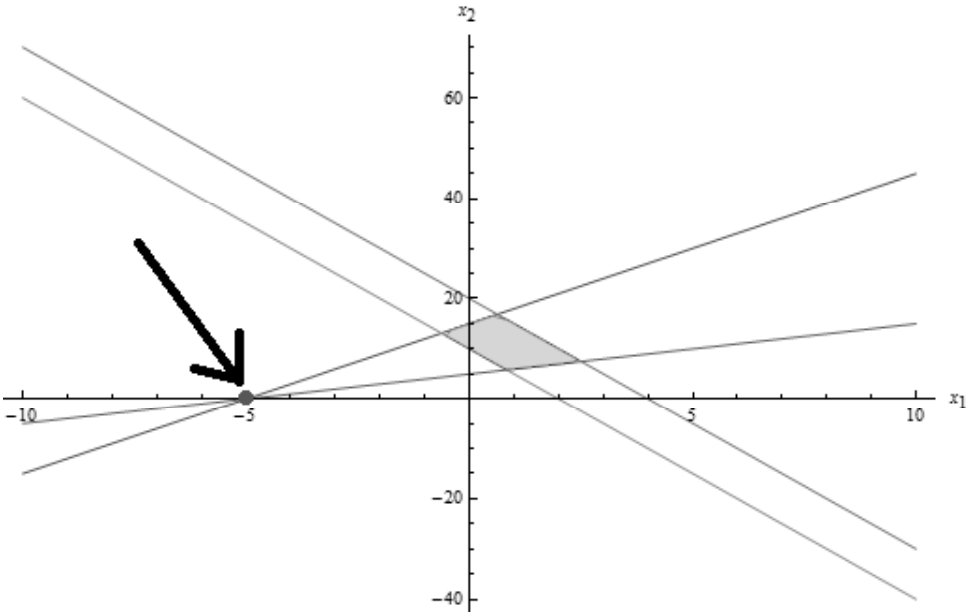
Lösningen till den här sortens system, som också är skärningspunkten, ges av Cramer's formel [22]:

$$\begin{cases} x_1 = \frac{ed-bf}{ad-bc}, & ad - bc \neq 0 \\ x_2 = \frac{af-ec}{ad-bc}, & ad - bc \neq 0 \end{cases} \tag{3}$$

Skärningspunkter

I algoritmen för att hitta hörnpunkterna jämfördes alla villkor med alla vilket gav samtliga skärningspunkter mellan villkoren. Det är dock inte säkert att alla dessa skärningspunkter är hörnpunkter i polyedern, vissa punkter kan ligga utanför området, jämför med Figur 21. Därför behövs även en rutin för att undersöka alla skärningspunkter och se om de uppfyller

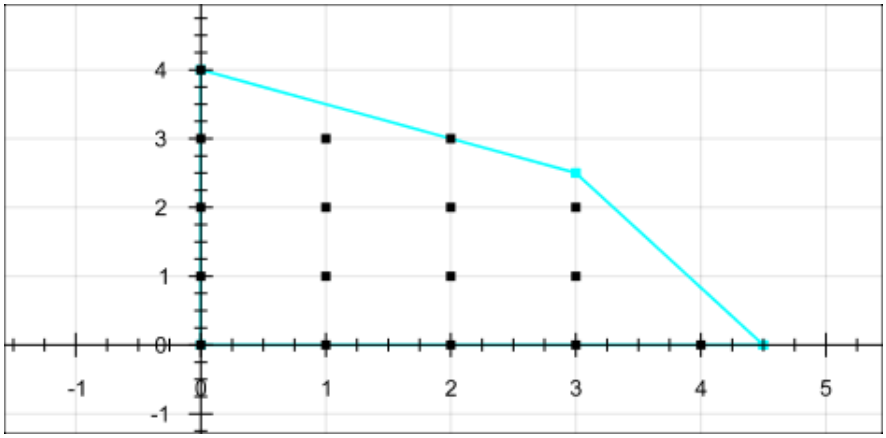
[19] *Lesser General Public License*



Figur 21: Villkor med skärningspunkt utanför område (punkten $(-5, 0)$).

de ursprungliga villkoren på området. De punkter som gick igenom denna rutin utgör således hörnpunkterna i den polyeder som spänns upp av villkoren.

