Umeå Universitet
Examensarbete 20p
Saab Aerosystems AB

Generering och optimering av bitström på Gripens 1553B-databuss

Jens Lönn <prodigy@acc.umu.se>
TDCW-EXL, A1621

15 juni 2004

Handledare:

Saab AB: Peter Uhlin <peter.uhlin@saab.se>
Umeå Universitet: Per Lindström <perl@cs.umu.se>

Abstract

When designing and redesigning messages for a databus, one must always keep the efficiency of the bus in mind. The 1553B-bus is one of the main buses in the Swedish fighter JAS 39 Gripen, developed by Saab AB and BAE Systems. It is of a military standard which is widely used in military aircrafts, helicopters, ships, missiles etc.

This master thesis report describes the development and implementation of a software tool which is to be used for optimizing messages for a databus. The main problem was how to place data in a fashion that gave the best performance of the bus.

Because of the integer properties of the optimization problem it was formulated as a combinatorial optimization problem. For solving the resulting NP-hard problem, various methodologies were evaluated. The most suitable one, Simulated Annealing, was chosen for implementation. Simulated annealing is a relatively new strategy for solving large and complex optimization problems. It is particularly good at solving NP-hard combinatorial optimization problems.

The implemented tool optimize busmessages with a satisfactory result within a reasonable amount of time. The Simulated annealing method was also combined with a local search heuristic, which slightly improved the convergence rate.

Innehåll

1 Inledning

Affärsenhet Gripen International skapades den 3:e september 2001 och samägs av Svenska Saab AB och Brittiska BAE Systems. Företaget ansvarar för försäljning och marknadsföring av stridsflygplanet JAS 39 Gripen.

Verktyget som implementerats i detta examensarbete, är tänkt att användas vid design och omdesign av de meddelanden som skickas över en av Gripens huvuddatabussar. Examensarbetet har utförts inom affärsenhet Gripen International, vid avdelningen för Systemutveckling och verifiering på Saab Aerosystems AB.

1.1 Rapportens upplägg

Rapporten börjar med att i kapitel 2 först ge en bakgrund till problemet, för att sedan specificera problemet och förklara målsättningen med detta arbete. I kapitel 3 analyseras problemet mer ingående med avseende på krav och egenskaper hos problemet. Kapitlet om Matematisk programmering (kap. 4) inleder med en förklaring av begreppet ifråga, och optimeringslära i allmänhet, för att sedan ge en kort introduktion till vilka problemklasser som förekommer inom optimeringsläran. Orsaken till att de olika problemklasserna tas upp, är för att det leder fram till den problemklass som själva problemet i detta arbete tillhör. Det ger även en större förståelse för problemet. När väl problemet är specificerat och noga analyserat kommer en genomgång i kapitel 5-6 av möjliga lösningsmetoder. I kapitel 7 utvärderas sedan de olika lösningsmetoderna varefter den mest lämpliga av dessa metoder väljs att implementeras för detta problem. I kapitel 8 beskrivs implementeringen av verktyget och vald lösningsmetod, varefter resultatet av implementationen redovisas i kapitel 9. En diskussion och utvärdering av verktyget samt resultatet ges i kapitel 10. Slutligen föreslås i kapitel 11 vad som kan göras i framtiden för att utveckla och förbättra detta verktyg.

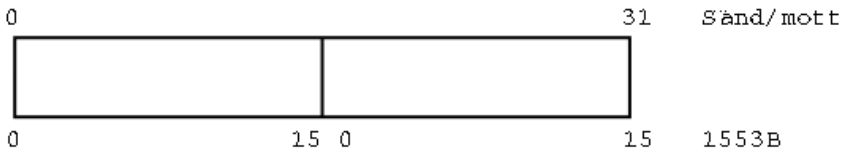
I appendix A tackas de som hjälpt mig med mitt examensarbete. En ordlista över notation och använda förkortningar finns i appendix B. Användarmanualen till verktyget finns under appendix C, sist i rapporten finns de använda referenserna.

2 Bakgrund

I detta kapitel ges en bakgrund till databussen och själva problemet. Sedan specificeras problemet samt målsättningen med arbetet.

Vid design av bussmeddelanden som ska skickas över Gripens 1553B-databuss, måste hänsyn tas till hur datatyper i bussmeddelanden är placerade i förhållande till varandra. Att på förhand optimera meddelandestrukturen för specifika meddelanden är ett sätt att höja prestandan på bussen. Orsaken till att denna optimering överhuvudtaget behövs är att bussen är en relativt långsam buss.

För att överföringen av data ska ske på ett så effektivt sätt som möjligt bör ett antal regler beaktas vid designen. Exempel på två viktigare regler är att ingen datatyp bör hamna över någon ordgräns. På 1553B-bussen räknas ett ord som 16 bitar, men på mottagar/sändarsidan av bussen räknas ett ord som 32 bitar, se *figur 1*.



Figur 1: Skillnad mellan bussens och sändar-/mottagarsidans syn på ordlängd.

Utplacering av data över dessa ordgränser bör undvikas för att bussens prestanda ska vara så optimal som möjligt. De regler som används vid designen av bussmeddelanden redovisas i kapitel 3.2.

2.1 Standarden MIL-STD-1553B

MIL-STD-1553 är en militär standard som definierar egenskaper hos en databuss. Standarden konstruerades så tidigt som 1973, och den användes först i stridsflygplanet F-16 och attackhelikoptern Apache. Den senaste revisionen av standarden, MIL-STD-1553B släpptes 1978 och är den som gäller idag. Dock har ett antal modifieringar och förbättringar av standarden skett under årens lopp. Databussen används främst inom militära tillämpningar som stridsflyg, helikoptrar, missiler, båtar och stridsvagnar. Civilt används den i t.ex. satelliter och även på den internationella rymdstationen. Standarden är accepterad av NATO och ett flertal utländska regeringar [12].

I tabell 1 sammanfattas de viktigaste egenskaperna hos MIL-STD-1553B. Notera att

Överföringshastighet	1 MHz
Ordlängd	20 bitar
Databitar / ord	16 bitar
Meddelandelängd	Max 32 ord
Överföringsteknik	Halv duplex
Operation	Asynkron
Feltolerans	dubbelt redundant

Tabell 1: Några av de viktigare egenskaperna hos bussen.

med beteckningar som databitar och dataord i tabellen, menas de bitar per ord som innehåller det data som ska skickas över bussen. Resterande bitar är kontrollinformation som bussen lägger till. Till exempel är ett ord på bussen egentligen 20 bitar långt, men bara 16 bitar data får plats i ordet. resterande 4 bitar består av en paritetsbit och 3 bitar för synkning. Synkningen ger stabilitet hos överföringen.

2.2 Problemställning och målsättning

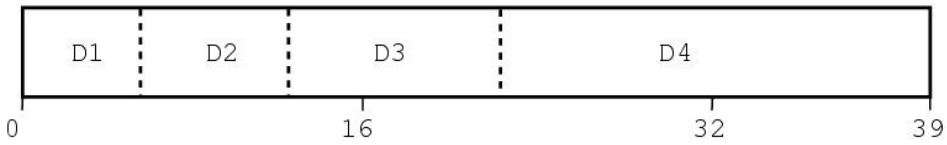
Huvudproblemet bestod av att designa och implementera ett mjukvaruverktyg som ska användas vid designen av de bussmeddelanden som skickas över Gripens databussar. Detta krävde att ett optimeringsproblem, dess krav och begränsningar identifierades och modellerades. Dessutom behövdes en teoretisk utredning angående möjliga lösningsmetoder för optimeringsproblemet, samt en utvärdering av vilken metod som var mest lämplig att implementera.

Verktyget måste sköta placeringen av datatyper i meddelanden enligt vissa givna regler. Reglerna är av olika hårdhetsgrad och påverkar prestandan olika mycket beroende på vilka som bryts. Alla regler kan inte följas strikt, utan en avvägning måste kunna göras angående vilka regler som är bäst att bryta beroende på situation. Verktyget kommer att vara till nytta vid designen av nya meddelanden såväl som då designen av befintliga meddelanden ska ändras eller kontrolleras.

Orsaken till att ett verktyg behövs är bland annat att det för närvarande inte finns något liknande verktyg, som kan hantera optimeringen av bussmeddelanden deklarerade i pascal/d80 som används för meddelandena. Alternativet har varit att hantera all denna optimering för hand vilket är ett väldigt tidsödande arbete. En generell uppdatering av busstrafiken tar i snitt i anspråk ungefär 2250-3000 arbetstimmar varje gång den utföres. Vanligen uppdateras busstrafiken c.a. 2 gånger på 3 år. Andra orsaker till att verktyget behövs är t.ex. att all optimering bör hanteras på ett konsistent sätt. Ett

verktyg minskar även ett annars intensivt och tidspressat arbetsmoment. Troligtvis kan även bättre lösningar hittas än vad som är möjligt då optimeringen sker för hand. Detta medför att bussens prestanda kommer att kunna ökas.

För att ge en bild av vad verktyget kommer att göra kan figur 2 betraktas. Denna illustrerar ett bussmeddelande som är 39 bitar långt och består av 4 dataelement, D1-D4.



Figur 2: Exempel på representationen av ett bussmeddelande.

Element D4 är ett exempel på en situation där en regel av stark hårdhetsgrad bryts. Inget dataelement får ligga över en 32 bitars ordgräns om det är möjligt att undvika det. En regel av svagare hårdhetsgrad är att ingen 16 bitars ordgräns får brytas, vilket t.ex. D3 bryter emot. Varje regelbrott kommer innebära en kostnad i form av sämre utnyttjande av bussens prestanda. Verktyget kommer att hantera omflyttning av element för att minimera den totala kostnaden av de regler som bryts. Fallet i figur 2 hade verktyget hanterat genom att flytta om datatyperna för att undvika att D3 och D4 bryter några utplaceringsregler. Om en situation uppstår där minst en av reglerna måste brytas, kommer utplaceringen ske på det sätt som ger minimal negativ inverkan på bussens effektivitet. T.ex. kommer en 16 bitars ordgräns mycket hellre att brytas än en ordgräns på 32 bitar.

Det övergripande målet med detta examensarbete är att konstruera ett verktyg, som placerar ut datatyperna i bussmeddelanden på ett optimalt sätt och inom rimliga tidsgränser. Verktyget ska sedan kunna vara till hjälp vid design och omdesign av bussmeddelanden.

3 Problemanalys

Först förklaras strukturen hos, samt använd notation för, meddelanden som skickas över bussen. Sedan redovisas kraven på verktyget, varefter en optimeringsmodell konstrueras med avseende på dessa. En analys av optimeringsproblemets komplexitet kommer även utföras.

Innan en lösning till problemet kan konstrueras måste först en bakgrund ges angående hur bussmeddelanden kan se ut. Det som komplicerar optimeringsförfarandet är bland annat att bussmeddelanden kan innehålla substrukturer med datatyper inuti som i sin tur måste optimeras. Antalet typer i ett meddelande kan dessutom vara stort, över 100 st.

3.1 Struktur hos generellt bussmeddelande

Data skickas alltid över bussen i den ordning det har deklarerats i bussmeddelandet om inget annat anges. Med datatyp menas alla konstruktioner som inte kan delas upp i mindre datatyper. Ett enkelt bussmeddelande innehållandes en enda datatyp har följande struktur:

```
BUSSMEDDELANDE_TYP
                =packed record
        variabel     : Typnamn;
    endrecord;
```

Ett bussmeddelande deklareras som ett record. Det består av en header med meddelandets namn, sedan deklareras meddelandets innehåll. Meddelandet avslutas alltid med ett endrecord. De konstruktioner som kan ingå i ett bussmeddelande är:

Enkel datatyp/variabel:

```
var1 : Typ_X;
```

En enkel datatyp är en vanlig variabel och de deklareras antingen på detta sätt eller som flera på en rad separerade av kommatecken. Variabelnamnet är var1 och Typ_X är vilken typ av variabel det handlar om. Ibland behöver reservbitar läggas till i bussmeddelandet för att fylla upp till en jämn multipel av ord eller av andra orsaker. Dessa reserver deklareras alltid som enkla variabler och kan vara av vilken typ som helst,

bara de har den storlek som krävs. Reservbitar har alltid namn på formen `Spare` eller `Reserv` och numreras för att namnen ska skilja sig åt i recordet.

Array:

```
array_variabel : array  [Typ_X]  of  Typ_Y;
```

Antalet element i en array anges av värdet på det som finns inom klamrarna []. Storleken på varje element ges av den sista typen på raden.

Uppräkningsbar typ:

```
Enum_variabel  : (Elem1, Elem2, Elem3, Elem4);
```

Den uppräkningsbara typens namn anges först och innanför parenteserna listas de element som ingår i typen. Antalet element i en uppräkningsbar typ bör vara ett jämnt antal element.

Case:

```
case Case_namn : Typ_Av_Case of
        :val_1:
        (Reserv : Typ_X;
         var1 : Typ_Y;);
        :val_2:
        (var2 : Typ_Y;);
endcase;
```

Ett case består av ett eller flera val där det som hör till respektive val omsluts av parenteser. Själva valet identifieras av det som finns emellan två kolon inuti caset.

Typen på caset anges efter casets namn på första raden (i detta fall `Typ_Av_Case`). När bussmeddelandet sänds kommer endast ett av valen i caset att skickas, och värdet på `Typ_Av_Case` används för att ange vilket av valen i caset som skickas. Typen på själva caset är i sig en datatyp som tar plats i meddelandet, och denna datatyp lagras alltid först i caset. Vid sändning av bussmeddelande som innehåller ett case behöver dock ej denna typ alltid skickas med, det kan skilja sig från gång till gång. De olika valen i ett case behöver inte ha samma storlek, men ska ha det efter optimeringen. Det innebär att reservbitar kan komma att läggas till i vissa av valen.

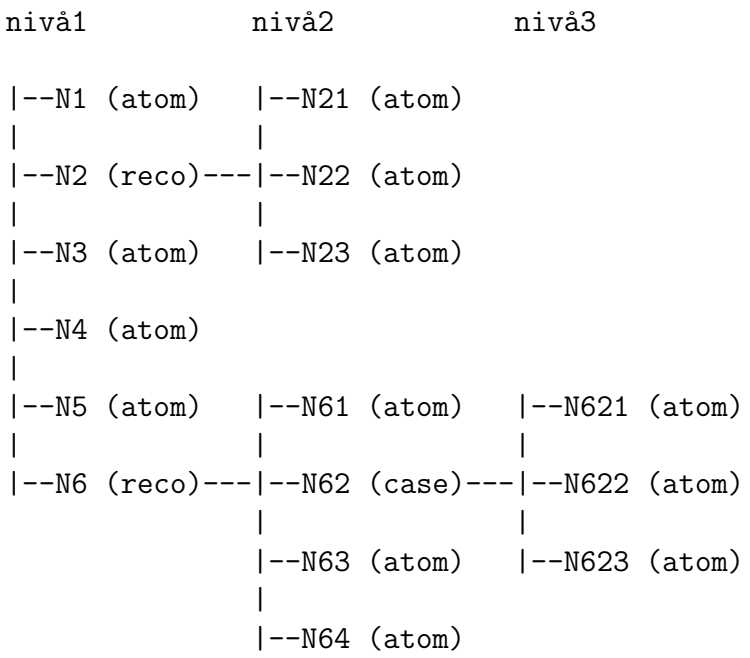
Record:

```
RECORD_NAMN : packed record
        vanlig_variabel     : Typnamn;
    endrecord;
```

Även ett record kan ingå i ett bussmeddelande vilket innebär att records får vara nästlade. Notera skillnaden i deklarationen av det record som utgör bussmeddelandet (namn på record och '=') och de records som ingår i bussmeddelandet (namn på record och ':'), dessa kommer kallas subrecords i denna rapport. Records och cases kommer gemensamt benämnas som substrukturer i bussmeddelanden. Dessa konstruktioner kan finnas i princip vart som helst inuti ett bussmeddelande, vilket kan ge relativt komplexa meddelanden. Ett exempel på ett mer komplext bussmeddelande ges här nedan:

```
MER_KOMPLEXT_MEDDELANDE_Typ
               =packed record
          record1 : packed record
               var1 : typA;
               var2 : typB;
               var3 : typA;
               var4 : typD;
               var5 : typC;
               enum1 : (E1, E2, E3, E4, E5, E6, E7, E8);
         endrecord;
          var6 : typD;
          var7 : typE;
          var8 : typD;
          var9 : typA;
          record2 : packed record
              case case1 : Boolean_type of
             :Sann:
             (record3 : packed record
                   var10 : typC;
                   var11 : typC;
               endrecord;);
             :Falsk:
             (record4 : packed record
                   var12 : typA;
                   var13 : typB;
               endrecord;
          Array1 : array [ Array1_Typ ] of  boolean;);
             endcase;
         endrecord;
          var14 : typX;
      endrecord;
```

För att enklare visualisera hur bussmeddelanden ser ut kan de betraktas enligt följande. Om datatyperna representerar basfallen (eftersom de inte kan delas upp mer) så kan de cases och records som finns inuti ett meddelande ses som en trädstruktur enligt exemplet i figur 3.



Figur 3: Grafisk representation av ett meddelandes trädstruktur.

Det som ligger på nivå 1 utgör själva meddelandet, denna nivå benämns rot-nivån i denna rapport. De typer av noder, d.v.s. delträd, som finns i trädet är records (`reco`) och cases (`case`). Löven benämns `atom` och dessa består av datatyper som inte innehåller andra datatyper, d.v.s. de är atomära datatyper.

Med bilden av denna trädstruktur blir det lättare att inse problemets struktur, och för att förstå hur problemet hanteras vid optimering av nästlade meddelanden. Det är på denna form som bussmeddelanden kommer lagras i verktyget. Meddelanden som innehåller substrukturer kommer benämnas `nästlade` meddelanden, medan de som inte inte innehåller substrukturer kommer benämnas som `enkla` meddelanden.

För varje bussmeddelande måste samtliga storlekar för de typer som används vara kända. Hur storleken på de olika typerna kan tas fram visas i tabell 2. De typer som ej omfattas av denna tabell måste slås upp via en mappningsfil som mappar typnamn till dess storlek.

Datatyp	**Antal bitar**
boolean	1 bit
IntX	X bitar
Int0_n	$\frac{log(n)}{log(2)}$ bitar (n = antal element)
char	8 bitar
Uppräkningsbar	$\frac{log(n)}{log(2)}$ bitar (n = antal element)
array [X] av boolean	X bitar
array [X] av char	X*8 bitar
array [X] av int	X*8 bitar
array [X] av Y	Om $Y < 33$ bitar: $X * 32$ bitar Om $32 < Y < 65$ bitar: $X * 64$ bitar osv

Tabell 2: Storlekar på packade datatyper.

3.2 Krav

Följande krav är de ursprungliga kraven på verktyget som extraherats fram ur beskrivningen av uppgiften, samt genom diskussioner med extern handledare på Saab. De flesta kraven har med utplacering av datatyper att göra, men inte alla. För varje krav presenteras en kort beskrivning av hur det hanteras. Krav som kan formuleras som optimeringsregler kan antingen vara matematiska eller tumregelsbaserade (heuristiker). De krav som varken kan formuleras som matematiska eller tumregelbaserade optimeringsregler måste på annat sätt beaktas vid utvecklingen av verktyget. Kraven på verktyget är följande:

1. Aldrig gå över 32 bitars gräns. Gränsen får brytas om typen är större än 32 bitar.
   - Formuleras som matematisk optimeringsregel.
2. Undvik att gå över gränsen för 16 bitars ord om det är möjligt. Detta krav gäller endast de 16 bitars ordgränser som ej är en 32 bitars gräns.
   - Formuleras som matematisk optimeringsregel.
3. Om en typ är mellan $16 < X < 32$ stor, måste en 16 bitars gräns korsas. Se då till att ej gå över en gräns som även är en 32 bitars ordgräns. Gå mycket hellre över 16 bitars gränsen.
   - Formuleras som matematisk optimeringsregel.
4. Förutsatt att en typ är större än ett ord är det alltid bättre att lägga den så att den slutar på en ordgräns, än att börja på en ordgräns.

- Formuleras som matematisk optimeringsregel.

5. Skjut in reservbitar (spares) där det behövs.
   - Hanteras genom att enbart fylla upp till jämn ordgräns i meddelandet. Användaren får sedan välja om fler reservbitar ska användas eller ej.

6. Hålla antal reservbitar till ett minimum.
   - Hanteras genom att enbart fylla upp till jämn ordgräns i meddelandet.

7. Se till att placera reservbitarna inne i ord, helst i början av ord. Eftersom en "right shift" kommer behöva utföras för att läsa innehållet i variabeln närmast innan reservbiten. Undvik i största mån att placera reservbitar i slutet av ord.
   - Hantera med heuristik som vid slutet av optimeringen (då all annan optimering är färdig) går igenom meddelandet och försöker flytta alla reservbitar till slutet av de ord de ligger i.

8. Typer för uppdatering av räknare ska helst läggas sist i bussmeddelandet.
   - Hanteras genom att utplaceringen av räknaren ges en kostnad som är proportionell emot avståndet till slutet av meddelandet.

9. Kontrollera att alla val i casen är lika stora, om inte, fyll ut till det största valets storlek.
   - Hanteras då caset ska optimeras, valen kommer fyllas ut med reserver.

10. Kontrollera att bussmeddelandet är mindre än 32 ord, annars varna att det är bäst att dela upp i två meddelanden.
    - Hanteras innan optimeringen påbörjas. Optimering av meddelanden över 32 ord kommer inte tillåtas av verktyget.

11. Bussmeddelandet ska ha en storlek som är en multipel av sändar/mottagarsidans ordstorlek (d.v.s. 32 bitars ord).
    - Hanteras genom att lägga till fler bitar innan optimering påbörjas. De bitar som läggs till kommer markeras som reservbitar.

12. Cases och subrecords behöver inte ha en storlek som är en multipel av antalet ord.
    - Behöver ej hanteras, låter bara bli att lägga till reservbitar i subrecords/cases för att fylla upp till någon gräns.

13. Lösningen behöver ej vara CPU-snål, men dock ej heller av brute-force typ.

    - Går varken att formulera som optimeringsregler eller implementeras som en hanteringsfunktion eller liknande då det beror på problemets komplexitet, samt val av lösningsmetod. Kommer hållas i åtanke under utvecklingen av verktyget.

14. Flera reservbitar ska kunna väljas att läggas till bussmeddelandet.

    - Implementeras som ett specialfall i verktyget där användaren får välja om och hur många extra reservbitar som ska läggas till. Förnärvarande kommer endast extra reservbitar tillåtas att läggas till i enheter om 16 bitar.

15. Det ska gå att välja om reservbitar ska läggas in i nästlade meddelanden eller ej. För vissa typer av meddelanden får inte storlekarna i subrecords/cases ändras.

    - Implementera båda fallen i verktyget. Användaren får sedan innan optimeringen välja om reservbitar ska kunna läggas till i substrukturer eller ej.

3.3 Optimeringsmodellen

Modelleringen av det kombinatoriska optimeringsproblemet har skett enligt följande. Låt i representera datatypens index[1] och låt j representera dess position i sekvensen av datatyper. Då får vi att:

$$x_{ij} = \begin{cases} 1, & \text{om datatyp } i \text{ är på position } j \\ 0, & \text{annars} \end{cases}$$
$$i, j \text{ heltal}$$
$$1 \leq i, j \leq n \tag{1}$$

Där n är antalet typer i sekvensen av datatyper. Utplaceringen av varje typ i är associerad med en kostnad $C(\cdot)$ för utplaceringen i position j. Denna kostnad är nollskild om placeringen bryter mot någon av reglerna för utplacering i kap. 3.2. Objektfunktionen blir därmed en kostnadsfunktion $C(i, j)$ över alla kostnader som alla utplaceringar givit upphov till. Notera att parametrarna i, j till kostnadsfunktionen antyder att kostnaden beror på typerna och dess inbördes positioner i sekvensen. Kostnadsfunktionen som ska minimeras blir då

[1] Ett unikt identifikationsnummer som sätts för varje datatyp för att det alltid ska gå identifiera den. Kan t.ex. sättas för typerna enligt den initialkonfiguration de är i.

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n}(x_{ij} \cdot C(i,j) \tag{2}$$

Eftersom det finns flera regler att beakta kommer kostnadsfunktionen $C(i,j)$ i sin tur att bestå av ett antal delkostnader.

$$\begin{aligned} C32\_tot(i,j) &= \sum C32(i,j) \\ C16\_tot(i,j) &= \sum C16(i,j) \\ CS\_tot(i,j) &= \sum CS(i,j) \end{aligned} \tag{3}$$

Där $C32\_tot(i,j)$ är totala kostnaden att placera ut datatypen i på positionen j. Denna kostnad är nollskild såvida inte placeringen innebär att typen kommer ligga över en 32 bitars gräns. Notera att $C32\_tot(i,j)$ är en summa av alla gränser som typen i bryter. En typ kan bryta flera gränser. Kostnaden $C16\_tot(i,j)$ är kostnaden att placera ut datatypen i på position j över en 16 bitars gräns. Kostnaden $CS\_tot(i,j)$ är totala kostnaden att placera ut en typ som är en specialtyp (typ som används för uppdatering av räknare). Denna kostnad är nollskild enbart då denna typ ligger sist i sekvensen.

Summan för $C32\_tot$ bör vara noll eftersom ingen typ mindre än 32 bitar får ligga över denna gräns. En specialregel, förutsatt att typen slutar på en ordgräns, är att typer större än 32 bitar inte får en kostnad för den gräns den oundvikligen måste bryta. Att ta med den kostnaden kan annars bli missvisande och även leda till att vissa lokala minimum undviks. Slutar typen inte på ordgränsen tas dock kostnaderna för de andra gränserna också med.

Kostnaderna $C32$, $C16$ samt CS måste beräknas för varje typ, som sedan summeras till den totala kostnaden. Kostnaden $C16$ beräknas m.h.a. Algoritm 9 på sidan 54, och $C32$ beräknas m.h.a. Algoritm 10 på sidan 55. Kostnaden CS är rättfram då den bara mäter avståndet till slutet av meddelandet, och multiplicerar sedan detta med en vikt för att få fram kostnaden.

Den totala kostnaden för utplacering av en typ blir således:

$$C(i,j) = \sum_{i=1}^{n}(C32\_tot(i,j) + C16\_tot(i,j) + CS\_tot(i,j)) \tag{4}$$

För att kunna använda modellen måste omgivningen till lösningarna definieras. D.v.s. den mängd lösningar som ska gå att nå genom att utföra en enkel operation på nuvarande lösning. Omgivningen i detta problem har valts till den mängd lösningar som kan nås genom att byta plats på två av typerna i bussmeddelandet. Denna omgivning är enkel och ger en lämplig storlek på mängden lösningar. För n st typer blir storleken på omgivningen $\sum_{i=1}^{n}(n-i)$.

Förutom lämplig omgivning att hitta nya lösningskandidater ur, måste beräkning av kostnader i kostnadsfunktionen ske på ett sätt som styr lösningen mot bättre lösningar. De kostnader och vikter som används för beräkningen av kostnader i verktyget kan ses i tabell 3.

Kostnadsfaktorer/vikter	**Betydelse**
BREAK_16_BORDER_COST	Kostnad, 16-gräns om $typ < 16$
BREAK_32_BORDER_COST	Kostnad, 32-gräns om $typ < 32$
DIFF_FROM_16_BORDER_WEIGHT	Vikt, avstånd från 16-gräns
DIFF_FROM_16_BORDER_COST_OFFSET	Kostnad, 16-gräns om $typ > 16$
DIFF_FROM_32_BORDER_WEIGHT	Vikt, avstånd från 32-gräns
DIFF_FROM_32_BORDER_COST_OFFSET	Kostnad, 32-gräns om $typ > 32$
UPDC_COST_FACTOR	Kostnadsfaktor för räknartyp

Tabell 3: Kostnader och vikter för styrning av lösning mot optimalitet.

Vid beräkning av kostnader för typer som får plats inom respektive ordstorlek, kommer enbart en straffkostnad (BREAK_16_BORDER_COST resp. BREAK_32_BORDER_COST) tillämpas då en gräns bryts. Är typen större än det ord som representerar ordgränsen, kommer det avstånd som typen går över gränsen ”åt fel håll” att läggas till kostnaden. Med ”åt fel håll” menas hur långt änden på typen sträcker sig över gränsen[2]. Kostnaden att ej hamna direkt emot gränsen blir antalet bitar typen går över gränsen viktat med DIFF_FROM_16_BORDER_WEIGHT resp. DIFF_FROM_32_BORDER_WEIGHT. Denna vikt kommer styra lösningen mot att det ska vara gynnsamt desto närmare gränsen den ligger. Utöver detta används en s.k. offset-kostnad (DIFF_FROM_16_BORDER_COST_OFFSET resp. DIFF_FROM_32_BORDER_COST_OFFSET) som kompenserar för att typerna annars gärna hamnar någon enstaka bit över gränsen. Eftersom det fallet annars ger en relativt låg kostnad.

I samtliga fall bör kostnaderna för 32 bitars gränsen vara mycket högre än kostnaderna för 16 bitars gränsen. Kostnaden för utplacering av specialtypen för uppdatering

[2]Detta kommer av kravet ”Förutsatt att en typ är större än ett ord är det alltid bättre att lägga den så att den slutar på en ordgräns än att den får börja på en ordgräns.”.

av räknare, viktas som antalet bitar ifrån slutet av bussmeddelandet viktad med faktorn UPDC_COST_FACTOR. För optimering av reserver används inga kostnader utan enbart heuristiska regler eftersom dessa kostnader i vissa fall annars kan krocka med motiven för de viktigare reglerna, d.v.s. reglerna angående gränser.

Val av lämpliga värden på de kostnader och vikter i tabell 3 som användes vid implementeringen anges under implementationsdelen, kapitel 8, i tabell 6. Där finns även algoritmbeskrivningar över beräkning av kostnaderna.

3.4 Problemets komplexitet

Problemet karakteriseras av att det har följande egenskaper:

1. En mängd av n st oberoende datatyper ska utplaceras.
2. Utplaceringen ska skötas sekvensiellt av en enda dator.
3. Då utplacering av en datatyp påbörjats, ska den utföras tills den är klar. Utan att bli avbruten för att göra något annat på vägen.

Detta ger en 1-1 (ett-till-ett) korrespondens [19] mellan sekvensen av n datatyper och en permutation av datatypernas index $1, 2, ..., n$. Det totala antalet lösningar till problemet blir då $n!$, vilket är totala antalet möjliga permutationer av de n elementen. Beräkningskomplexiteten är således $O(n!)$ vilket innebär att antalet möjliga lösningar är väldigt stort. Det är en praktisk omöjlighet att studera alla dessa lösningar för att hitta den bästa. En jämförelse av hur snabbt antalet lösningar växer med andra funktioner av n kan ses i tabell 4.

n	n^2	e^n	$n!$
5	25	148	120
10	100	22026	3628800
50	2500	$5.18 \cdot 10^{21}$	$3.04 \cdot 10^{64}$
100	10000	$2.68 \cdot 10^{30}$	$9.33 \cdot 10^{157}$

Tabell 4: Hur snabbt olika funktioner av n växer jämfört med $n!$.

Som exempel kan ett bussmeddelande bestående av 20 typer betraktas, detta ger $2.43 \cdot 10^{18}$ lösningar. Om en snabb dator hinner gå igenom t.ex. 10^{12} lösningar per sekund så tar det c.a. 28 dygn att gå igenom alla. Ökar antalet till $n = 30$ tar det runt 8.41 miljarder år att gå igenom samtliga.

Detta problem är ett NP komplett problem. Förutom beräkningskomplexiteten finns det ett antal ytterligare komplicerande faktorer, dessa är:

1. Modellen är olinjär p.g.a. att kostnaderna som beräknas är olinjära funktioner.
2. Vissa parametrar i modellen får bara anta diskreta värden, vilket i och för sig minskar antalet lösningar m.a.p. det kontinuerliga fallet. Men å andra sidan går inte de teorier för kontinuerliga modeller att använda.
3. Optimeringsalgoritmen måste köras för varje subrecord och subcase som finns i meddelandet. Tyvärr så går det inte att bara optimera varje subrecord/subcase, eftersom det på förhand inte går att veta vart 16/32 bitars gränserna kommer att gå någonstans. Orsaken till detta är att det på förhand inte går att veta vart starten på substrukturen kommer att hamna i meddelandet innan den yttre strukturen optimerats. Det går inte att anta att varje substruktur kan läggas emot starten av en gräns.

4 Matematisk programmering

I detta kapitel ges en introduktion till optimering som vetenskap samt en klassificering av optimeringsproblem. Här läggs grunderna till de kunskaper som behövs för att förstå den problemklass optimeringsproblemet som behandlas tillhör. Slutligen ges en översikt av den indelning av lösningsmetoder som används i denna rapport.

Matematisk programmering[3] (MP) innebär en matematisk planeringsprocess där matematiska modeller och metoder används för att finna bästa handlingsalternativ i olika beslutssituationer. Begreppet matematisk programmering är ett samlingsnamn för optimeringsområdet och i det ligger underförstått att lösningen skall erhållas på ett algoritmiskt sätt. MP innebär med andra ord studie av metoder för att lösa klasser av optimeringsproblem.

Optimering innebär maximering eller minimering av en s.k. *målfunktion* $f(\mathbf{x})$ som beror av en eller flera *beslutsvariabler* $\mathbf{x}$. Beslutsvariablerna $\mathbf{x} = (x_1, ...x_n)^T$ är en ändligt-dimensionell vektor som representerar beslut som ska göras i optimeringsmodellen. Optimeringen sker oftast med avseende på *bivillkor* som begränsar värdena på beslutsvariablerna att höra till en mängd X av tillåtna lösningar, den s.k. *tillåtna mängden* eller *lösningsrummet*. Optimeringen av x innebär alltså att hitta den lösning $\mathbf{x} \in X$ som ger det optimala värdet hos objektfunktionen. Ett optimeringsproblem (antar minproblem) kan generellt formuleras som:

$$\begin{array}{ll} \text{minimera} & f(\mathbf{x}) \\ \text{med avseende på} & \mathbf{x} \in X \end{array}$$

Den tillåtna mängden X kan representeras av ändligt många bivillkor i form av olikheter. Optimeringsproblemet ovan kan alternativt formuleras som:

$$\begin{array}{ll} \text{minimera} & f(\mathbf{x}) \\ \text{med avseende på} & g_i(\mathbf{x}) \leq b_i, \quad i = 1, ..., m \end{array} \tag{5}$$

En omgivning $N(\mathbf{x}, \sigma)$ till $\mathbf{x}$ kan definieras som består av alla lösningar som i någon mening ligger i närheten av lösningen. Omgivningen nås genom att modifiera den nuvarande lösningen med en enkel operation σ. För kombinatoriska problem är omgivningen till en lösning relativt enkel att definiera. I det problem som behandlas i denna rapport innebär operationen σ en permutation av två typer.

Beroende på egenskaper och utseende hos objektfunktionen, bivillkoren och beslutsva-

[3]Inom optimeringsläran har ordet "programmering" betydelsen "planering" (från eng. begreppet *program*). Termen uppkom innan programmering blev en förkortning av datorprogrammering.

riablerna kan optimeringsproblemet klassificeras och lösas på olika sätt. Vilket kommer beskrivas kortfattat i följande sektioner.

4.1 Linjär och ickelinjär programmering

Ett optimeringsproblem är ett linjärt programmeringsproblem (LP) om det uppfyller följande villkor [8]:

1. Alla beslutsvariabler är icke-negativa och kontinuerliga.
2. Objektfunktionen är en linjär funktion av beslutsvariablerna.
3. Bivillkoren är linjära olikheter (eller ekvationer) av beslutsvariablerna.

Denna typ av problem är relativt lätta att lösa till optimalitet eftersom en optimallösning till ett LP problem alltid återfinns i en hörnpunkt i lösningsrummet. Detta är ett resultat av faktumet att alla LP problem är konvexa[4]. Om problemet är konvext kommer varje lokalt optimum också att vara ett globalt optimum [8]. Det finns en välkänd metod, *simplexmetoden*[5], som används för att lösa LP problem som utnyttjar detta faktum. Simplexmetoden är en generell metod som löser alla LP problem till optimalitet oavsett struktur och storlek.

Ett optimeringsproblem är ett ickelinjärt problem (ILP), om minst någon av funktionerna (objektfunktionen eller något av bivillkoren) är en olinjär funktion. Till skillnad från LP problem så finns det ingen generell metod att lösa alla ickelinjära problem med. Lösningsmetoderna måste istället anpassas till den struktur som problemet har [8], d.v.s. till utseendet på de ingående funktionerna. Olika metoder har därför utvecklats för olika typer av problem.

För lösning av ILP problem är det mycket viktigt att veta huruvida problemet är konvext eller ej. Då de flesta lösningsmetoder oftast bara finner lokala optima till denna typ av problem, kommer konvexiteten hos problemet att avgöra huruvida det säkert går att uttala sig om den erhållna lösningen är ett globalt optimum eller ej.

4.2 Linjär och ickelinjär heltalsprogrammering

Ett optimeringsproblem är ett linjärt heltalsproblem (HP) om en delmängd av beslutsvariablerna (minst en) är definierade som *diskreta variabler*. Generellt sett är HP problem mycket svårare att lösa än LP problem [8]. Rent intuitivt kan det tyckas att det

[4]Konvexitetsbegreppet kommer bara att beröras ytligt här då det inte går att tillämpa på det optimeringsproblem som rapporten behandlar.

[5]En bra genomgång av simplexmetoden kan hittas i t.ex. [8] eller [18].

borde vara tvärtom eftersom ett kontinuerligt problem har ett oändligt antal tillåtna lösningar, medan ett diskret problem har en ändlig mängd tillåtna lösningar. Det som gör HP problem så svåra är att HP problem alltid är icke-konvexa [8].

Problemets icke-konvexitet är ej enda problemet eftersom samtliga punkter mellan två närliggande heltalspunkter i lösningsrummet till ett HP problem är otillåtna. Det innebär att det ej går att beräkna derivatan eller utnyttja gradientriktningar för att hitta vägen till bättre lösningar. Det är dessutom mycket svårt (och ofta omöjligt) att avgöra huruvida en tillåten lösning som är ett lokalt optimum också är ett globalt optimum [8]. Ett allmänt heltalsproblem med n variabler och m bivillkor kan formuleras som:

$$\begin{aligned} \min \quad & z = \sum_{j=1}^{n} c_j x_j \\ \text{då} \quad & z = \sum_{j=1}^{n} a_{ij} x_j \leq b_j \\ & i = 1, ..., m, \quad j = 1, ..., n \\ & x_j \geq 0 \text{ heltal} \end{aligned}$$

Där c_j är kostnadskoefficienter, x_j är beslutsvariabler och a_{ij} är koefficienter till bivillkorsfunktionerna. Om alla variabler är heltalsvariabler har vi ett rent heltalsproblem. Om endast vissa variabler är heltalsvariabler kallas det ett blandat eller mixat heltalsproblem.

Vissa HP problem går att lösa genom att heltalskravet på beslutsvariablerna tas bort, relaxeras, vilket ger ett LP problem istället. Detta kallas för LP relaxationen av HP problemet. Lösning av den motsvarande LP relaxationen ger en gräns för HP problemet. I sällsynta fall kan alla beslutsvariablerna i lösningen till LP relaxationen bestå av heltal, då har även en giltig och dessutom optimal HP lösning erhållits [17]. Observera att om ej alla beslutsvariabler i lösningen till LP relaxationen är heltal, går det inte att få en optimal lösning till HP problemet genom att avrunda till närmaste heltal [17]. Det är inte ens säkert att avrundningen ger en tillåten lösning.