För att det ska vara möjligt att dra rätt linjer mellan dessa hörn behövs en rutin som hittar alla hörnpunkter som ligger på ett av villkoren. Denna rutin bakas in i rutinen där hörnpunkterna hittas. Genom att jämföra ett villkor med alla andra och sedan upprepa för alla villkor får man hörnpunkterna i par, ett par för varje villkor. Varje par representerar två punkter som det ska dras en linje emellan. På så sätt kan man mata in dessa par i GRAL och få dem uppritade. Den resulterande plotten kommer då att visa det område som ska optimeras över. Ett exempel på uppritning av ett område kan se ut så här:



Figur 22: Exempel på uppritning av ett område.

B.4 Nodspecifik information - graf för handelsresandeproblemet

För att rita upp de bågar som ingår för respektive nods tur i vår Branch-and-Bound-algoritm för handelsresandeproblemet finns klassen `TSPGraph`. Med hjälp av klassens metoder ska en

graf över alla noder och bågar i handelsresandeproblemet ritas upp. Det svåra med uppritningen är i första steget att få ett lagom avstånd mellan varje nod. Lösningen blev att lägga alla noder, n till antalet, på en cirkel med $360/n$ grader mellan dem, se Figur 24.

Den första staden placeras en bit nedanför mittpunkten, satt som origo i beräkningarna nedan, på den `JPanel` [17] som grafen ritas upp på. Den andra noden hittas sedan genom att rotera den första $360/n$ grader moturs. Den tredjes position hittas i sin tur genom att rotera den andra med samma gradantal och så fortsätter det till dess att alla noder är utplacerade. Översatt till javakod innebär det:

```
1  private void nodePositions() {
2   d=this.getSize();
3   midpoint=new Point(d.width/2,d.height/2);
4   double angle=360/numberOfCities;
5   Point cityOne=new Point(0,(int) (d.height/2-d.height*0.2));
6   coordinates.clear();
7   coordinates.add(cityOne);
8   for(int i=1;i<numberOfCities;i++) {
9    Point node=new Point(getRotatedPoint(coordinates.get(i-1),angle));
10   coordinates.add(node);
11  }
12 }
```

Själva rotationen utförs med anropet av metoden `getRotatedPoint`, se rad 9 i koden ovan. I metoden, som tar in en punkt, från `coordinates`, och en vinkel som parametrar, roteras sedan punkten genom en multiplikation med rotationsmatrisen:

$$\begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}. \tag{4}$$

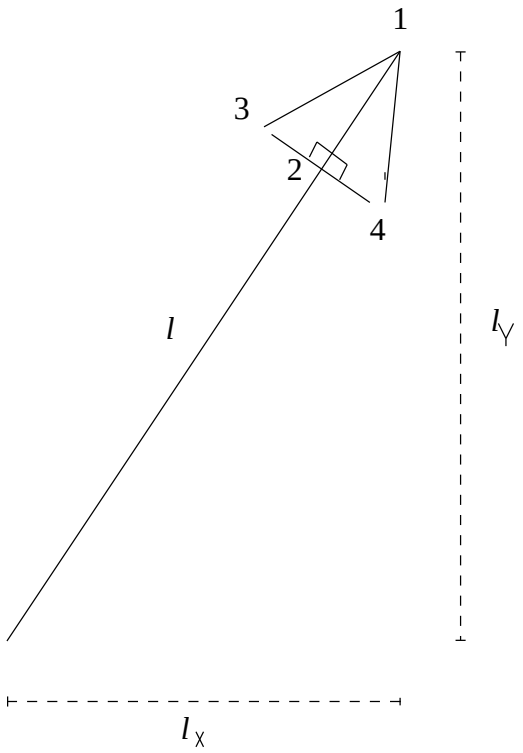
Kodmässigt ser det ut som:

```
1  private double rotationMatrix(int i, int j,double angle) {
2   angle=angle*Math.PI/180;
3   if(i==0 && j==0) {
4    return Math.cos(-angle);
5   }
6   else if(i==0 && j==1) {
7    return -Math.sin(-angle);
8   }
9   else if(i==1 && j==0) {
10   return Math.sin(-angle);
11  }
12  else {
13   return Math.cos(-angle);
14  }
15 }
```