Ett optimeringsproblem är ett ickelinjärt HP problem (IHP) om minst någon av funktionerna är en olinjär funktion (gäller objektfunktionen och bivillkoren). Det finns väldigt lite teori och få lösningsmetoder utvecklade för problemformuleringar av denna typ [8]. Anledningen är att heltalsproblem och ickelinjära problem, generellt sett, är svåra problem att lösa var för sig och när de kombineras så kan extremt svårlösta

problem erhållas. En vanlig approach för dessa problem är att försöka approximera olinjära funktioner med linjära funktioner, samt att försöka relaxera heltalsvillkoren [8]. Dessa metoder kan dock innebära att avkall på modellens exakthet måste göras. En annan approach som på senare tid börjar bli alltmer vanlig är att använda en s.k. heuristisk metod, se vidare kap. 4.4.

Det kan här vara lämpligt att här ge ett citat ifrån Lundgren [8] angående svårigheter med att lösa HP problem:

> Sammanfattningsvis kan man säga att det inte finns något enkelt resonemang som förklarar vilka strukturer som gör ett heltalsproblem lättare eller svårare att lösa, utan det är en "insikt" som man erhåller ju mer man lär sig om, och arbetar med, heltalsoptimering.

Det problem som behandlas i denna rapport är ett ickelinjärt heltalsproblem. Problemet hör till ett optimeringsområde som brukar benämnas kombinatorisk optimering.

4.3 Introduktion till kombinatorisk optimering

Då en eller flera av beslutsvariablerna är begränsade till att vara heltal är problemet av kombinatorisk natur. Denna klass av optimeringsproblem kallas ofta för *kombinatoriska optimeringsproblem* (KO). Med KO menas att en lösning till ett problem sökes, som optimerar en målfunktion över en diskret (kombinatorisk) mängd. Både HP och IHP är delmängder till familjen av KO problem. Även LP och ILP problem som har en ändlig mängd lösningar är delmängder till KO [11]. I KO problem kan både objektfunktionen och bivillkorsfunktionerna vara olinjära och även okontinuerliga [17].

Detta medför att KO problem i allmänhet är svåra att lösa. En naiv lösningsstrategi vore att lista alla möjliga lösningar och välja ut den lösning som är bäst. Men med tanke på det stora antalet möjliga lösningar är denna approach inte lämplig för annat än väldigt små problem. För problemlösning inom KO existerar inte några generella lösningsmetoder, istället är angreppssättet mer problemspecifikt. Problem som studeras brukar oftast delas in i problemklasser med gemensamma egenskaper. Genom att studera varje problemklass kan sedan algoritmer "skräddarsys" för högsta möjliga effektivitet [11].

Några klassiska exempel på kombinatoriska optimeringsproblem är:

`Handelsresandeproblemet:` En försäljare ska åka till m st städer och får bara besöka varje stad en enda gång och ska slutligen återkomma till den stad han kom ifrån. Problemet är att planera rutten så att vägen han ska åka minimeras.

`Kappsäcksproblemet:` I en kappsäck som rymmer m st objekt, ska n st objekt placeras, där $n > m$ och objekt i har vikten w_i och profiten p_i. Problemet är att packa kappsäcken så att profiten maximeras.

`sekvensierings- och scheduleringsproblem:` m st maskiner ska hantera n st uppgifter, problemet består av att finna vilken sekvens dessa uppgifter ska hanteras hos maskinerna.

Kombinatorisk optimering går ofta under andra benämningar, vanligen anses orden "kombinatorisk" och "diskret" vara synonyma, detsamma gäller även för orden "optimering" och "programmering". Termen "kombinatorisk optimering" används alltså synonymt med termerna "kombinatorisk programmering", "diskret optimering" och "diskret programmering" [11].

4.4 Klasser av lösningsmetoder

Lösningsmetoder för KO problem delas vanligtvis in i två klasser av metoder, *optimerande* och *icke-optimerande* metoder. Skillnaden ligger i att optimerande metoder kan garantera att en globalt optimal lösning till problemet hittas. Medan icke-optimerande metoder inte kan ge några garantier för att den funna lösningen är optimal. De optimerande metoderna är de mest använda metoderna, bland annat på grund av att de funnits och varit accepterade längre än de icke-optimerande metoderna. Under senare år har de icke-optimerande metoderna utvecklats enormt och blivit populära framförallt på grund av att de flesta är relativt enkla att designa och implementera samt ofta ger mycket bra lösningskvalitet.

Till det optimeringsproblem som behandlas i denna rapport har lösningsapproacher inom både optimerande och icke-optimerande metoder utvärderats. Eftersom det inte är helt självklart vad den icke-optimerande optimeringsparadigmen innebär, kommer den förklaras närmare här.

Icke-optimerande metoder brukar kallas för heuristiska[6] metoder eller kort och gott heuristiker [15]. Heuristiker brukar i sin tur beskrivas som "tumregelbaserade metoder", d.v.s. metoderna är designade för en viss typ av optimeringsproblem och de är anpassade till att utnyttja det specifika problemets struktur. Heuristiska metoder kan bygga på mycket enkla principer och vara baserade på någon ide som inte kräver någon speciell optimeringskunskap. Men de kan även vara mycket sofistikerade och baseras på avancerad optimeringsteori [8]. Heuristiker används ofta för att lösa svåra optime-

[6]Ordet heuristik kommer från det grekiska ordet heuriskein som betyder "att finna" eller "att upptäcka", och myntades första gången när den grekiske matematikern Arkimedes upptäckte principen för flytande kroppar och utbrast "Heureka Jag fann det". [8]

ringsproblem som t.ex. heltalsproblem och kombinatoriska problem. Lösningen hamnar ofta nära optimum men det går oftast inte att avgöra hur nära optimum den är [8]. Metodernas "godhet" brukar bedömas med hjälp av experimentell erfarenhet och/eller statistisk analys [11].

Det finns många definitioner på begreppet heuristik. En av de bättre definitioner som på ett bra sätt förklarar begreppet är Reeves [15] definition som här citeras:

Definition
A heuristic is a technique which seeks good (i.e. near-optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is.

En av de stora fördelarna med att använda heuristiker är att de är väldigt toleranta vad gäller modellen av det verkliga problemet. Heuristiker är ofta mer flexibla än optimerande metoder, och är kapabla att klara av mer realistiska (komplexa) bivillkor och målfunktioner [15]. Det innebär att inskränkningar på modellen inte brukar behöva göras då en heuristisk lösningsstrategi används.

Förutom ovanstående resonemang finns det flera anledningar till att överväga att använda en heuristik för att lösa ett problem:

- En optimerande metod tar för lång tid.
- En optimerande metod kräver alltför mycket minnesutrymme.
- Indata till problemet är osäkert och därför kan det räcka med att nöja sig med att finna en när-optimal lösning.
- En enkel heuristik är lättare att förstå för en "icke-expert", vilket underlättar beskrivning av vad som händer i optimeringsproceduren.
- Många heuristiker klarar av de mest olinjära problem.

5 Optimerande metoder

I detta kapitel beskrivs den optimerande metod som är en möjlig aspirant till att lösa optimeringsproblemet.

5.1 Branch & bound

Branch & bound är en divide-and-conquer approach. Den delar successivt upp (eng. branch) mängden av tillåtna lösningar i mindre delmängder och löser de resulterande delproblemen över dessa delmängder istället. Uppdelningen kan betraktas som att mängden av tillåtna lösningar delas upp i en trädstruktur där varje nod består av ett delproblem. Iden med B&B är att den optimala lösningen är den bästa bland de optimala lösningarna till delproblemen. Gränser (eng. bounds) för bästa funna lösningen hittills fås fram genom att lösa de relaxerade delproblemen[7]. Detta används till att reducera antalet delproblem. Om bästa lösningen för ett delproblem vid en nod är sämre än gränsen kommer delproblemet att elimineras från trädet. Vad som sker vid elimineringen är att metoden skär bort delmängder av lösningsmängden och minskar antalet lösningar som ska undersökas [8].

De noder som kapas bort bokförs för de ska kunna gå att nå vid senare tillfälle. Om metoden kommer till ett läge där ingen optimal lösning hittas kan den bakåtspåra till de bortkapade noderna och på så sätt kunna fortsätta sökandet [11]. B&B processen kallas implicit uppräkning eftersom alla lösningar inte behöver undersökas [4].

Den generella lösningsproceduren är som sagt baserad på lösningen av LP relaxerade problem. Stegen i denna procedur är:

Steg 1: Relaxera lösningsrummet genom att släppa på heltalskravet (d.v.s konvertera HP problemet till ett LP problem).

Steg 2: Lös det relaxerade LP problemet till optimum (vanligtvis m.h.a. simplexmetoden). Är alla beslutsvariabler i lösningen heltal så är denna lösning även en lösning till HP problemet. Är åtminstone en av variablerna skilt ifrån heltal, fortsätt till steg 3.

Steg 3: Starta i det kontinuerliga optimum som erhölls i steg 2, lägg till speciella villkor som iterativt kommer att tvinga den optimala LP lösningen emot den önskade optimala heltalslösningen.

De adderade villkoren kommer att eliminera delar av det relaxerade lösningsrummet, men aldrig någon av de tillåtna heltalslösningarna. Det finns dock ingen garanti att

[7]Kravet på heltalighet slopas och problemet löses via en vanlig LP-metod.

denna metod alltid är effektiv då det gäller att lösa HP problem.

Fördelar:
Ger garantier för att funnet optimum är ett globalt optimum.

Nackdelar:
För stora problem blir beräkningstiden orimligt stor. Metoden kan komma att kräva inskränkningar på optimeringsmodellen.

6 Icke-optimerande metoder

I detta kapitel beskrivs de icke-optimerande metoder som är möjliga aspiranter till att lösa problemet. Notera att sektionen som beskriver strategin Simulated annealing är mycket mer omfattande och detaljerad än övriga. Detta beror på att det var den metoden som slutligen implementerades i verktyget.

6.1 Lokalsökning

Lokalsökningsmetoder baseras på iden att för en given lösning söks närliggande omgivningar igenom efter en ny bättre lösning. När en bättre lösning hittas kommer sökningen fortsätta leta i omgivningen till denna lösning. Eftersom sökningen begränsas till den närliggande omgivningen är den lokal. När ingen bättre lösning påträffas i omgivningen är lösningen ett lokalt optimum, och antas vara en approximation till den optimala lösningen och sökningen är färdig. Lokalsökningsmetoder brukar vara attraktiva att använda för kombinatoriska optimeringsproblem då de ofta har en naturlig omgivning, och är enkla att förstå och implementera [15].

Lokalsökningsmetoder uppträder dock närsynt då förflyttning alltid sker till en bättre lösning. Detta innebär att metoden har lätt för att fastna i lokala optimum. Det som skiljer lokalsökningsmetoder ifrån giriga algoritmer (se kap. 6.3) är att de arbetar med partiella lösningar, som kompletteras med ett element i taget. Lokalsökning arbetar hela tiden med fullständiga lösningar [11].

Fördelar:
Enkel att förstå och implementera. Ger snabbt en lösning. Bra att tillämpa på KO då dessa oftast har en naturlig omgivning.

Nackdelar:
Fastnar i lokala optima. Ger ingen som helst garanti för att lösningen ens ligger i närheten av den optimala. Resulterande lösning blir beroende av startlösning.

6.2 Multistart

Multistart fungerar som lokalsökning, men som namnet antyder startas metoden om ett antal gånger. Varje start sker med olika startlösningar, vilket leder till att ett flertal lokala optima genereras, av vilka den bästa sedan väljs ut.

Fördelar:
Snabb och enkel att implementera. Lätt att förstå.

Nackdelar:
Resultatet blir beroende av val av startlösning. Om något av de funna lokala optimumen är av bra kvalitet beror mest på slumpen/tur.

6.3 Giriga algoritmer

Giriga algoritmer attackerar problemet genom att konstruera lösningen i en serie av steg. Denna typ av algoritmer är väldigt populära eftersom de är enkla att förstå och implementera. Den allmänna idén är enkel: Tilldela värden för alla beslutsvariabler en efter en och för varje steg gör det bästa möjliga beslutet. Algoritmen utför alltså det drag som för tillfället ger bäst "profit", därav namnet *girig*. Denna approach är dock kortsiktig då de optimala besluten vid varje steg inte alltid ger det optimala slutresultatet [10].

Fördelar:
Enkel att förstå, enkel att implementera. Ger väldigt snabbt en lösning.

Nackdelar:
Ger ingen som helst garanti för att lösningen ens ligger i närheten av den optimala. Fastnar snabbt i lokala optima.

6.4 Simulated annealing

Simulated annealing[8] (SA) är en heuristisk teknik kapabel att tillämpas på väldigt svåra kombinatoriska optimeringsproblem, och som ändå ger relativt goda lösningar [15].

Simulated annealing baseras på en analogi ifrån termodynamiken. Den simulerar den fysikaliska process som sker då ett smält material långsamt kyls. Algoritmen för att simulera kylningen av ett smält material publiserades av Metropolis *et al.* [9] redan 1953, algoritmen kom att kallas Metropolisalgoritmen. Kylningsprocessen kan simuleras genom att betrakta materialet som ett system av partiklar. Om ett smält material kyls kommer strukturen hos det fasta materialet som bildas att bero på hur snabbt temperaturen sänks [15]. Det som händer då temperaturen långsamt minskas är att partiklarna ordnar sig på ett sätt som minimerar den totala energinivån i materialet. Om kylningen sker för fort kommer ojämnheter att frysa fast i strukturen och det innebär att energinivån är högre än i det perfekt strukturerade materialet [10]. Detta

[8]Simulated annealing brukar på svenska kallas simulerad kylning, simulerad stelning eller simulerad anlöpning. I denna rapport kommer det engelska begreppet användas då det är mest känt under det namnet, och även oftast används även på svenska.

kallas quenching, eller härdning på svenska. Orsaken till att ojämnheter fryser fast i materialet är att alla partiklar inte hinner ordna sig så de hamnar i de tillstånd som minimerar totala energinivån hos systemet.

Fysikaliskt sett simulerar Metropolisalgoritmen ändringen av systemets energi då det kyls. Algoritmen genererar en störning i systemet och beräknar sedan den resulterande ändringen i energi. Om denna nya energi är lägre än tidigare kommer den alltid att accepteras. Är energin högre kan den accepteras med en viss sannolikhet. Processen repeteras ett visst antal iterationer vid varje temperatur tills systemet frusit. Accepterandet av högre energitillstånd sker på ett kontrollerat sätt genom en sannolikhetsfunktion som hanterar hur stor andel av dessa sämre tillstånd ska accepteras. Denna funktion förändras då algoritmen fortskrider. I termodynamiken har ett system med temperaturen T en sannolikhet att hamna i ett högre energitillstånd. Om energiökningen är ΔE ges sannolikheten av

$$p(\Delta E) = e^{(-\Delta E/kt)} \tag{6}$$

där k är Boltzmanns[9] konstant. Andelen högre energitillstånd som accepteras beror på storleken av energiökningen samt på vilken temperatur som råder. Denna regel för acceptans av nya tillstånd kallas för *Metropolis kriteriet* [6].

Kirkpatrick *et al.* och Cerny föreslog, oberoende av varandra, 30 år efter Metropolis att denna algoritm kunde tillämpas till att söka efter lösningar i kombinatoriska optimeringsproblem [15]. I tabell 5 visas analogier mellan termodynamiska begrepp som används i Metropolis algoritm och motsvarande begrepp inom optimering.

Fysikaliskt system	**Optimeringsproblem**
Systemets tillstånd	Möjlig lösning
Energi	Kostnadsfunktion
Grundtillstånd	Optimal lösning
Härdning	Lokalsökning
Temperatur	Kontrollparameter T
Långsam kylning	Simulated annealing
Ändring av tillstånd	Lösning i omgivningen

Tabell 5: Analogi mellan begrepp inom termodynamik och simulated annealing. [10]

[9]Boltzmanns konstant, $k = 1.38066 \cdot 10^{-23} JK^{-1}$.

Simulated Annealing ska inte ses som en enda algoritm, utan mer som en familj av algoritmer eller som en heuristisk strategi. Flera beslut måste tas då metoden ska implementeras för ett specifikt problem. Vissa av dessa är generiska beslut relaterade till parametrar i metoden. Dessa inkluderar beslut om vilken starttemperatur som ska väljas, vilket kylschema som ska användas och vilket avbrottsvillkor ska användas för att metoden ska avsluta. Det sista beslutet innebär att reda ut när systemet kan anses vara 'fryst'. Det finns även en del problemspecifika beslut som t.ex. vad som ska utgöra en möjlig tillåten lösning, utseende hos kostnadsfunktion (d.v.s. målfunktion), struktur på omgivningen till en lösning och hur en ny lösning ska slumpas fram ur omgivningen.

Den generella algoritmen för Simulated Annealing ser ut på följande sätt [15]:

Algoritm 1: Simulated Annealing.

1 Välj en initial lösning s_0

2 Välj en initial temperatur $T_0 > 0$

3 Välj en temperaturminskningsfunktion $decrease\_temp()$

4 Så länge som avbrottvillkoret ej är uppfyllt

- **4.1** Så länge som antalet iterationer är mindre än maximalt antal
  - **4.1.1** Välj slumpmässigt en ny lösning $s \in N(s_0)$
  - **4.1.2** Beräkna kostnadsändringen $\Delta = kostnad(s) - kostnad(s_0)$
  - **4.1.3** Om $\Delta < 0$ (lösning s är bättre än s_0)
    - **4.1.3.1** Sätt $s_0 = s$
  - **4.1.4** Annars (om lösningen är sämre)
    - **4.1.4.1** generera x slumpmässigt inom området (0, 1)
    - **4.1.4.2** Om $x < e^{-\Delta/T}$ (d.v.s. om x mindre än acceptanssannolikhet)
      - **4.1.4.2.1** Sätt $s_0 = s$ (sämre lösning accepterades)
- **4.2** $T = decrease\_temp(T)$

5 s_0 är en approximation till den optimala lösningen

Notera att temperaturparametern T nu bara är en kontrollparameter som kontrollerar förloppet och inte har någon fysikalisk relevans. Därför har Boltzmanns konstant uteslutits från acceptanssannolikheten. Dock brukar kontrollparametern T ändå refereras som 'temperaturen'.

Det finns åtskilligt med forskning om metoden, dess statistiska egenskaper, tillämpningar m.m. som inte kommer behandlas i detalj här. En del relevant material kommer tas

upp i nedanstående sektioner. Den intresserade läsaren hänvisas vidare till böckerna av van Laarhoven och Aarts [6] samt Aarts och Korst [2] som tar upp det mesta om teorin bakom Simulated Annealing. För mer information om metoden, dess tillämpningar och resultat m.m. bör Aarts och Lenstra [1], Reeves [15] samt Michalewicz och Fogel [10] studeras.

6.4.1 Konvergens

I varje temperatursteg kommer ett antal tillstånd att genereras. Sannolikheten att gå från ett tillstånd i till ett annat tillstånd j kan representeras på matrisform p_{ij}. Eftersom temperaturen är konstant, på grund av att alla genererade tillstånd sker inom samma temperatursteg, kommer övergångssannolikheten p_{ij} endast att bero på tillstånd i och j. Denna typ av övergångsmatris ger upphov till en s.k. homogen Markovkedja [6]. Så länge det är möjligt att hitta en sekvens av transformationer som kan transformera en godtycklig lösning till en annan godtycklig lösning med nollskild sannolikhet, kommer processen att konvergera emot en stationär fördelning. Denna stationära fördelning kommer vara oberoende av startlösningen [15]. Detta kan ses som att systemet uppnått en form av jämvikt för denna temperatur.

I Simulated annealing är dock inte temperaturen konstant utan kommer att minskas efter ett antal iterationer i varje temperatursteg. Detta kan ses som att ett antal homogena Markovkedjor genereras (en för varje temperatursteg), eller att en enda lång inhomogen Markovkedja bildas. Här kommer inte teorin för den inhomogena Markovkedjan att behandlas då den innebär att p_{ij} kommer bero på temperaturen också, d.v.s. den kommer bero på det antal iterationer som tidigare utförts.

Då temperaturen minskar kommer den stationära fördelningen att gå emot en uniform fördelning över mängden av optimala tillstånd [15]. Det som händer är att väntevärdet och variansen för kostnadsfunktionen minskar med minskande temperatur [17]. Generationsprocessen repeteras för varje ny temperatur tills sannolikhetsfördelningen av systemets tillstånd når Boltzmannfördelningen [17]

$$\Pi_i = \frac{e^{-C(i)/T_k}}{\sum_{j \in \Omega} e^{-C(j)/T_k}}, \quad \text{där } i \in \Omega \tag{7}$$

där Π_i är sannolikheten att vara i punkt i, och funktionen $C(\cdot)$ är en kostnadsfunktion. Denna fördelning säger att en högre sannolikhet för att uppnå en lägre kostnad för slutpunkten bara kan uppnås efter tillräckligt många iterationer vid temperaturen T_k

[17]. Låt en homogen Markovkedja bestående av ett ändligt antal tillstånd vara en sekvens $\{X_m \mid m = 1, 2, ...\}$ av slumpvariabler som antar värdena $\Omega = \{1, 2, ..., n\}$. Det har visats [6] att om en Markovkedja bland annat är av oändlig längd, kommer SA att hitta en globalt optimal lösning med sannolikheten 1. Låt Ω^* vara mängden av index av optimala tillstånd i Ω. Då, när T_k närmar sig noll medan k går emot oändligheten följer att:

$$\lim_{T_k \to 0} \Big(\lim_{m \to \infty} \Pr(X_m \in \Omega^*) \Big) = \lim_{T_k \to 0} \sum_{i \in \Omega^*} \Pi_i = 1 \tag{8}$$

Uttrycket för gränsvärdet i ekv. 8 kan uttryckas i ord som att då markovkedjans längd går emot oändligeheten, samtidigt som temperaturen går emot noll, kommer sannolikheten att hitta en globalt optimal lösning att gå emot ett. Det vill säga att en globalt optimal lösning kommer att hittas under dom förutsättningarna. Eftersom Markovkedjan är av ändlig längd för alla implementationer av algoritmen kan den inte konvergera asymptotiskt emot den globalt optimala lösningen. Alltså måste den oändligt långa markovkedjan approximeras med en kedja av ändlig längd. Vilket innebär att algoritmen inte är garanterad att finna ett globalt optimum med sannolikheten 1.

Enligt Reeves [15] har Aarts och van Laarhoven visat att om det ska gå att approximera ett temperatursteg i Simulated annealing med en homogen Markovkedja måste antalet iterationer minst vara kvadratiskt i storleken på lösningsrummet. Då lösningsrummet oftast är exponentiellt i storlek betyder detta att exekveringstiden för Simulated annealing med sådana garantier på optimalitet kommer vara exponentiella. En svårighet är att ha en bra avvägning mellan antal övergångar som ska utföras på varje temperaturnivå och hur många steg som sänkningen av temperaturen ska ske med.

Ovanstående konvergensresultat har gett Simulated annealing en grad av respekt, större än vad vanliga heuristiker brukar kunna få. Reeves [15] anser att det bidragit till intresset för, och populariteten hos SA.

6.4.2 Kylschema

Ett sätt att påverka konvergenshastigheten för Simulated annealing är genom valet av kylschema. Ett kylschema består av övre och undre gränser för temperaturparametern, hastigheten med vilken temperaturen minskas med, samt längd på Markovkedjorna.

Det finns två typer av kylscheman, antingen enkla (*eng.* simple) eller utarbetade (*eng.* elaborate). Skillnaden ligger i att de som kallas enkla är alla baserade på empiriska regler istället för teoretiskt baserade val [6]. De kylscheman som behandlas i denna

rapport hör till de enkla. Förutom att dela in kylscheman enligt hur de är framtagna (empiriskt eller teoretiskt) brukar de även delas in i klasserna A eller B beroende på hur regler för hur kontrollparametern minskas samt hur längden på markovkedjorna hanteras. Uppdelningen i klasser ser ut enligt följande [6]:

- Klass A: Variabel längd på markovkedjorna och fix minskning av kontrollparametern.
- Klass B: Fix längd på markovkedjorna och variabel minskning av kontrollparametern.

Med "fix" och "variabel" menas oberoende, respektive beroende, på utvecklingen av algoritmen. Dock finns det exempel på kylscheman där både temperaturminskningen och längden på markovkedjorna varierar [6]. Då ett kylschema valts finns det inget enkelt sätt att få fram värden på alla parametrar i schemat för ett specifikt problem. Det bästa verkar vara att experimentera och testa sig fram, ett citat av Reeves [15] belyser detta

> In terms of deciding on the values of the parameters for the schedule chosen, there is no easy way of achieving this and almost all the successful applications reported in the literature state that the best parameters were determined after much experimentation.

Starttemperatur
Enligt Aarts och Laarhoven [6] bör en starttemperatur väljas på ett sätt, som ger att nästan alla övergångar accepteras av metropoliskriteriet (ekv. 6). Orsaken till att de flesta övergångar bör kunna accepteras vid start är att den slutliga lösningen ska vara oberoende av startlösningen [15]. Detta innebär att för en ändring av kostnaden C mellan lösning i och j bör $exp(-\Delta C_{ij}/T_0) \approx 1$ gälla för nästan alla i och j. Kirkpatrick *et al.* [16] föreslog följande empiriska regel: välj ett stort värde på T_0 och genomför ett stort antal övergångar och titta på proportionen av acceptanssannolikheten. Acceptanssannolikheten χ definieras som

$$\chi = \frac{\text{Antalet accepterade övergångar}}{\text{Antalet föreslagna övergångar}}. \tag{9}$$

De föreslår att om χ är mindre än ett önskat värde χ_0 (Kirkpatrick *et al.* [16] använde $\chi_0 = 0.8$), dubbla nuvarande värdet för T_0. Fortsätt denna procedur tills $\chi_0 > \chi$. Denna empiriska regel har använts och även förädlats av många författare enligt Laarhoven

[6]. Enligt Laarhoven och Aarts har David Johnson bestämt T_0 genom att beräkna den genomsnittliga ökningen i kostnad $\overline{\Delta C}^{(+)}$ för ett antal slumpmässiga övergångar. Om denna parameter används istället för bara skillnaden i kostnad för en övergång i metropoliskriteriet, fås initiala[10] acceptanssannolikheten som

$$\chi_0 = exp(-\overline{\Delta C}^{(+)}/T_0). \tag{10}$$

Ur ekvation 10 kan sedan starttemperaturen T_0 lösas ut som

$$T_0 = \frac{-\overline{\Delta C}^{(+)}}{ln(\chi_0^{-1})}. \tag{11}$$

Ekvationer likartade 11 har föreslagits av ett antal forskare[11].

Sluttemperatur
För bestämning av sluttemperaturen (d.v.s. avbrottsvillkoret för optimeringen) finns det principiellt sett två val. Antingen bestäms ett visst antal temperatursteg som ska utföras, eller bestäms att optimeringen ska avslutas då ett antal markovkedjor i följd inte förbättrat kostnaden för lösningen. I det första valet väljs en slutgiltig temperatur, medan det senare kan ses som ett försök att skatta systemets fryspunkt.

I teorin ska temperaturen minska ner till noll innan avbrottsvillkoret är uppfyllt [15]. Dock är det inte nödvändigt att låta den gå så långt eftersom då temperaturen går mot noll kommer sannolikheten att gå ut ur ett lokalt optimum att bli försumbar. Detta kan tolkas som att systemet frusit. Analogin med termodynamiken innebär att det intuitivt går att inse att en optimal lösning kan hittas, innan temperaturen noll uppnåtts, eftersom de flesta material har en fryspunkt över den absoluta nollpunkten. Problemet är att veta vilken "frystemperatur" ett specifikt optimeringsproblem har. Enligt Reeves [15] föreslår Lundy och Mees följande ekvation för att beräkna sluttemperaturen (d.v.s. då temperaturen anses ha passerat materialets fryspunkt)

$$T \leq \frac{\epsilon}{ln[(|S|-1)/\theta]}, \tag{12}$$

där S är storleken på lösningsrummet. Ekvation 12 är designad att producera en lösning som ligger inom ϵ av optimum med en sannolikhet θ.

[10]Med initiala acceptanssannolikheten menas den sannolikhet som fås innan den första temperaturändringen.

[11]T.ex. Leong *et al.*, Skiscim och Golden Morgenstern och Shapiro, Aarts och Laarhoven, Lundy och Mees och Otten och van Ginneken.

Längd på Markovkedjor
Längden L_k på markovkedja k anger hur många övergångar som sker i lösningsrummet innan temperaturen sänks ett steg, varvid en ny markovkedja påbörjas. En regel för längden på markovkedjan som föreslagits av flera forskare[12] fungerar enligt följande; För varje värde på kontrollparametern T_k krävs ett minimum av accepterade övergångar. Detta innebär att L_k bestäms genom att antalet accepterade övergångar minst måste vara lika med ett fixt antal övergångar η_{min}. Men då temperaturen minskar kommer sannolikheten för accepterade övergångar att minska och slutligen kommer $L_k \to \infty$ då $T \to 0$. Detta medför att en gräns $\overline{L}$ måste sättas för hur långa Markovkedjorna får bli. Oftast väljs denna konstant att vara polynomisk m.a.p. problemstorleken [6].

Både Reeves [15] och Michalewicz [10] anser att antalet övergångar per temperatursteg bör vara proportionellt emot storleken på omgivningen. Vilket innebär att valet av omgivningsfunktion kommer att påverka längden på Markovkedjorna. Enligt Reeves [15] är det även naturligt att låta längden variera mellan olika temperatursteg. T.ex. bör längden vara längre för lägre temperaturer för att försäkra sig om att de lokala optimumen blir så bra utforskade som möjligt. De vanligaste sätten att förändra längden är att öka den genom att multiplicera längden med en faktor > 1, eller genom att addera en konstant faktor för varje ny temperatur.

Minskning av temperaturparametern
Minskningen av temperaturparametern bör väljas så att det räcker med relativt korta markovkedjor för att återställa jämvikt i aktuell temperatur efter en temperaturminskning. Detta innebär att temperaturförändringarna måste vara små. Kirkpatrick *et al.* [16] föreslog en geometrisk minskningsregel som ges av

$$T_{k+1} = \alpha \cdot T_k, \;\; k = 0, 1, 2, \ldots, \tag{13}$$

där α är en konstant som är mindre än, och ligger i närheten av 1. De använde $\alpha = 0.95$ vilket gav små minskningar. Nackdelen med att ha små minskningar är att antalet temperatursteg blir många fler, medan få stora steg kräver väldigt långa Markovkedjor per steg. Denna regel har använts av många andra forskare[13] med värden på α mellan 0.5 till 0.99. Erfarenhet har visat att relativt höga värden för α fungerar bäst. De flesta rapporterade framgångar använder värden mellan 0.8 till 0.99, med en tonvikt mot den övre delen av spektrat [6].

En temperaturminskningsmetod som Aarts och Korst [2] härlett garanterar att den

[12]Bland annat av Kirkpatrick *et al.*, Johnsson *et al.*, Leong *et al.* och Morgenstern och Shapiro.
[13]T.ex. Johnson *et al.*, Bonomi och Lutton, Burkard och Rendl, Leong *et al.*, Morgenstern och Shapiro och Sechen och Sangiovanni-Vincentelli.

slutgiltiga distributionen nära approximerar den stationära fördelningen - en situation som de kallar kvasi-jämvikt. Metoden antar följande form

$$T_{k+1} = \frac{T_k}{1 + (T_k \cdot ln(1+\delta)/3\sigma_T)} \tag{14}$$

där σ_T är standardavvikelsen vid temperaturen T_k. Konstanten δ (som här kallas distansparametern) styr hur snabbt temperaturen ska minskas. Små δ ger små minskningar, medan stora δ ger stora temperaturminskningar. Dock är kylningshastigheten m.h.a. denna minskningsmetod långsammare än de flesta som används praktiskt [15].

Enligt Reeves [15] indikerar både teoretiska och empiriska forskningsresultat att det viktigaste med temperaturminskningsfunktioner är hastigheten med vilken temperaturen sänks och inte sättet den sänks på. Det är alltså inte så stor skillnad mellan den geometriska minskningsregeln i ekv. 13 och den mer avancerade regeln i ekv. 14 så länge som de kyler över samma temperaturområde med ungefär samma hastighet.

Omgivning
För att metoden ska kunna konvergera måste ett s.k. nåbarhetsvillkor upprätthållas. Detta villkor innebär att metoden kan konvergera så länge som en godtycklig lösning kan nås ifrån en annan godtycklig lösning m.h.a. ett antal övergångar [15]. Val av omgivning måste göras så att detta villkoret upprätthålls. Oftast är detta villkor enkelt att verifiera. Det är även bra att hålla nere storleken på omgivningen så den kan sökas av snabbare, samt att hålla omgivningen enkel och okomplicerad [15].

Kostnadsfunktion
Målfunktionen vid optimering med Simulated annealing brukar kallas för kostnadsfunktion p.g.a. analogin med termodynamiken. Inom fysiken beräknar funktionen energitillståndet för systemet, medan vid optimering beräknar funktionen kostnaden för aktuell lösning. Kostnadsfunktionen bör vara konstruerad så att den kan leda annealingprocessen mot lokala optima.

Effektiva kylningsscheman är svåra att designa. Kravet på en väldigt lång Markovkedja för varje temperatursteg implicerar att SA kräver långa beräkningstider för att finna en optimal eller när-optimal lösning. Det är möjligt att förbättra beräkningstiden genom att kombinera SA med andra algoritmer [17].

6.4.3 Fördelar och nackdelar

Generellt sett anses Simulated annealing vara en relativt långsam metod. Många lyckade applikationer av heuristiken har varit för komplicerade problem där inga tidigare

algoritmer funnits, eller där skräddarsydda algoritmer inte presterat tillräckligt bra [4]. Utplacering (placement) är ett av de områden där SA har tillämpats framgångsrikt. Inom IBM var SA vida accepterat redan på 80-talet och användes då till att lösa många olika sorters placeringsproblem [6].

Fördelen med denna metod gentemot vanlig heuristik är möjligheten att kunna acceptera sämre lösningar på ett kontrollerat sätt. Detta innebär att metoden kan undvika att fastna i lokala optima. Då metoden hamnar i ett lokalt optima finns det inga bättre lösningar att hitta. Slutligen kommer en eller flera sämre lösningar i omgivningen att accepteras och metoden kommer förhoppningsvis att ta sig ur det lokala optimat och fortsätta sökandet. Eftersom metoden kan förflytta sig mellan lokala optima kommer den därmed att undersöka flera kandidater till optimallösningen. Då sannolikheten att acceptera sämre lösningar minskar kommer metoden att alltmer likna en lokalsökningsmetod. Detta leder till att metoden börjar konvergera, och förhoppningsvis leder denna konvergens mot ett bra lokalt optimum.

Vanliga metoder inom optimering är att förenkla strukturen för det problem som ska behandlas. Vanligtvis görs antaganden om strukturen eller lösningsrummet och sedan tillämpas algoritmer från existerande teori. En styrka med Simulated annealing är att den tolererar att komplicerade, och därmed realistiska, strukturer används. Modellen som redan är en approximation av det verkliga problemet, behöver inte approximeras ytterligare för att passa den optimeringsmetod som ska användas [4].

Fördelar:
Relativt enkel att implementera.
Ställer inga krav på att funktioner i modellen ska vara linjära eller ens kontinuerliga.
Går med andra ord att tillämpa på de mest olinjära problem.
Existerar bevis för att teoretiska modellen för SA konvergerar asymptotiskt mot den optimala lösningen[14].
Följer upp och utforskar omgivning till bra lösningar.
Undviker att fastna i lokala minimum.
Lämplig att kombinera med andra metoder och algoritmer.
Relativt enkla bakomliggande mekanismer.

Nackdelar:
Val av parametervärden och framförallt val och konstruktion av kylschema spelar relativt stor roll för hur bra lösningen blir.
Tidsödande exekvering av algoritmen.
Vid för snabb kylning “fryser ojämnheter” i lösningen fast. Om metoden lyckas eller inte beror till stor del på hur temperaturen sänks.

[14]Det går förvisso inte att köra algoritmen i oändlighet, men det ger ändå en fingervisning om att den åtminstone konvergerar mot den optimala lösningen över tid.

Det finns problemspecifika parametrar i metoden som kan vara svåra att anpassa till problemet. Samt att det kan vara svårt att veta hur dessa ska anpassas på bästa sätt. Garanterar inte att lösningen är ett globalt optima.
Långa exekveringstider för bra konvergens mot optima.

6.5 Tabusökning

Tabusökning (TS) försöker, liksom SA, även den att undkomma lokala optima genom att försöka ta sig ur dessa. Till skillnad från SA som är en stokastisk metod, är TS en deterministisk metod [10]. Tabusökning lägger in intelligens i optimeringsförfarandet genom att använda sig av minne för att kunna påverka hur sökningen fortlöper [4].

Givet en funktion $f(x)$ som ska optimeras över en mängd S av möjliga lösningar, kan grunden för TS förklaras enligt följande. Tabusökningen börjar som en vanlig lokalsökning med att gå till den lösning som har bäst värde. Under sökningen efter bättre lösningar kommer TS att modifiera omgivningen $N(x)$ för nuvarande lösning till $N(x, h)$ genom att bland annat lägga till de nyligen utforskade lösningarna i en tabulista. Metoden fortsätter som vanligt tills den hamnar i ett lokalt optimum där alla lösningar i omgivningen är sämre. Med hjälp av minnet kommer då metoden att kunna ta sig ut ur lokala optima genom att gå över dessa sämre lösningar. Förutom att kunna ta sig ur lokala optima förhindrar tabulistan cykling, d.v.s. att metoden går runt i cirkel [10].

Tabuterminologin används för att beskriva ett hinder eller ett förbud, som under rätt villkor kan åsidosättas. Samliga lösningar i denna tabulista förbjuds att ligga i mängden $N(x, h)$. Minnet kommer alltså att bestämma hur $N(x, h)$ ser ut och på så sätt (till viss del) styra hur lösningsrummet ska utforskas. Varje gång en ny lösning undersöks kommer föregående att läggas till i listan, och den äldsta lösningen i listan tas bort. Det är inte enbart de nyligen undersökta lösningarna som läggs till i tabulistan utan även andra relaterade lösningar kan läggas till, t.ex. lösningar som har någon egenskap gemensam med den lösning som är tabu. Den process som ger lösningar tabustatus är designad att tvinga fram utforskandet av nya lösningar [10]. Hur länge lösningar ligger kvar i tabulistan beror på tabulistans längd.

Vid vissa tillfällen kan dock tabulistan ignoreras, nämligen då en förbjuden lösning leder till den bästa lösningen som hittills återfunnits. Denna förbjudna lösning har såklart inte hamnat i tabulistan p.g.a. att den blivit besökt tidigare, utan den har hamnat där eftersom den har en relaterad egenskap med en av de lösningar som lagts till [8].

I TS finns det två minnen [8], ett kortsiktigt och ett långsiktigt. det kortsiktiga minnet är själva tabulistan. Det långa minnet samlar statistik över hur lösningarna sett ut. Det kan exempelvis vara hur många gånger som en variabel antagit ett visst värde. T.ex. hur många gånger en typ befunnit sig i en viss position. Statistiken kan användas på två sätt. Antingen kan en intensifiering utföras, vilket betyder att t.ex. en variabel fixeras till det värde som varit mest vanligt under lösningsproceduren. Om en variabel har haft samma värde många gånger betyder det att den gärna vill anta just det värdet. Genom denna intensifiering kommer sökningen att intensifieras bland lösningar med den egenskapen. Om istället värdet på variabeln fixeras till det minst vanliga, leder det till en diversifiering och innebär att sökningen förflyttar sig till områden där algoritmen inte letat så mycket tidigare [8].