I nästa fas ritas cirklar innehållandes nodnummer ut utifrån de ovan beräknade koordinaterna. I Java är dock inte koordinatsystemet det kartesiska utan ett vänsterorienterat system med origo i det övre vänstra hörnet av panelen. Det betyder att de framräknade punkterna måste translateras till ”javakoordinater”, vilket görs genom att addera halva panelbredden respektive panelhöjden till punkternas x- och y-koordinater.

För att rita grafens bågar dras därefter linjer mellan nodpunkterna vilket inte är speciellt krångligt. Desto svårare är det att få till så att även pilar ritas för att illustrera bågens riktning, det vill säga om bågen går mellan nod i och j eller mellan nod j och i. Det finns nämligen ingen inbyggd funktion i Javas `Graphics`-klass för detta ändamål utan pilspetsarna måste ritas upp som egna linjer.

Att hitta ett enkelt sätt att beräkna koordinaterna för dessa linjers start- och slutpunkter krävde en del tankeverksamhet från vår sida. Ett problem är att pilspetsen inte kan ritas ut i änden på den linje som binder samman två noder utan spetsen ska ritas ut en bit in på linjen, se Figur 24, eftersom nodcirklarna täcker delar av bågarna.



Figur 23: Beteckningar för att rita ut pilar i grafen för handelsresandeproblemet.

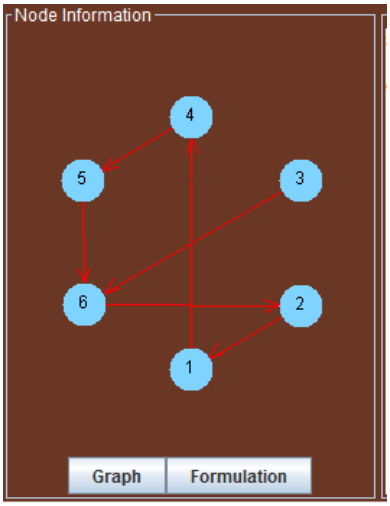
Vår lösning för att hitta punkten för pilspetsen, punkt 1 i Figur 23, går ut på att först beräkna kvoterna l_x/l och l_y/l, med beteckningar tagna från Figur 23. Dessa värden multipliceras sedan med (båglängden $-$ 15) och läggs till koordinaterna för bågens startpunkt:

```
1  double length=Math.sqrt(Math.pow(xEnd-xStart, 2)+Math.pow(yEnd-yStart
       ,2));
2  int xDiff=Math.abs(xEnd-xStart);
3  int yDiff=Math.abs(yEnd-yStart);
4  double arcsin=xDiff/length;
5  double arccos=yDiff/length;
6  int xPos, yPos;
7  int xPos1, yPos1;
8  int xPos2, yPos2;
9
10 if(xEnd>xStart) {
11  xPos=(int) (xStart+arcsin*(length-15));
12  xPos1=(int) (xStart+arcsin*(length-25));
13 }
14 else if(xEnd<xStart) {
15  xPos=(int) (xStart-arcsin*(length-15));
16  xPos1=(int) (xStart-arcsin*(length-25));
17 }
18 else {
19  xPos=xStart;
20  xPos1=xStart;
21 }
22
23 if(yEnd>yStart) {
24  yPos=(int) (yStart+arccos*(length-15));
25  yPos1=(int) (yStart+arccos*(length-25));
26 }
27 else if(yEnd<yStart) {
```

```
28    yPos=(int) (yStart-arccos*(length-15));
29    yPos1=(int) (yStart-arccos*(length-25));
30   }
31   else {
32    yPos=yStart;
33    yPos1=yStart;
34   }
```