En generell algoritmbeskrivning för tabusökning [8], (antar minproblem) är

Algoritm 2: Tabusökning.

1 Utgå ifrån en tillåten lösning x_0 med kostnad $c(x_0)$. Sätt $k = 0$.

2 Kontrollera avbrottskriteriet.

3 Bestäm alla punkter i omgivningen $N^*(x_k)$.

4 Välj $x_k + 1 \in N^*(x_k)$som är tillåten enligt tabulistan och som minimerar $c(x)$. (Möjligt är att $c(x_{k+1}) > c(x_k)$.)

5 Uppdatera tabulistan. Sätt $k = k + 1$ och gå till steg 1.

Ett vanligt avbrottskriterium är att ett maximalt antal iterationer bestäms initialt. Sökningen kan även avbrytas när ingen bättre lösning hittats under ett visst antal iterationer, eller när metoden besökt samma optimum ett visst antal gånger.

Den viktigaste parametern att kalibrera vid tabusökning är längden på tabulistan. Det magiska talet 7 brukar anges som ett lämpligt val för många problemtyper, men betydligt större värden kan också komma ifråga [8].

Fördelar:
Undviker att fastna i lokala optima.
Empiriskt sett fungerar den riktigt bra, även för svåra problem.
Metoden ställer inga krav på att funktioner i modellen ska vara linjära eller ens kontinuerliga.
Lämplig att kombinera med andra metoder och algoritmer.

Nackdelar:
Beräkningsintensiv då den vid varje steg beräknar kostnaden för alla lösningar i omgivningen till nuvarande lösning.
Fler parametrar än SA att bestämma. Metoden är dessutom mer komplex än SA.

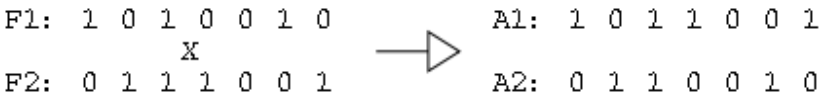
6.6 Genetiska algoritmer

Genetiska algoritmer (GA) kan ses som ett intelligent användande av slumpmässig sökning. GA utvecklades först av Holland *et. al.* vid University of Michigan under 60- och 70-talen [15]. Benämningen Genetiska algoritmer kommer ursprungligen från en analogi mellan representationen av en struktur med hjälp av vektorkomponenter, och den genetiska strukturen hos en kromosom. Denna kromosom brukar i optimeringssammanhang ofta kallas för en vektor, sträng eller lösning.

Vid framavling av t.ex. djur, letar uppfödaren bland avkommor efter vissa önskvärda karaktärsdrag. Dessa karaktärsdrag bestäms på den genetiska nivån genom sättet djurets föräldrars kromosomer kombinerats på. På liknande sätt kan GA användas för att söka efter bättre lösningar genom att kombinera delar av redan existerande lösningar. Denna parallell är inte helt exakt men Holland ansåg den tillräckligt bra för att föreslå den som en lösningsmetod för optimeringsproblem.

I många applikationer av GA används ofta en kromosom som enbart består av ettor och nollor, som t.ex. `1 0 1 0 0 1 0`. I diskussionen nedan kommer denna representation att användas för att ge bättre förståelse för GA. Flera s.k. *genetiska operatorer* har identifierats för att manipulera dessa kromosomer. De vanligaste är *korsbefruktning* (utbyte av sektioner i föräldrarnas kromosomer), och *mutation* (slumpmässig modifiering av kromosomen).

Vid korsbefruktning väljs korsbefruktningspunkter ut i föräldrakromosomerna. Dessa punkter bestämmer vilka delar av kromosomerna som ska byta plats för att skapa en ny kromosom. I figur 4 visas korsbefruktningen i punkt `X` mellan två föräldrars kromosomer, F1 och F2. Avkomman blir de två nya kromosomerna A1 och A2.



Figur 4: Korsbefruktning av två kromosomer.

Muteringsoperatorn är enkel, den går bara igenom strängen (t.ex. `1 0 1 0 0 1 0`) och ändrar varje bit med en viss sannolikhet. Sannolikheten är vanligtvis låg för varje bit, oftast lägre än $1/n$ där n är antalet bitar i strängen.

Variabler i en kromosom kallas för *gener*, de möjliga värdena på generna kallas *alleler* och positionen av en variabel i kromosomen kallas dess *locus*. I t.ex. kromosomen `1 0 1 0 0 1 0` har den andra genen värdet 0 och locus 2. Med genotyp menas själva strukturen hos kromosomen, d.v.s. den kodade sträng som processas av algoritmen. Med fenotyp avses den fysiska strukturen, i detta fall de avkodade parametrarna.

För kombinatorisk optimering används GA genom att algoritmen håller en population av M st. kromosomer av potentiella föräldrar, vilkas fitnessvärden beräknats. Varje kromosom är en lösning till problemet, och dess fitness-värde relaterar till det värde på målfunktionen som denna lösning ger. Då en avkomma ska produceras väljs sedan korsbefruktningspunkterna i föräldrakromosomerna.
Vid val av föräldrar finns många olika alternativ, antingen väljs en eller båda ut m.a.p. dess fitnessvärden, eller så slumpas en eller flera ut. Det finns även fall där fler än två föräldrakromosomer används. Flera korsbefruktningspunkter i kromosomerna kan väljas och sedan måste även beslut om mutationer tas vid olika tidpunkter. Vad som är bäst att göra för ett specifikt optimeringsproblem är upp till den som designar heuristiken. Startpopulationen av lösningar har ofta stor betydelse för slutresultatet.

Det koncept som Holland utvecklade för den teoretiska analysen av GA kallade han för ett *schema*[15]. Schemat är en mall som möjliggör att likheter mellan kromosomer kan kvantifieras. Ett schema kan byggas genom att introducera symbolen $*$ i kromosomerna. Symbolen $*$ är ett s.k. "wild card" som kan anta vilket som helst av värdena 0 eller 1. Det används för att markera vilka gener som två eller flera kromosomer inte har gemensamma. T.ex. så har vektorerna `1 0 0 1 1 0 1` och `0 0 1 1 1 0 1` det gemensamma schemat `* 0 * 1 1 0 1`. Scheman kan ses som att de definierar delmängder av liknande kromosomer, eller hyperplan i ett n-dimensionellt rum där n är antalet gener i kromosomen [15].

Om en kromosom har längden n, består den av 2^n distinkta scheman eftersom varje locus kan anta antingen sitt riktiga värde eller ett $*$. En konsekvens av detta är att varje gång fitnessvärdet beräknas för en given kromosom, kommer information om medelfitnessen för samtliga av kromosomens scheman erhållas [15]. På detta sätt kan ett stort antal möjligheter testas genom att utföra ett litet antal försök. Denna egenskap kallar Holland för inneboende parallellism, egenskapen brukar även kallas för implicit parallellism. Frågan är hur många scheman som hanteras per generation. Det går att visa att under vissa villkor kommer en population av storlek M leda till att $O(M^3)$ scheman hanteras för en generation [15].

GA-algoritmen ser ut på följande sätt [17]:

Algoritm 3: Generell algoritm för Genetiska algoritmer.

1 Initialisera parametrar för den genetiska algoritmen

2 Skapa old_population slumpmässigt

3 För generation nr. 1 till max antal generationer

3.1 Radera new_population

[15]Ordet schema kommer från det grekiska verbet $\epsilon\chi\omega$ (uttalas echo och betyder ungefär "att ha"). Därav kom det att betyda form; dess plural är *schemata* [15].

3.2 Beräkna fitness för varje individ i old_population
3.3 Kopiera den individ som har högst fitness till lösningsvektorn
3.4 Så länge som antalet individer är mindre än populationsstorleken
3.4.1 Välj två föräldrar från old_population baserat på deras fitnessvärde
3.4.2 Utför korsbefruktning mellan föräldrarna för att skapa två avkommor
3.4.3 Mutera varje avkomma (baserat på mutation_rate)
3.4.4 Placera avkomman i new_population
3.5 Byt ut old_population mot new_population

4 lösnings_vektorn är den slutliga lösningen

Ovanstående GA simulerar en evolutionär process med n st individer (n punkter) i lösningsrummet.

Fördelar:
Implicit parallelism.
Ställer inga krav på linjäritet eller kontinuitet hos funktioner i modellen.
Undviker att fastna i lokala optima.
Lämplig att kombinera med andra metoder och algoritmer.

Nackdelar:
Kräver många kontrollparametrar.
Startpopulationen av lösningar har ofta stor betydelse för slutresultatet.

6.7 Neurala nät

Artificiella neurala nät (ANN) är en beräkningsparadigm som skiljer sig starkt emot de ovan beskrivna metoderna som baseras på vanliga von Neumann arkitekturer. Filosofin för ANN är att utifrån koncept ifrån biologin konstruera enkla matematiska modeller för att efterlikna hjärnans arbetssätt. Förutom analogin med biologin har metoden även rötter inom statistisk fysik och behandling av system med många frihetsgrader [15].

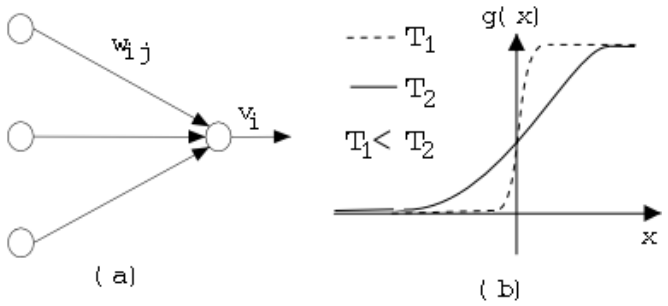
De grundläggande beräkningsenheterna i ANN är neuroner, som betecknas v_i. Dessa kan anta reella värden inom intervallet [0, 1] (ibland används istället [-1, 1]). Dessa neuroner är en förenklad modell av biologiska neuroner. De flesta neurala modeller har en *uppdateringsregel* (se figur 5a) på formen:

$$v_i = g(\sum_j \omega_{ij} v_i - \theta_i), \tag{15}$$

där v_j är alla neuroner som matar information till neuronen v_i (observera skillnaden i index på v här), genom vikterna (synapserna) ω_{ij}. Vikterna kan vara både positiva och negativa. Faktorn θ_i är ett gränsvärde, motsvarande membranpotentialen i en biologisk neuron. Den olinjära överföringsfunktionen $g(\cdot)$ är en sigmoidformad[16] funktion i stil med:

$$g(x) = 0.5[1 + tanh(x/T)], \tag{16}$$

där 'temperaturen' T sätter den inversa ökningen, se figur 5b. En låg temperatur motsvarar en brant funktion, medan hög temperatur ger en approximativt konstant funktion $g(\cdot)$.



Figur 5: (a) Neuronuppdatering och (b) respons hos överföringsfunktionen för två olika temperaturer.

Denna enkla modell över en artificiell neuron imiterar de huvudsakliga egenskaperna för en biologisk neuron. I en biologisk neuron adderas inkommande signaler linjärt, medan den resulterande utsignalen är starkt olinjär med avseende på den totala adderade insignalen. Om den adderade insignalen från matarneuronerna är större än ett visst gränsvärde θ_i kommer neuronen att ge en utsignal.

Det finns två typer av arkitekturer för att modellera neurala nät, *feed-forward nät* och *feed-backward nät*. I feed-forward nät processas signaler från en mängd in-enheter i botten av nätet till ut-enheter i toppen, lager för lager m.h.a. uppdateringsregeln, ekvation

[16]En sigmoidformad funktion är vanligtvis en *tanh*-funktion, d.v.s. en symmetrisk S-formad funktion.

15. I feed-backward nät är synapserna (vikterna) dubbelriktade. Aktivering fortsätter tills en fix punkt nåtts som påminner om ett statistiskt mekaniskt system. Generellt sett måste forwardnät läras upp istället för att programmeras med regler. Huvuddelen av applikationer för forwardnät ligger inom igenkänningsområdet. Backwardnät är bäst lämpade för optimering, och de arbetar genom att "känna" sig fram till bra lösningar.

Den typ av nät som visat sig bäst lämpad för kombinatorisk optimering är backward nät. Vid användning av feed-backward nät för att lösa KO-problem, måste problemet mappas till s.k. energifunktioner med ett diskret antal frihetsgrader. De ekvationer som används är s.k. mean-field ekvationer (MFT)

$$E = -\frac{1}{2}\sum_{i,j}\omega_{ij}s_i s_j, \tag{17}$$

Optimeringsproblemet mappas till formen av ekvation 17 med hjälp av noga utvalda värden på vikterna ω_{ij}. För varje problem väljs och fixeras värdena på ω_{ij} och får sedan inte ändras under optimeringen[17]. Neuronerna s_i tillåts sedan att hamna i ett stabilt tillstånd där lösningen till problemet ges av konfigurationen $\bar{s} = (s_1, s_2, ...)$ med minimal energi.

ANN approachen skiljer sig mot de exakta och de flesta heuristiska metoderna genom att det finns ingen trial-and-error mekanism i metoden. Istället 'känner' den sig fram till en bra lösning m.h.a. MFT approximationen. En fördel med denna lösningsapproach är att den är parallell till naturen vilket förenklar parallellisering av implementationen. Stegen för att lösa ett KO problem med ett artificiellt neuralt nätverk är följande [1]:

- Mappa problemet till det neurala nätet genom att välja en lämplig kodning av lösningar och en lämplig energifunktion.
- Använd så mycket som möjligt av kunskapen om systemet genom att studera fasövergångar för det lineariserade systemet (det system som ska hanteras är olinjärt).
- Lös de korresponderande MFT ekvationerna iterativt, genom minskning av parametern T.
- Då lösningarna till MFT ekvationerna konvergerat måste lösningarna kontrolleras att de verkligen uppfyller de begränsande villkoren. Om inte alla lösningar

[17]Här skiljer sig förfarandet gentemot andra tillämpningar av ANN genom att ω_{ij} i andra fall kan vara adaptiva och kan ändras.

uppfyller dessa villkor går det att antingen lägga till en girig algoritm till algoritmen för att hantera detta, eller prova att räkna fram nya lösningar till MFT genom att modifiera koefficienterna.

Med avseende på kvalitet ger generellt ANN metoden resultat som är jämförbara med de som fås med andra approximativa state-of-the-art metoder, som t.ex. Simulated annealing [15].

Fördelar:
Ny approach som skiljer sig stort med de andra möjliga metoderna. Underlättar parallell implementation.
Förmågan att lära sig hur problemet kan lösas ger metoden en mycket stor flexibilitet och gör den dessutom väldigt kraftfull.
Fastnar inte i lokala optima.

Nackdelar:
Många parametrar som måste bestämmas, fler än för SA, TS och GA.
Metoden är dessutom klart mer komplex än SA, TS och GA.

7 Utvärdering av lösningsmetoder

I detta kapitel kommer de tidigare beskrivna lösningsapproacherna att utvärderas med avseende på lämplighet för det givna problemet.

En av de diskuterade optimeringsstrategierna (Branch & Bound) garanterar att en globalt optimal lösning hittas, men metoden är väldigt tidsödande. Vissa av de andra (lokalsökning, multistart och giriga algoritmer) har en tendens att fastna i lokala optima. De senare nämnda metoderna kan direkt uteslutas p.g.a. att de snabbbt fastnar i lokala optima, samt inte kan ge några garantier på lösningens kvalitet. Branch & Bound däremot är inte lika lätt att avfärda. Nackdelen med Branch & Bound är att den ställer krav på modellen som innebär att modellen måste approximeras. Den nuvarande modellen är relativt exakt, d.v.s. den beskriver problemet relativt bra. Men tyvärr innebär detta att funktionerna som är involverade är för komplexa för denna lösningsapproach. De är t.ex. både okontinuerliga och olinjära.
Utifrån denna situation finns det två möjliga val. Välja att använda den optimerande (exakta) metoden, eller välja en icke-optimerande metod. Väljs den optimerande metoden måste modellen förenklas för att kunna tillämpa metoden. Då måste t.ex. olinjära funktioner approximeras med linjära, och på så sätt kunna lösa optimeringsproblemet och få en exakt lösning som dessutom är optimal m.a.p. modellen. Men notera att denna exakta lösning är en lösning till en förenklad modell, och inte till det riktiga problemet. Väljs en icke-optimerande metod kan modellen behållas som den är. En lösning till denna modell kan erhållas som troligtvis ligger nära optimum, men som inte kan ge några garantier för hur nära den ligger. De två approacherna kan sammanfattas på följande sätt:

Den första använder en approximativ modell, Modell_a, av ett problem, och hittar sen en exakt lösning Lösning_e till den approximativa modellen:
Problem ⇒ Modell_a ⇒ Lösning_e(Modell_a)

Den andra approachen använder en exakt (precis) modell Modell_e till problemet, och hittar en approximativ lösning Lösning_a:
Problem ⇒ Modell_e ⇒ Lösning_a(Modell_e)

Med stöd av ovanstående resonemang kan Branch & Bound metoden uteslutas då den enbart kan ge en lösning till en approximativ modell. De heuristiska metoderna kan här anses vara överlägsna den optimerande med avseende på hur användandet av dessa påverkar modellen, dessutom har dessa med viss säkerhet kortare exekveringstid än Branch & Bound. Bivillkoren och målfunktionen behöver inte antas vara linjära för heuristikerna, vilket gör det möjligt att enklare modellera problemet mer exakt.

Samtliga av de utvärderade heuristikerna, Simulated Annealing, Tabusökning, Genetis-

ka algoritmer och Artificiella neurala nät, har empiriskt visat sig fungera bra vid kombination med annan lokalsökningsheuristik. Vilket innebär att det finns goda möjligheter att hitta sätt att snabba upp lösningsförloppet.

För SA, TS och GA behöver funktionerna inte vara linjära eller kontinuerliga vilket gör dem mest lämpade. Detta utesluter ANN eftersom approximationer i optimeringsmodellen inte kommer tolereras.

Effektivitetsmässigt sett är de tre återstående metoderna, SA, TS och GA i princip likvärdiga. Komplexiteten hos de tre metoderna är i den nämnda ordningen, med minst komplexitet först. Simulated annealing har minst antal parametrar som måste anpassas till modellen, den är även enklast att implementera. Vad gäller konvergens är har dock SA det bästa matematiska beviset (om ej helt vattentätt), på att den i längden kommer konvergera mot en optimal lösning. GA kan uteslutas då den är den mest komplexa av de tre återstående metoderna.

En jämförelse mellan SA och TS visar att de skiljer sig på det sätt de försöker undkomma lokala optima. TS försöker vanligtvis bara gå uppför när den fastnat i ett lokalt optima, medan SA kan gå uppför närsomhelst. Här har TS en fördel gentemot SA eftersom det finns en klar risk att SA inte utforskar varje lokala optima tillräckligt noga. Detta kan dock relativt enkelt åtgärdas genom att kombinera SA med en eller flera lokalsökningsheuristiker som söker sig ner till ”botten” av det lokala optimat. SA är dessutom intuitivt enklare att ha att göra med gå den ska kombineras med andra metoder.

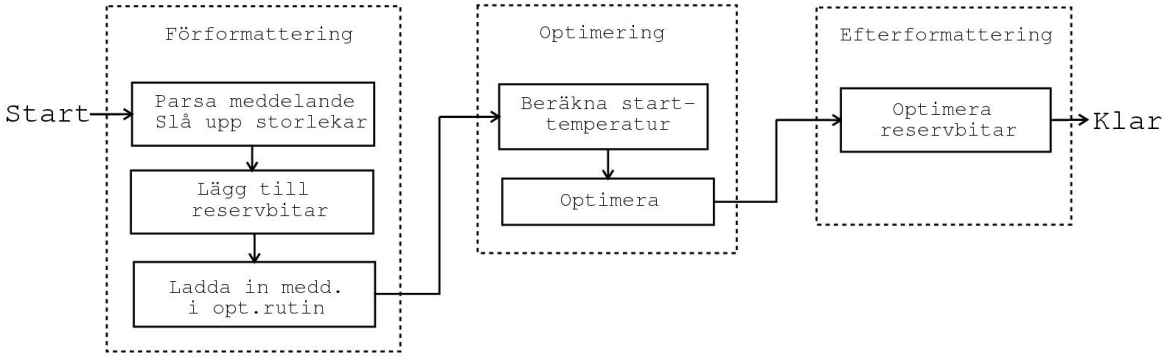
Den approach som kommer väljas är Simulated annealing. För att försöka snabba upp denna kommer den att kombineras med en eller flera ’skräddarsydda’ heuristiker.

8 Implementation

I denna sektion beskrivs implementationen av verktyget, vilka val som gjordes samt grundstrukturen hos verktyget. Samtliga val vid implementationen av optimeringsmetoden Simulated Annealing redogörs och motiveras.

8.1 Verktygets faser

Hanteringen av meddelandet är uppdelat i tre olika faser: förformattering, optimering samt efterformattering. En översiktlig bild av vad som hanteras i de olika faserna visas i figur 6.



Figur 6: Flödesschema över faserna i verktyget.

Den första fasen, förformatteringsfasen, syftar till att göra allting redo innan meddelandet laddas in i själva optimeringsrutinen. Under optimeringsfasen kommer optimeringsalgoritmen att startas och när denna är färdig går verktyget in i efterformatteringsfasen där heuristiska metoder sköter efteroptimering av meddelandet. Den sista fasen hanterar delar av optimeringen som flyttats bort från optimeringsfasen av effektivitetsskäl. T.ex. är det ingen mening med att optimera reserverna innan den slutliga strukturen hos meddelandet är känd.

Om meddelandet är nästlat kan optimeringen inte på förhand avbrytas för att erhålla en lösning då vissa delar av meddelandet inte är hanterade ännu. En sådan lösning kommer inte hålla mycket bättre kvalitet än det ursprungliga meddelandet som ska optimeras. Den lösningen kommer slopas[18]. Dock, om meddelandet är enkelt och om

[18]Att implementera denna feature ansågs som i stort sett meningslös då den bara hade tagit tid från processen att förfina själva optimeringen.

förformatteringsfasen passerats kan optimeringen när som helst avbrytas och en lösning erhållas. Orsaken till att denna lösning kan användas är att hela meddelandet hunnit optimerats (dock ej till optimum).

Vad som sker i de olika faserna beskrivs mer ingående här nedan.

Förformattering
Först kommer en parser att gå igenom meddelandet och slå upp samtliga storlekar för alla typer via en mappningsfil. Uppslagningen av typer sker förnärvarande enbart via mappningsfilen som mappar typer till storlekar[19]. Den ser även till att kontrollera att meddelandet är syntaktiskt korrekt. Parsern är skriven i perl för att få denna hantering så flexibel som möjligt, det ska vara så enkelt som möjligt att modifiera denna hantering vid vidare utveckling. När parsern är färdig kommer all information skrivas ner i ett visst format till en temporär fil[20] Efter detta laddar verktyget[21] in filen i en dynamisk trädstruktur, varefter alla storlekar på substrukturer beräknas och uppdateras i trädet.

Storleken på varje bussmeddelande fylls upp till att vara en jämn multipel av 32-bitars ord. Utfyllnaden som läggs till är s.k. reservbitar. Användaren kan även välja att lägga till flera reserver, men endast i steg om 16 bitar. Då reservbitar ska läggas till i substrukturer kommer en minimal Simulated annealing optimering utföras på varje substruktur för att få en känsla för dess struktur. Med hjälp av resultatet från denna optimering, estimeras antalet reservbitar som behöver läggas till för att förbättra framtida optimering av denna struktur. Vid estimeringen tas hänsyn till substrukturens storlek, totala antalet reservbitar som ska läggas till bussmeddelandet, ungefärliga värden på antalet 16/32-bitars gränser (betecknas n16/n32) som substrukturen bryter efter testoptimeringen samt om den substruktur som kontrolleras är den sista i meddelandet eller ej.

Algoritmen som bestämmer hur många spares som ska läggas till i en substruktur fungerar enligt algoritmbeskrivningen nedan. De förkortningar som används i algoritmen är:
$dist$ = avståndet till nästa 32-bitars gräns från slutet av denna struktur. Används för att storleken på strukturen ej får gå över en gräns. Om reserver läggs till så storleken går över en gränsstorlek, kommer den strukturen alltid att bryta en till gräns.
num_subs = antal substrukturer kvar att hantera efter denna struktur.
$left$ = totalt antal reservbitar kvar att dela ut.

[19] I framtiden kommer dessa data tas direkt ifrån en databas i systemet.

[20] Av säkerhetsskäl skapas denna fil under /tmp av systemet eftersom verktyget inte tillåts att skriva över eller ta bort några filer i systemet.

[21] Verktyget (inklusive optimeringsrutinen) är skrivet i C.

Algoritm 4: Utdelning av reservbitar.

1 Beräkna $dist$.

2 Om enbart 16-bitars gränser bryts.

- **2.1** Om ($dist >= 4$) och $left >= 3 \cdot num\_subs$. Dela ut 4 bitar.
- **2.2** Om ($dist >= 3$) och $left >= 3 \cdot num\_subs$. Dela ut 3 bitar.
- **2.3** Om ($dist >= 2$) och $left >= 2 \cdot num\_subs$. Dela ut 2 bitar.
- **2.4** Om ($dist >= 1$) och $left >= num\_subs$. Dela ut 1 bit.

3 Om 32-bitars gränser bryts.

- **3.1** Om $dist \geq n32 \cdot 8$ och $left > 12 \cdot n32$ och $left \geq 3 \cdot num\_subs$. Dela ut $n32 \cdot 8$ bitar.
- **3.2** Annars om $dist \geq n32 \cdot 7$ och $left > 10 \cdot n32$ och $left \geq 3 \cdot num\_subs$. Dela ut $n32 \cdot 7$ bitar.
- **3.3** Annars om $dist \geq n32 \cdot 6$ och $left > 7 \cdot n32$. Dela ut $n32 \cdot 6$ bitar.
- **3.4** Annars om $dist \geq n32 \cdot 5$ och $left > 6 \cdot n32$. Dela ut $n32 \cdot 5$ bitar.
- **3.5** Annars om $dist \geq n32 \cdot 4$ och $left > 5 \cdot n32$. Dela ut $n32 \cdot 4$ bitar.
- **3.6** Annars om $dist \geq n32 \cdot 3$ och $left > 4 \cdot n32$. Dela ut $n32 \cdot 3$ bitar.
- **3.7** Annars om $dist \geq n32 \cdot 2$ och $left > 3 \cdot n32$. Dela ut $n32 \cdot 2$ bitar.
- **3.8** Annars om $dist \geq n32$ och $left > 2 \cdot n32$. Dela ut $n32$ bitar.
- **3.9** Annars om $dist$ och $left > dist$. Dela ut $dist$ bitar.
- **3.10** Annars om det finns reservbitar kvar, dela ut $left$ bitar.
- **3.11** Annars, dela inte ut några reservbitar.

För substrukturer mindre än 16 bitar adderas inga reserver eftersom dessa redan får plats inuti den minsta ordstorleken. Det ökar bara på komplexiteten att lägga till mer data i strukturen. För strukturer som är en multipel av 16/32 bitar kommer inte reserver läggas till, då det bara ger ännu mer problem. Strukturen passar redan precis in i ett ord, en bit till innebär att en till gräns kommer att brytas. Parametrar som X i uttryck som $X \cdot n32$ har bestämts empiriskt genom tester. Användaren kan även välja att lägga till flera reservbitar än upp till närmaste jämna ordgräns, i detta fall kommer en liknande algoritm användas.

Det är inte ett helt trivialt problem att lägga till reservbitar i form av typer, eftersom hänsyn måste tas till hur många bitar som ska läggas till samt hur stora varje reservtyp ska vara. Används enbart reserver av en bits storlekar växer antalet typer onödigt mycket, vilket snabbt ökar beräkningskomplexiteten. Desto större reservtyper som används desto mer problem vid utplacering av de andra typerna, eftersom större

typer blir svårare att placera ut på ett bra sätt. Det finns alltså ett trade-off förhållande mellan storleken på reserverna och antalet reservbitar.[22]

När reserver ska delas ut används följande algoritm, denna är i grunden väldigt enkel men tas ändå med här för att redogöra för hur reserver läggs till i verktyget.

Algoritm 5: Fördelning av reservtypers storlekar.

1 Så länge det finns reservbitar kvar att dela ut.
- **1.1** Om antalet bitar som ska adderas > 28.
  - **1.1.1** Lägg till en reserv av storleken 6.
- **1.2** Om antalet bitar som ska adderas > 20.
  - **1.2.1** Lägg till en reserv av storleken 4.
- **1.3** Om antalet bitar som ska adderas > 16.
  - **1.3.1** Lägg till en reserv av storleken 3.
- **1.4** Om antalet bitar som ska adderas > 8.
  - **1.4.1** Lägg till en reserv av storleken 2.
- **1.5** Annars
  - **1.5.1** Lägg till en reserv av storleken 1.

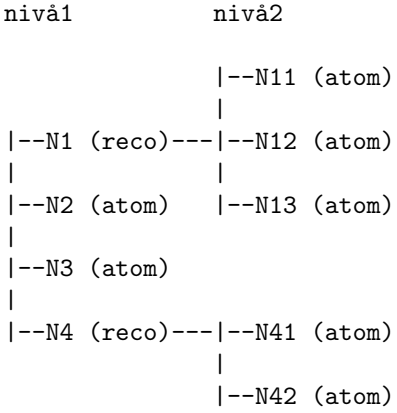
De storlekar som valts för olika antal reservbitar som ska läggas till har tagits fram genom omfattande tester. Den konfiguration som används nu är den som verkar ge bäst resultat med tanke på rimlig beräkningskomplexitet. När samtliga reservbitar lagts till måste samtliga storlekar för substrukturerna uppdateras. För de olika valen i cases kommer dessa storlekar jämnas ut med extra reservbitar till det största valets storlek.

När de andra momenten i förformatteringen är klara kommer sedan den totala exekveringstiden som användaren valt för optimeringen att delas upp mellan substrukturerna. Meddelandet är sedan redo för att laddas in i optimeringsrutinen.

[22] Självklart förekommer det ett antal specialfall där det är mycket bättre att använda större reserver. T.ex. om en struktur innehåller många 10 bitars typer kan det vara mer fördelaktigt att använda på t.ex. 6 bitars reserver, än många 1 eller 2 bitars reserver.

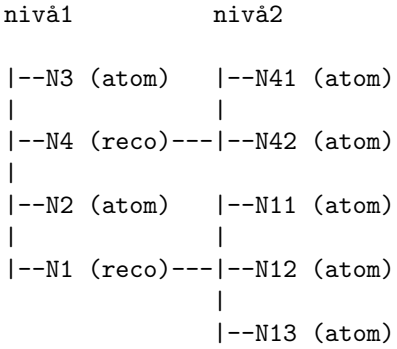
Optimering

I början av Simulated annealing rutinen kommer en starttemperatur för varje struktur beräknas. För nästlade meddelanden startar optimeringen vid nivå 1, rotnivån, och optimerar utplaceringen av noder (löv och delträd) på denna nivå. Under optimeringen ses alla noder på nivån som enbart löv, ingen hänsyn tas till innehållet i noderna. I meddelandet i figur 7 kommer delträden N1 och N4 i detta steg ses som löv. När samtliga löv på denna nivå är optimalt utplacerade låses deras positioner fast, vilket innebär att de ej får flyttas hädanefter.



Figur 7: Exempel på optimering av ett litet bussmeddelande.

När noderna i en nivå låses fast markeras utplaceringen av dessa som optimerad. Opptimeringen av noderna på nivå 1 gav en placering som visas i figur 8. Notera att delträden N1 och N4 nu ändrat platser. Algoritmen börjar sedan leta efter icke-optimerade delträd på nuvarande nivå.



Figur 8: Meddelandets utseende efter optimeringen av noderna på nivå 1.

Det närmaste icke-optimerade delträdet N4 hittas, optimeras och sedan låses dess innehåll på nivå 2 fast. Delträdet N4 innehåller inga fler delträd, varvid algoritmen backar

tillbaka till föräldranodens nivå och letar vidare efter nästa icke-optimerade delträd. Delträdet N1 hittas och optimeras. Det innehåller inga fler delträd och agoritmen backar till föräldranodens nivå och letar vidare. Inga fler icke-optimerade delträd existerar nu på nivå 1 vilket innebär att optimeringen av hela trädstrukturen är färdig. Algoritmen för optimeringen av strukturerna i trädet är följande.

Algoritm 6: Optimering av nästlat bussmeddelande.

1 Starta på första nivån i trädet.

2 Betrakta samtliga noder (delträd och löv) på nuvarande nivå som löv och optimera utplaceringen av dessa.

3 Lås fast positionerna för dessa löv (deras positioner får hädanefter ej ändras). Markera utplaceringen av alla noder på denna nivå som optimerade.

4 Börja med första noden i nuvarande nivå.

- **4.1** Om nuvarande nod är roten till ett icke-optimerat delträd.
  - **4.1.1** Gå in i delträdet och gå till steg 2.
- **4.2** Annars
  - **4.2.1** Om vi är i sista noden i nivån.
    - **4.2.1.1** Om vi är i nivå 1.
      - **4.2.1.1.1** Optimeringen av hela bussmeddelandet är klar.
    - **4.2.1.2** Annars
      - **4.2.1.2.1** Backa till föräldranoden.
  - **4.2.2** Annars
    - **4.2.2.1** Gå till nästa nod i nivån.

5 Gå till steg 4.

En avvikelse kommer att ske om en nod är ett case som innehåller olika val. Där kommer varje val av caset att optimeras var för sig, och oberoende av varandra eftersom enbart ett av valen ska vara i det meddelande som skickas. därefter markeras noden som optimerad.

Optimeringen av varje struktur sker med en modifierad Simulated Annealing heuristik som är kombinerad med en lokalsökningsheuristik. Algoritmen för denna optimeringsmetod är följande.

Algoritm 7: Simulated annealing kombinerad med lokalsökningsheuristik.

1 Beräkna starttemperatur.

2 Om lösningen trivial, ta fram denna och avbryt.

3 Beräkna startkostnad för nuvarande lösning.

4 Öka antalet iterationer med ett.

5 Gör

- **5.1** Gör
  - **5.1.1** Beräkna kostnad för ny lösning.
  - **5.1.2** Om kostnad för ny lösning är mindre än nuvarande.
    - **5.1.2.1** Spara denna lösning.
    - **5.1.2.2** Försök korrigera denna lösning med lokalsökningsheuristik.
  - **5.1.3** Annars, om kostnaden är större än gamla lösningen.
    - **5.1.3.1** Acceptera sämre lösning med viss sannolikhet.
    - **5.1.3.2** Uppdatera nuvarande kostnad till nya kostnaden.
  - **5.1.4** Om nuvarande kostnad är noll har ptimal lösning funnits, trigga avbrottvillkor.
  - **5.1.5** Om maximal exekveringstid uppnåtts, och ingen extra tid finns.
    - **5.1.5.1** Trigga avbrottsvillkor.
  - **5.1.6** Generera ny lösning ur omgivningen.
  - **5.1.7** Så länge avbrottsvillkor och max antal iterationer ej uppnåtts.
- **5.2** Minska temperaturen.
- **5.3** Öka Markovkedjans längd (ökar maxiter).
- **5.4** Om föregående MAX_SAME_MARKOV_CHAINS st. temperatursteg inte förbättrat lösningen.
  - **5.4.1** Trigga avbrottsvillkor.
- **5.5** Så länge avbrottsvillkoret ej triggats.

6 Om max exekveringstid ej uppnåtts, lägg resterande tid i allmänna tidspotten.

7 Om allvarligare optimeringsregler brutits, uppmärksamma användaren m.h.a. statusmeddelande.

Notera att innan Simulated annealing startas sker en kontroll om optimeringen är trivial eller inte. Kriterier för att optimeringen ska räknas som trivial är någon av följande punkter:

- Alla typer är av samma storlek (ingen mening att placera om dom).

- Antalet typer är mindre än MAX_BRUTE_FORCE.

Då antalet typer är mindre än MAX_BRUTE_FORCE kommer den bästa möjliga lösningen tas fram genom brute-force, vilket innebär att en globalt optimerande metod testar alla möjliga lösningar. Gränsen för när detta sker ligger förnärvarande på 8 typer (40 320 möjliga lösningar).

När optimering av en substruktur tar kortare tid än dess maximala tillåtna exekveringstid, kommer resterande tid att läggas i en allmän tidspott. Ur denna tidspott kan sedan tid tas för optimeringar vars maxtid tagit slut. Denna mekanism används för att balansera mellan optimeringar av olika substrukturer, eftersom alla tar olika lång tid att hantera. Vissa kan gå väldigt snabbt, medan andra kan ta lång tid på sig.
Vid optimering av substrukturer måste dessutom en offset från starten av meddelandet användas, eftersom gränserna räknas med avseende på dessa och inte från starten av substrukturen. Den lokalsökningsheuristik som Simulated annealing kombinerats med, används för att försöka korrigera varje ny lösning som SA producerar som är den bästa som hittills hittats. Orsaken till att denna heuristik inte används då t.ex. en sämre lösning accepteras, är att risken är stor att den då går tillbaka till den tidigare (bättre) lösningen. Det lokalsökningsheuristiken praktiskt gör är att identifiera de typer som orsakar problem, samt de typer som inte orsakar problem. Sedan försöker metoden byta plats på dessa för att minska den totala kostnaden. Algoritmen för lokalsökningsheuristiken är följande.

Algoritm 8: Lokalsökningsheuristik.

1 För varje typ i strukturen. (Leta efter typer som orsakar problem samt typer som kan flyttas för att fixa problem med).

1.1 Om typen korsar en 16/32-bitars gräns samt ej är en reserv.

1.1.1 Markera att typen orsakar problem (en kostnad), och beräkna det avstånd (i bitar) som den måste flyttas för att eliminera problemet.

1.2 Om typen är mindre än 16 bitar och inte orsakar några problem.

1.2.1 Markera att typen går att flytta utan att orsaka problem.

2 För varje typ i strukturen. (För varje problemtyp, försök hitta flyttbara typer som de kan fixas med hjälp av).

2.1 Om typen orsakar problem

2.2 Gå igenom efterföljande typer och kolla hur många som kan flyttas.

2.3 Om tillräckligt många typer kan flyttas så att problemet löses helt, byt plats på dessa med problemtypen.

2.4 Annars, om storleken inte är tillräckligt stor. Flytta dom ändå om deras sammanlagda storlek är större än hälften av avståndet för problemtypen.

2.5 Annars, flytta inga typer, returnera.

3 Om nya korrigerade lösningen är bättre än tidigare lösning, sätt den till att vara nuvarande.

Notera att algoritmen enbart letar efter typer att flytta efter den aktuella problemtypen. Orsaken till det är att om typer före dom flyttas kommer gränserna hela tiden att förändras för problemtyper vars positioner redan korrigerats. Om inte alla efterföljande typer kan flyttas sker enbart flyttning om deras sammanlagda storlek är större än halva det avstånd som problemtypen måste flyttas. Värdet 0.5 är framtestat som lämpligt värde, vid mindre värden på 0.5 kan lösningarna t.o.m. bli sämre under vissa förutsättningar.

Efterformattering

Efter optimeringsfasen hanteras den slutgiltiga placeringen av reservtyper. Optimeringen av reservtyper har flyttats till efterformatteringsfasen av effektivitetsskäl, det minskar belastningen vid beräkning av kostnader. Inga kostnader beräknas för reservtyper som korsar ordgränser under optimeringsfasen. Inga kostnader beräknas heller för reservtyper som inte ligger längst fram i ord. Detta hanteras här istället eftersom det bara behöver hanteras då allt annat redan är optimalt utplacerat[23].