På liknande sätt som för pilspetsen, med koordinater (`xPos`, `yPos`), fås också med koden ovan koordinaterna för punkt 2 i Figur 23. Dessa koordinater, (`xPos1`, `yPos1`),används sedan för att hitta punkt 3 och 4 i figuren med hjälp av lite linjär algebra. Grundidéen där är att hitta en vektor som är ortogonal mot vektorn från punkt 1 till 2 i Figur 23. Vektorn normeras för att därefter multipliceras med tre och läggas till respektive dras ifrån punkt 2:

```
1   ArrayList<Double> orthogonalVector=new ArrayList<Double>(2);
2   double length2=Math.sqrt(Math.pow(xPos1-xPos,2)+Math.pow(yPos1-yPos
        ,2));
3   orthogonalVector.add((yPos-yPos1)/length2);
4   orthogonalVector.add((xPos1-xPos)/length2);
5   xPos2=(int) (xPos1+3*orthogonalVector.get(0));
6   yPos2=(int) (yPos1+3*orthogonalVector.get(1));
7   g.drawLine(xPos+d.width/2, yPos+d.height/2, xPos2+d.width/2, yPos2+d.
        height/2);
8   xPos2=(int) (xPos1-5*orthogonalVector.get(0));
9   yPos2=(int) (yPos1-5*orthogonalVector.get(1));
10  g.drawLine(xPos+d.width/2, yPos+d.height/2, xPos2+d.width/2, yPos2+d.
        height/2);
```



Figur 24: Graf för handelsresandeproblemet.

Sist ritas också pilspetslinjerna ut genom anropet av `g.drawLine`, rad 7 och 10 ovan.

C Att skapa en webbapplikation av vår programvara

Ett av målen med vår programvara *Branch-and-Bound Illustrator* var att den skulle kunna användas i kurser i optimeringslära vid Chalmers tekniska högskola och Göteborgs universitet. För att förenkla åtkomsten av programvaran för studenter vid dessa skolor valde vi att lägga upp det som en Java-applikation [17] på kursen *Tillämpad optimerings* kurshemsida[20]. Programmet gjordes även tillgängligt som en nedladdningsbar jar-fil som kan köras manuellt på datorer med Java installerat. En jar-fil är ett projektarkiv innehållandes alla relevanta klasser och filer för Java-projektet. Även externa jar-arkiv som används av programmet kan bäddas in i jar-filen, vilket är en stor fördel i vårt projekt då både simplexlösaren `QSopt` [16] och plot-verktyget `GRAL` [20] är av den typen.

Skapa jar-fil med Eclipse
När vi skapade den körbara jar-filen för vårt projekt använde vi oss av den inbyggda export-funktionen i Eclipse [5]. Fördelen med att låta Eclipse sköta skapandet är att alla relevanta klasser och kopplingar samt dess struktur bibehålls automatiskt. För att det ska vara möjligt att starta jar-filen måste även Java veta vilken klass som ska köras först, vilket kan väljas vid exporteringen i Eclipse, i vårt fall är det klassen `SelectionFrameController`.

Körning av jar-fil
Den resulterande jar-filen som skapas av Eclipse är körbar på linux- och mac-datorer i terminalen med kommandot:

```
java -jar filnamn.jar,
```

förutsatt att Java[21] är installerat på datorn.

På windowsdatorer kan man högerklicka på jar-filen och välja ”öppna med” och i menyn välja Java, alternativt, om Java redan är standardprogrammet för öppning av jar-filer, räcker det med att dubbelklicka på jar-filen.