Först körs en heuristik `cluster_spares` som delar upp de reserver som ligger över en 16 bitars gräns. Här hanteras enbart 16-bitars gränser då dessa även inbegriper 32 bitars gränser. Därefter klustras samtliga reservbitar inom varje ord ihop till en enda reservtyp.

Sedan kommer ytterligare en heuristik `optimize_spares` att köras som flyttar fram varje reservtyp till starten inom det 16-bitars ord det ligger inom. Att flytta omkring typerna inom samma ord kommer inte att förändra något, eller se till att någon regel kan komma att brytas.

Resultatet skrivs sedan ut till en fil.

[23]Det finns ingen mening med att optimera reservbitar innan den slutliga strukturen hos meddelandet är känd

8.1.1 Beräkning av kostnader

För val av kostnader har en generell regel tillämpats, samtliga kostnader gällande 32 bitars gränser är satta så de är mycket högre än de för 16 bitars gränser. En tumregel är att runt 10st 16 bitars gränser måste brytas för att väga upp mot brytandet av en 32 bitars gräns. I tabell 6 redovisas de val av kostnader som gjorts.

Kostnadsfaktorer/vikter	**Värde**
BREAK_16_BORDER_COST	32
BREAK_32_BORDER_COST	340
DIFF_FROM_16_BORDER_WEIGHT	1
DIFF_FROM_16_BORDER_COST_OFFSET	16
DIFF_FROM_32_BORDER_WEIGHT	10
DIFF_FROM_32_BORDER_COST_OFFSET	160
UPDC_COST_FACTOR	1

Tabell 6: Använda värden för kostnader och vikter.

Orsaken till att BREAK_32_BORDER_COST valdes som 340 istället för 320 var p.g.a. att för värden under detta kan det vara billigare att bryta två 32-gänser under vissa omständigheter.

Den algoritm beräknar kostnader relaterade till 16 bitars gränsen visas i algoritm 9. Om typen är större än 16 bitar kommer den alltid att bryta en sådan gräns. Nästa gräns blir en 32 bitars gräns vilket till viss del kommer sammanbinda resultatet av denna algoritm med algoritmen för beräkning av brytning av 32 bitars gränser. De variabler som används i denna algoritm är:
dist - avstånd som typen går över den gräns den egentligen bör ligga emot för optimal placering.

Algoritm 9: Kostnad för 16-bitars gräns.

1 Om typen är mindre än 16 bitar.

  1.1 Om typen bryter en 16-bitars gräns.

    1.1.1 Sätt kostnad till BREAK_16_BORDER_COST och returnera.

2 Annars

  2.1 Sätt kostnad till noll och returnera.

3 Annars (om typen är större än 16 bitar).

  3.1 Om typen bryter 32 bitars gränsen som kommer efter 16-bitars gränsen.

3.1.1 Beräkna *dist*.
3.1.2 Sätt kostnad till:
dist·DIFF_FROM_16_BORDER_WEIGHT+
DIFF_FROM_16_BORDER_COST_OFFSET.

Den algoritm som används för hantering av kostnader relaterade till 32-gränsen visas i algoritm 10. De variabler som används i denna algoritm är:
dist - avstånd som typen går över den gräns den egentligen bör ligga emot för optimal placering.
num_borders - antalet 32-bitars gränser som typen bryter.

Algoritm 10: Kostnad för 32-bitars gräns.

1 Om typen är mindre än 32 bitar.
1.1 Om typen bryter en 32-bitars gräns.
1.1.1 Sätt kostnad till BREAK_32_BORDER_COST och returnera.
1.2 Annars
1.2.1 Sätt kostnad till noll och returnera.
2 Annars (om typen är större än 32 bitar).
2.1 Om typen bryter 32 bitars gräns.
2.1.1 Beräkna *num_borders*.
2.1.2 Sätt kostnad till:
dist·DIFF_FROM_32_BORDER_WEIGHT+
num_borders· DIFF_FROM_32_BORDER_COST_OFFSET.

Heuristiken som beräknar kostnaden för placeringen av typ för uppdatering av räknare är rättfram. Den mäter bara avståndet till slutet av bussmeddelandet och multiplicerar denna med en vikt för att få kostnaden.

8.2 Kylschema till optimeringsalgoritmen

Generellt sett, vid val av regler och ekvationer till kylschemat, har viss hänsyn tagits till om dessa innehållit parametrar m.m. som måste tas fram genom empiriskt testande. Vissa regler och ekvationer har delvis valts ut för att minska andelen empiriska tester som måste utföras för att anpassa alla dessa till problemet.[24]

[24]Med tanke på tidsaspekten för detta arbete var det inte rimligt att testa fram bra värden på ett för stort antal parametrar.

Vid val av kylschema stod valet länge mellan att använda enkla eller utarbetade, men slutligen valdes de enklare eftersom de är empiriskt utvecklade. Nackdelen med att använda ett teoretiskt utarbetat kylschema är att det finns ett väldigt stort antal av dom, och det är väldigt svårt att veta vilken som passar bäst för det givna problemet. Dessutom innebär inte de teoretiskt utarbetade att de automatiskt ger bra resultat. Det ansågs därför vara säkrare att experimentera med metoder inom enklare kylscheman, eftersom dessa empiriskt sett visat sig fungera bra. Dessutom fanns det många fler studier av dessa kylscheman som styrkte att de givit bra resultat.

Starttemperatur
För att beräkna starttemperaturen valdes ekvation 11. I enlighet med bland annat Aarts och Laarhoven's [6] diskussion angående att temperaturen bör vara såpass hög att de flesta sämre lösningar ska accepteras, valdes en acceptanssannolikhet på $\chi_0 = 0.8$. Vilket även är samma som Kirkpatrick *et al.* [16] använde sig av. Detta val innebär att 80% av alla lösningar som är sämre än nuvarande kommer att accepteras i det första temperatursteget. Orsaken till att inte en ännu högre acceptanssannolikhet valdes var att det tar såpass lång tid att sänka temperaturen från en sannolikhet på 100% till 80%, medan den slumpmässiga rörelsen mellan olika lösningar i lösningsrummet i princip är lika stor ändå. Det ansågs bara ge ett stort tillägg i exekveringstid att ha ännu högre sannolikhet.

Sluttemperatur
Valet föll på alternativet att avbryta optimeringen efter att ett visst antal Markovkedjor genererats utan någon förbättring av lösningen. Bland annat valdes denna metod eftersom den tar hänsyn till om lösningen förbättras med tiden eller inte. Detta sker m.h.a. ett fönster bakåt där några av de föregående Markovkedjorna ses. Att på förhand bestämma hur många temperatursteg en algoritm ska köra tar inte hänsyn till vilken historia lösningsproceduren haft. T.ex. kan då optimeringen avbrytas av det kriteriet, trots att den nyligen gjort flera stora framsteg. Genom att titta på de senaste markovkedjorna erhålls åtminstone lite insikt om det är läge att avbryta eller inte, d.v.s. om det förekommit några framsteg den senaste tiden eller ej.

Avbrottvillkoret träder i kraft då 6 st markovkedjor i rad ej förbättrat lösningen. Antalet kan ökas, men det innebär att alla körningar kommer ta längre tid. Orsaken till att just 6 st Markovkedjor måste köras innan den avbryter är p.g.a. att det antalet verkade ge en bra avvägning mellan exekveringstid och lösningskvalitet.[25] Optimeringen kommer såklart alltid att avbrytas då en lösning med kostnaden 0 hittas.

[25]Oftat hittades ingen bra lösning alls efter detta antal (eller färre som 4-5 st). Vid en del tillfällen hittades bättre lösningar, men inte förrän efter många markovkedjor ($> 10 - 15$ st) passerats.

Orsaken till att Lundy och Mees metod (ekvation 12) ej valdes för att beräkna när systemet passerat fryspunkten, var bland annat p.g.a. att metoden kan innebära svårigheter för problem där ingen tillräckligt bra lösning kan hittas, som ser till att optimeringen avbryts. Då kommer villkoret troligtvis inte att kunna avbryta förrän temperaturen minskat till noll. Vilket innebär väldigt långa exekveringstider (eftersom vissa meddelanden inte går att optimera till fullo, d.v.s de som oundvikligen alltid kommer bryta någon regel). Ännu en orsak till att den metoden inte valdes var att den artikel där Lundy och Mees härleder och beskriver sin metod inte gick att få tag i.

Längd på Markovkedjorna

Längden på Markovkedjorna valdes som $L = m \cdot n$, där n är antalet typer i problemet och m är storleken på omgivningen för en lösning. Parametern m valdes till $m = 40000$ vilket motsvarar ett meddelande bestående av 283 typer. Storleken på omgivningen till en lösning med n st typer kan beräknas[26] som $\sum_{i=1}^{n}(n-i)$. För $n = 283$ fås antalet möjliga lösningar i omgivningen till 39903. Just siffran 283 kommer ifrån antagadet att ett bussmeddelande (med hänsyn tagen till en väldigt stor standardavvikelse) borde rimligtvis högst bestå av hälften så många typer som det maximala antalet booleanska typer (512 st), d.v.s. 256. För att få mer marginal till detta antagande ökades siffran med med 10% vilket gav 282 st, och avrundning för storleken på omgivningen till detta antal, 39620 till 40000 gav $n = 283$. Detta är såklart en avvägning då det kan finnas meddelanden med fler typer, men dessa är troligtvis väldigt sällsynta. Det ansågs mest lämpligt att anpassa modellen efter de vanligast förekommande meddelandetyperna. Att anpassa modellen så den blir mest lämplig för meddelanden med mindre än 283 typer kommer täcka in de allra flesta förekommande fallen (troligtvis alla). Notera att valet av längd innebär att antalet iterationer per temperaturnivå alltid kommer vara mer än kvadraten av antalet typer i meddelandet. För 512 typer kommer längden att vara $L_k \geq 512 * 40000 = 20480000$, medan $512^2 = 262144$, för 256 typer motsvarar antalet iterationer ungefär antalet typer i kubik.

En jämförelse med teorin för längden på Markovkedjor i avsnitt 6.4.2, visar att den föreslagna regeln modifierats litet genom att för varje temperaturminskning öka på längden av kedjan med ungefär 1%. Detta ansågs behövas då varje temperaturminskning innebär att färre lösningar som kan förbättra nuvarande lösning kommer att hittas. Och processen får därmed lite större chans att hitta en bättre lösning innan den terminerar. Dessutom kommer acceptanssannolikheten att minska vilket enligt teorin ger längre Markovkedjor. Att öka längden på kedjorna en aning efter varje temperatursteg är en anpassning till detta faktum.

[26]För n st typer och endast ett enda byte mellan två typer får göras, kommer antalet vid första gången vara n. När typerna bytt plats en gång kan en av dessa fixeras, varvid det blir $n-1$ typer kvar att försöka med vid nästa byte osv. I detta fall fås antalet lösningar i omgivningen till 283+282+...+1 = 39903.

Minskning av temperaturparametern
För minskning av temperaturparametern valdes den geometriska regeln i ekvation 13 som minskningsregel. Orsaken till att den mer avancerade metoden (se ekvation 14) inte valdes, är att den ger en klart långsammare kylning. Dessutom finns inga rekommendationer på lämpligt värde på Δ utan det måste experimenteras fram. Den valda metoden är enkel att förstå och det är lätt att inse med vilken hastighet (och med hur stora steg) temperaturen kommer sänkas. Den är även enkel att använda för att experimentera med olika hastigheter. Även för denna metod finns en okänd parameter som måste justeras för det givna problemet, men här fanns goda rekommendationer på lämpliga värden att återfinna i litteraturen. Valet baserades även på resultat av teori och empiri för temperatursänkning i kylscheman, som indikerar att hastigheten med vilken temperaturen sänks med är viktigare än sättet den sänks på.

Utförliga empiriska tester av parametern α har utförts för värden mellan 0.8 till 0.99. Det visade sig att ett värde mellan $0.85 - 0.9$ verkar ge bäst resultat för detta problem. Värdet $\alpha = 0.85$ valdes istället för $\alpha = 0.90$ för att hålla exekveringstiden nere.

Omgivning
Som omgivning till nuvarande lösning valdes de lösningar som kan nås genom att byta plats på två typer i meddelandet, d.v.s. med en permutation. Nåbarhetsvillkoret kommer upprätthållas med denna omgivningsfunktion eftersom en godtycklig lösning alltid kan nå en annan lösning med ett godtyckligt antal permutationer. Den funktion som genererar en ny lösning ur omgivningen blir mycket enkel då den bara behöver hantera en enda permutation av två typer i den aktuella lösningen. Denna omgivningsfunktion ger att omgivningen blir relativt liten, för n st typer i blir antalet lösningar i omgivningen $\sum_{i=1}^{n}(n-i)$ st.

Kostnadsfunktion
Den kostnadsfunktion som valts (se ekvation 2 i sektion 3.3) består av ett antal kostnader för utplacering av datatyperna. Kostnaderna kan ses som en sorts strafftermer för hur mycket problem typens placeringen innebär. En placering som är helt ok har kostnaden 0. Vilket leder processen mot lokala optimum genom att de bästa placeringarna får lägst kostnader.

8.3 Grafiskt gränssnitt

Till verktyget implementerades även ett grafiskt användargränssnitt. Detta beskrivs utförligt i användarmanualen, i appendix C.

Designmålet med gränssnittet var att göra det så enkelt att använda som möjligt genom att ha en simplistisk design. Samtliga inställningar som ska göras är på förhand satta till typiska default värden. Dessa värden behöver enbart ändras för optimeringar som kräver mer specialbehandling. Det enda som behöver fyllas i är vilken fil som bussmeddelandet ligger i. Fil som resultatet ska hamna i behöver inte fyllas i, eftersom utfilen blir namnet på infilen avslutat med '.tmp'.

Under optimeringen ges kontinuerlig feedback till användaren om vad som händer. T.ex. vart den är i optimeringen, om någon substruktur inte går att optimera helt etc.

8.4 Slumptalsgenerering

För flera av de utvärderade metoderna behövs det genereras slumptal. Begreppet 'slumpmässig' är inte helt trivialt att definiera. Ett bra sätt att klassificera en sekvens av heltal som slumpmässig är att säga att den är slumpmässig endast om den inte har någon kortare beskrivning än själva sekvensen av tal [5]. För att generera slumptal på en dator behövs dock en beskrivning av hur sekvensen ska tas fram, vilket på sätt och vis motsäger föregående klassificering. Slumptalen blir pseudoslumptal. Klassificeringen ger ändå insikt om att sekvensen av tal inte bör upprepas, eftersom den då absolut inte kan anses vara slumpmässig. Fördelarna med att ha en algoritm som genererar sekvenser av pseudoslumptal är att simuleringar kan testas om igen, försöken blir repeterbara. Därmed blir det möjligt att verifiera resultat och även enklare att testa algoritmer och debugga programkod.

Alla slumptalsgeneratorer har inte samma kvalitet. En god slumptalsgenerator bör ha en lång cykellängd och vara statistiskt robust. Det mest använda kvalitetsmåttet på slumptalsgeneratorer är cykellängden, d.v.s. antalet slumptal som genereras innan slumptalsgeneratorn upprepar sig igen. Desto längre cykellängd desto bättre. Slumpgeneratorns statistiska robusthet kan mätas i form av tecken på seriell korrelation för den sekvens som genereras. Korrelation är ett mått på det linjära sambandet mellan två variabler. Med seriell korrelation menas i detta fall att samband existerar mellan observationer av olika punkter i sekvensen av tal. Till exempel är det relativt vanligt med slumptalsgeneratorer som genererar starkt korrelerade sekvenser.

De slumptalsgeneratorer som följer med biblioteksrutiner bör i största mån undvikas, som t.ex. rand() som medföljer ANSI's C-bibliotek. Standarden specificerar att

RAND_MAX är minst 32767, d.v.s. minst så långa sekvenser kommer att genereras. Att tillägga är RAND_MAX just så stor på den arkitektur som verktyget ska exekveras på. En tumregel är att aldrig använda mer än 5% av slumptalsgeneratorns period. Om t.ex. 10^6 st tal ska genereras kommer sekvensen av slumptal att återvinnas över 30 gånger.

Den slumptalsgenerator som valdes till detta verktyg är L'Ecuyer's [7] som kombinerar två olika sekvenser. Sekvenserna har olika perioder, och ger en cykellängd som är den minsta gemensamma multipeln av de två perioderna. Detta ger en cykellängd på runt $2.3 \cdot 10^{18}$. Kombineringen av de två generatorerna bryter upp den seriella korrelationen avsevärt. Denna slumptalsgenerator har goda statistiska egenskaper och väldigt lång period, och anses vara en riktigt bra slumptalsgenerator [14]. Enda nackdelen är att den är långsammare än t.ex. ANSI's rand()-generator. Genom testkörningar med 10^8 anrop har det konstaterats att den medföljande ANSI-C generatorn är c.a. 5 ggr snabbare än denna. För att generera 10^8 slumptal krävde L'Ecuyer's generator i snitt 172 s., medan ANSI's krävde 33 s. Dock anses inte denna extra tidsåtgång vara såpass negativ att det tar bort de positiva effekterna av generatorn[27].

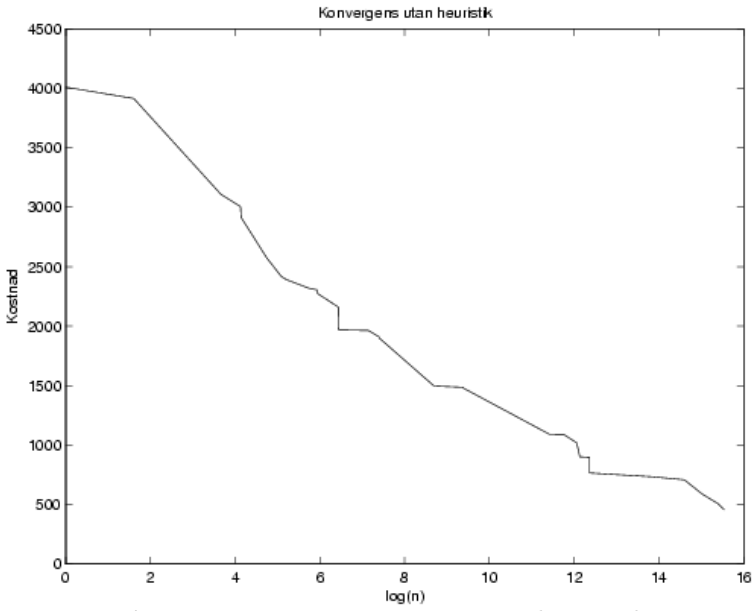
[27] Att tillägga kommer de flesta körningar hamna omkring 10^7 iterationer, eller färre, vilket medför att generatorns påverkan på tiden blir c.a. 18 sekunder extra.

9 Resultat

Det implementerade verktyget utför placering av data i bussmeddelanden med avseende på de krav som ställts för detta. Vilket motsvarar förväntningarna på verktyget.

Verktyget har även testats på de meddelanden som redan var optimerade för hand. Det visade sig att verktyget, i de flesta av fallen, lyckades hitta mer optimala placeringar av datat. Vilket var lyckat. För de flesta meddelanden, även de större meddelandena, hittas en optimal lösning under 20 minuters exekveringstid. Användningen av verktyget kommer att kunna minska den tid som behövs för att hantera optimering av bussmeddelanden avsevärt. Dessutom kommer användningen av detta verktyg innebära att detta arbetsmoment inte kräver lika mycket övertid. Slutligen kan användningen minska den stress som detta arbetsmoment annars framkallat bland de anställda.