Signering av jar-fil
För att kunna lägga upp jar-filen på en hemsida och göra den körbar från webbläsaren krävs att jar-filen är signerad. Med signerad menas att upphovsmannen garanterar att programmet fungerar som det ska. Signeringen ska också fungera som en underskrift av vem som har gjort programmet. Stora företag som jobbar med programmering brukar ha en egen nyckel som kan användas vid signeringen av deras applikationer. Då vi inte är ett stort företag behövde vi skapa en egen nyckel för underskrift för programmet. Innehållet i en nyckel är oftast följande:

- Lösenord för nyckel.
- Förnamn och efternamn på upphovsmannen.
- Namn på fakultet eller institution.
- Namn på universitet/organisation/företag.
- Stad.
- Land eller provins.
- Landskod.

När vi hade skapat en egen nyckel med hjälp av verktyget `keystore` i Java kunde vi signera vår jar-fil med följande kommando:

```
jarsigner -keystore nyckelFil -storepass lösenord filnamn.jar
              aliasNamnFörNyckel
```

[20] `http://www.math.chalmers.se/Math/Grundutb/CTH/mve165/1112/`
[21] Krävs minst version JDK-1.6.

Signeringen gäller sedan i sex månader.

Lägga upp applikation på hemsida
När vi skulle göra vår Branch-and-Bound-applikation tillgänglig via en hemsida valde vi att utnyttja JavaWS[22] [23]. JavaWS är ett system som ger användaren möjligheten att starta Java-applikationer från en webbläsare. Ett annat sätt hade varit att göra applikationen till en Java Applet, en vanlig lösning för webbaserade Java-tjänster. JavaWS skiljer sig från Java Applets [24] i den meningen att Java-applikationen inte startar i webbläsaren, istället startas applikationen i ett fristående applikationsfönster. Fördelen med att använda JavaWS jämfört med Java Applet är att vi slipper att anpassa all programkod för att fungera som en Applet, till exempel hade vi behövt skriva om alla fönsterklasser så att de öppnas i ett och samma fönster istället för i flera.

För att kunna starta en Java-applikation med hjälp av JavaWS krävs tre delar:

1. **Körbar och signerad jar-fil**

2. **JNLP-fil**
   En JNLP-fil[23] [25] är en fil som beskriver hur en JavaWS-applikation ska startas i en webbläsare. Till exempel bestäms här vilken klass som ska vara startklassen för applikationen. Vår JNLP-fil ser ut som följer:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="Absolute path to resources" href="startBandB.jnlp">
 <information>
  <title>Branch-and-Bound</title>
  <vendor>Kandidatgruppen</vendor>
  <description>A Branch-and-Bound Illustrator</description>
 </information>
 <security>
  <all-permissions/>
 </security>
 <resources>
  <j2se version="1.6+" />
  <jar href="BandBWithTSP.jar" />
 </resources>
 <application-desc main-class="graphics.SelectionFrameController" />
</jnlp>
```

3. **HTML-kod för JavaWS-applikationer**
   För att applikationen ska starta i webbläsaren krävs det även ett skript i den HTML-fil som applikationen ska ligga i. Koden för skriptet läggs in som vilken annan HTML-tagg som helst och innehåller länken till ovanstående JNLP-fil. Ett exempel på hur ett sådant skript kan se ut finns nedan:

```
<body>
...
<script src="http://www.java.com/js/deployJava.js"></script>
    <script>
      // using JavaScript to get location of JNLP file relative to HTML page
      var dir = location.href.substring(0, location.href.lastIndexOf('/')+1);
      var url = dir + "startBandB.jnlp";
      deployJava.createWebStartLaunchButton(url, '1.6.0');
    </script>
    <noscript>
      <a href="startBandB.jnlp">Launch</a>
    </noscript>
...
</body>
```

[22] Java Web Start
[23] Java Network Launching Protocol

När alla dessa delar är skapta och korrekta kan applikationen startas via en webbläsare. Det krävs dock att JavaWS är installerat på den dator som användaren sitter vid, vilket är ofta fallet nu för tiden eftersom JavaWS installeras när man installerar Java Runtime Environment (JRE) [26].