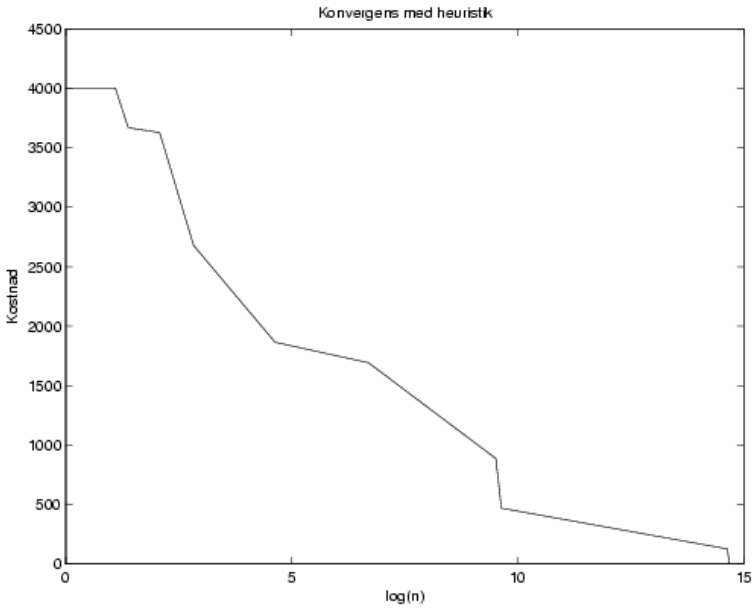
Resultatet av den förbättring av Simulated Annealing metoden som gjordes redovisas i graferna i figur 9 och 10. Förbättringen bestod i att kombinera Simulated Annealing algoritmen med en lokalsökningsheuristik. Datat till graferna i figurerna nedan kommer ifrån en testkörning på ett meddelande som kan anses vara relativt typisk för optimeringsproblemet. Meddelandet var av enkel typ (innehöll inga substrukturer) och bestod av 50 typer. I figur 9 visas först hur konvergensen ser ut då den extra heuristiken inte används.



Figur 9: Konvergens utan extra heuristik.

På y-axeln ses kostnadsfunktionen för utplaceringen av typerna, och på x-axeln visas logaritmen av antalet iterationer. Ett värde på $x \approx 14$ motsvarar ungefär 10 minuters

exekvering. Runt detta värde planar konvergenshastigheten ut för denna optimering, och efter en 1.5 timmars exekvering avbryts optimeringen då den inte hittat någon bättre lösning. Optimering med den kombinerade metoden kan ses i figur 10. Här hittas den optimala lösningen (kostnaden noll) redan efter c.a. 10 minuter.



Figur 10: Konvergens med extra heuristik.

Det ska upprepas att detta är ett relativt typiskt optimeringsfall. Dock hittar SA metoden utan extra heuristik ofta en optimal lösning inom rimlig tid. I de allra flesta fall som testats hittar dock den kombinerade metoden en lösning snabbare än den rena SA metoden.

10 Diskussion och slutsats

Optimeringsmetoden Simulated annealing valdes utifrån ett antal konkurrerande metoder därför att den hade mest lämpliga egenskaper för optimeringsproblemet. Det implementerade verktyget löser problemet med att optimera bussmeddelanden med goda resultat. Det kommer därmed vara till stor nytta för de anställda som sparar in arbetstid på ett annars slitsamt arbetsmoment. Verktyget är även till nytta då det kan optimera redan tidigare optimerade meddelanden, vilket innebär att prestandan hos Gripens databussar kommer kunna utnyttjas bättre.

En viss uppsnabbning av konvergensen jämfört med den ursprungliga Simulated Annealing algoritmen, lyckades även åstadkommas. Förbättringen bestod i att kombinera Simulated Annealing algoritmen med för problemet speciellt implementerad lokalsökningsheuristik.

Huvudorsaken till att detta verktyg behövdes var tidsbesparing. Andra skäl var att minska på ett mycket tidsödande arbetsmoment, att få ett mer konsistent förfarande vid optimeringen, samt att öka bussens prestanda. Det implementerade verktyget kan anses uppfylla dessa krav.

10.1 Begränsningar

Vid stora meddelanden (> 50 element) kan exekveringstiden bli lång, normal tid är runt en halvtimme och uppåt. För vissa meddelanden kan tiden ligga runt en timme eller två för att resultatet ska bli bra. Det finns ingen generell röd tråd mellan vilka meddelanden som orsakar riktigt långa exekveringstiderna. Det enda som gått att identifiera är att det brukar röra sig om meddelanden innehållandes många stora > 32 och många små < 6 typer men inga av mellanstorlek.

11 Vidare arbete

Som vidare arbete föreslås:

1. Att visa resultatet i ett interaktivt GUI där det för hand går att flytta om typerna och se resultatet direkt.
2. Förbättra SA-metoden m.h.a. bättre kylningsschema och förfinade metoder för att ta fram start/slut-temperaturer samt avbrottsvillkor.
3. Ta fram och implementera flera heuristiker liknande den som redan används för att snabba på lösningsförfarandet ännu mer.
4. Förbättra beräkning av kostnader med mer intelligenta beräkningar, t.ex. spara undan vissa beräknade värden m.m. som inte har ändrats sen senast.
5. Utvärdera (kanske m.h.a. statistik eller enbart empiri) sättet som reserver delas ut på samt vidareutveckla det. Konfigurationen av reservers storlekar ger en stor påverkan på optimeringen och även på det slutgiltiga resultatet.
6. Implementera andra optimeringsmetoder och jämföra vilken av dessa som är bäst.
7. Förbättra modellen genom att göra en studie över hur reglernas poängsättning kan förbättras så de matchar verkligheten bättre och därmed ger bättre resultat.

A Tack

Jag vill tacka min handledare och chef på Saab, Peter Uhlin, samt Roger Svensson på Saab för feedback och hjälp under mitt arbete med verktyget.

Jag vill också tacka min handledare Per Lindström vid Umeå Universitet för hjälp med rapporten.

Slutligen vill jag tacka familj och vänner för stöd och hjälp, i synnerhet vill jag tacka min flickvän Linda Wiklund som är en stor inspirationskälla för mig varje dag. Jag vill även passa på att tacka min vän Kristina Lindholm för ovärderlig hjälp med bostad i Linköping.

B Förkortningar och notation

Förkortningar

- ANN = Artificiellt Neuralt nät
- B&B = Branch and Bound
- GA = Genetiska algoritmer
- HP = Heltalsprogrammering
- ILP = Ickelinjärt optimeringsproblem
- IHLP = Ickelinjärt heltals optimeringsproblem
- KO = Kombinatorisk Optimering
- LP = Linjärprogrammering
- SA = Simulated annealing
- TS = Tabusökning

Notation

- Atomär - Odelbar datatyp
- Case - Innehåller ett eller flera val. Endast ett av valen kommer skickas i bussmeddelandet vid sändning. Varje val är en konstruktion som kan innehålla odelbara datatyper och substrukturer i ett bussmeddelande.
- Delträd - Substruktur (record eller case).
- Enkelt bussmeddelande - Bussmeddelande som enbart innehåller datatyper, och inga fler records eller cases.
- Löv - datatyp.
- Nod - ett löv eller ett delträd.
- Nästlat bussmeddelande - Bussmeddelande som innehåller records eller cases.
- Record - Konstruktion som kan innehålla odelbara datatyper och substrukturer i ett bussmeddelande.
- Reserv - Typer bestående av reservbitar som används för att fylla upp meddelandet eller ett case till en viss storlek.
- spare - Samma som Reserv
- Subrecord - record som deklareras inuti ett bussmeddelande.
- Substruktur - Record eller case.
- Struktur - Ofta använd förkortning av substruktur.

C Användarmanual

I denna sektion infogas användarmanualen som beskriver hur verktyget ska installeras och användas.

C.1 Installation

Innan verktyget kan användas måste två miljövariabler sättas i systemet. Dessa är:

`BUSMSGPARSINGSCRIPTPATH` - Sökväg till parsningsscriptet. Måste vara sökvägen till filen, ej enbart till den katalog som filen finns i.

`BUSMSGTYPESIZESFILEPATH` - Sökväg till fil för mappning mellan typer och dess storlekar. Måste vara sökvägen till filen, ej enbart till den katalog som filen finns i.

Systemkrav:
Applikationen är testad och utvecklad på SUN arbetsstationer under Solaris 8.
Mjukvarukraven på applikationen är följande:
Script: Perl4 eller senare.
Optimeringsrutin: ANSI-C samt POSIX C standard bibliotek.
GUI: Gtk+-1.2 eller senare.

C.2 Användning

För att snabbt kunna komma igång med verktyget ges först en mycket kort version för användandet. För den som vill veta mer ges även en längre version.

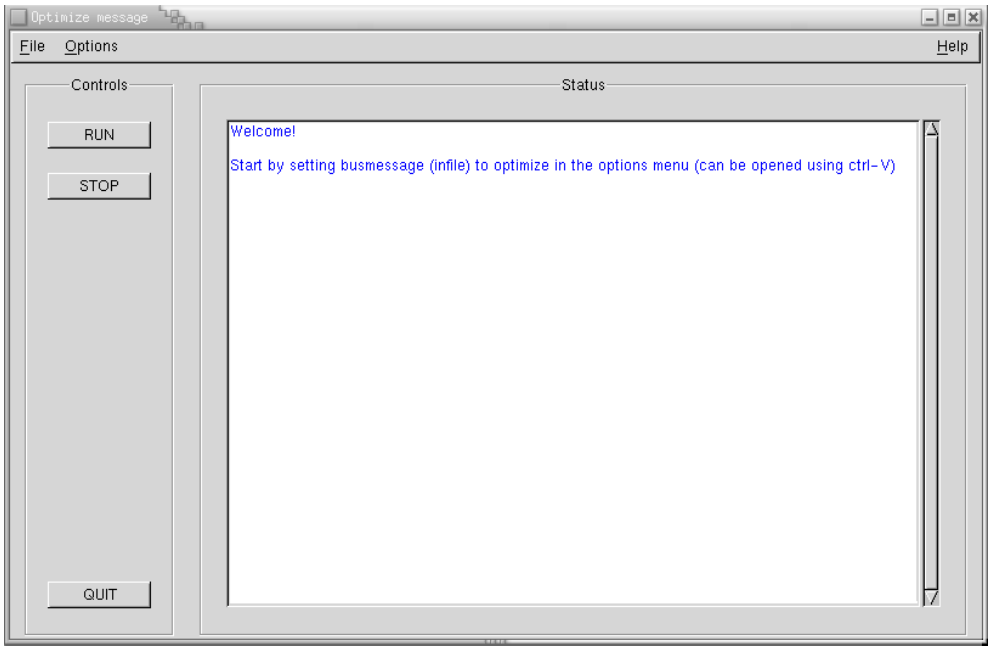
Kort version:

- Starta verktyget.
- Tryck ctrl-v för att visa menyn.
- Ange namn på infil.
- Om en optimering nyligen utförts, radera namnet på angiven utfil (alternativt kontrollera att namnet på utfilen inte krockar med någon existerande fil i filsystemet). Anges ingen utfil kommer det automatiskt att sättas till namnet på infilen avslutat med '.tmp'.
- Spara inställningarna och gå till huvudfönstret med "Ok".
- Tryck "RUN" för att starta optimeringen.

Resultatet kommer lagras i utfilen då optimeringen är färdig.

Lång version:

Då programmet startats, kommer fönstret i figur 11 visas. Innan optimeringen kan startas måste minst en inställning göras, namnet på den fil som bussmeddelandet ligger i måste anges. Ta fram inställningsmenyn m.h.a. ctrl-v eller välj Options→"View and set options" i menylistan.



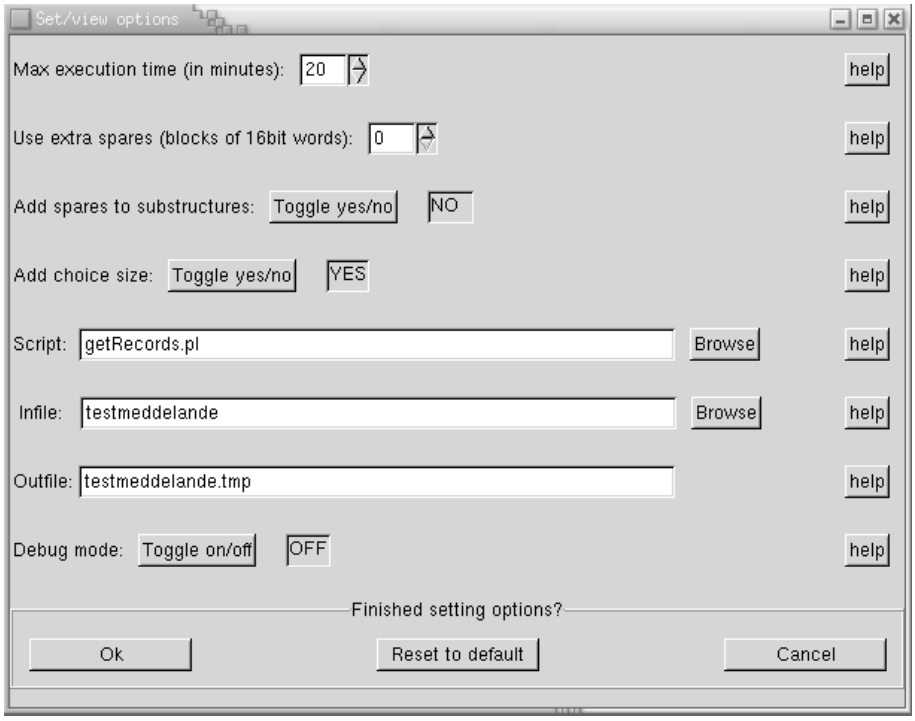
Figur 11: Applikationens huvudfönster.

I menyn som visas i figur 12 behöver i de flesta fall enbart namnet anges på filen med bussmeddelandet.

De inställningar som finns att göra är:

Maximal exekveringstid: Den maximala tiden som användaren är beredd att vänta på att optimeringen blir färdig. I de flesta fall kommer optimeringen bli klar innan 20 minuter. Dock kan vissa meddelanden ge stora problem och ta väldigt lång tid. Observera att det kan finnas vissa meddelanden där det är omöjligt att hitta en optimal lösning, oberoende av hur lång exekveringstid som väljs.
Defaultvärde: 20 minuter

Användande av extra reserver: Utifall flera reservbitar behövs, t.ex. för att underlätta optimeringen eller av andra skäl, anges detta här. Observera att det bara är



Figur 12: Menyn för inställningar inför optimeringen.

möjligt att lägga till reservbitar i grupper om 16 st åt gången.
Defaultvärde: 0 (inga extra reserver)

Lägga till reserver i substrukturer: För vissa meddelanden får inget nytt data läggas till i substrukturer[28], som records och cases, på grund av att dessas storlekar inte får ändras. Om denna flagga är satt kan utplacering av reservbitar även ske i substrukturer vilket kan underlätta optimeringen.
Defaultvärde: NO (reserver kommer ej att läggas till i substrukturer)

Räkna med storleken på flagga för val i case: I varje case finns det en flagga som används för att indikera vilket av fälten i caset som ska skickas över bussen. Denna flagga ligger först i det case den används och tar en viss plats. Det är möjligt att i vissa fall låta bli att skicka med denna flagga. Eftersom denna flagga tar plats kommer den även påverka optimeringen, varför den måste vara satt till NO om den storleken inte ska användas.
Defaultvärde: YES (denna flagga kommer skickas med över bussen)

Script: Namn på det parsningsscript som används. Denna är alltid satt till ett default filnamn, och ska inte ändras om inte t.ex. scriptet har modifierats tillfälligt etc.

[28]substrukturer kallas de cases och records som ligger inuti bussmeddelandet.

Defaultvärde: Beroende på vad miljövariabeln för denna sökväg är satt till

Infil: Namn på den fil som bussmeddelandet ligger i. Notera att verktyget förutsätter att endast ett meddelande ligger i denna fil, samt att bussmeddelandet är på rätt format. Anges inget namn här kommer inte optimeringen att starta.
Defaultvärde: Inget

Utfil: Namn på den fil som det optimerade bussmeddelandet ska skrivas till. Om inget namn anges kommer filnamnet automatiskt att sättas till namnet på infilen avslutat med '.tmp'. Om den fil som anges här redan finns kommer inte optimeringen att kunna startas, eftersom verktyget av säkerhetsskäl inte tillåts skriva över någon fil i filsystemet.
Defaultvärde: Inget

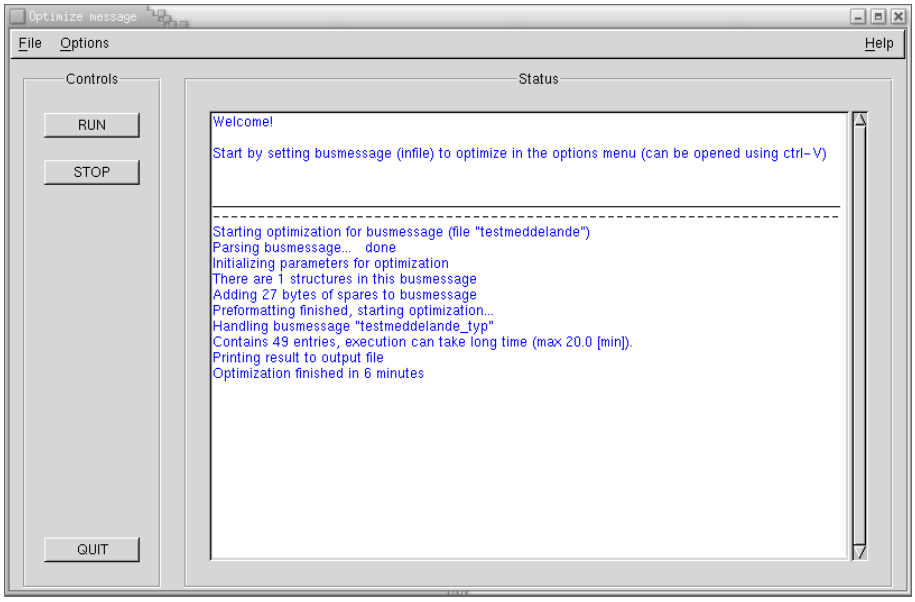
Debug läge: Då debugläget är på kommer storlekarna på alla typer och strukturer att skrivas till resultatfilen som kommentarer, d.v.s. på formen (* storlek *).
Defaultvärde: OFF

När alla inställningar är gjorda sparas resultatet genom att klicka på "Ok". Då stängs även menyfönstret ner.

Vill man återställa alla värden till defaultvärden ska "Reset to default" användas. Och om man ångrar de ändringar man gjort i menyn kan de gamla värdena återställas genom att klicka "Cancel".

För att sedan starta optimeringen, tryck bara på "RUN". Under optimeringens gång kommer feedback från optimeringsrutinen kontinuerligt att visas i statusfönstret, se figur 13. Feedbacken består av vad optimeringsrutinen gör just nu, lite information angående bussmeddelandet, problem som uppstått t.ex. om någon substruktur inte går att optimera helt etc.

Vill man stoppa optimeringen och få ett resultat innan den är färdig går det att göra med "STOP" knappen. Notera att resultat endast kan ges för meddelanden som är enkla, d.v.s. som ej innehåller några substrukturer. Detta anges i statusfönstret i början av optimeringen med raden "There are 1 structures in this busmessage". Då användaren stoppar optimeringen kommer först en varning att ges om det är möjligt att skriva ut resultatet till filen. Notera att en varning även kommer ges då det går att få resultatet, fast då går varningen ut på att meddelandet inte är helt optimerat.
Ett gott råd är att vänta och låta optimeringen bli klar, det ger bäst resultat. För att kunna få ut ett resultat då optimeringen stoppats i förtid är ett krav att optimeringen åtminstone har passerat förformatteringsfasen.
För att avsluta verktyget, använd "QUIT".
För att optimera ett nytt meddelande, gå in i menyn igen och ange ett nytt filnamn (samt radera/ändra namnet på utfilen).



Figur 13: Resultatet av en optimering.

Referenser

[1] Aarts, Emile & Lenstra, Jan Karel: Local Search in Combinatorial Optimization, Wiley, Chichester, 1997

[2] Aarts, E. H. L. & Korst, J. H. M.: Simulated Annealing and Boltzmann machines, Wiley, Chichester, 1989

[3] Bazaraa, Mokhtar S. & Shetty, C.M.: Nonlinear programming : theory and algorithms, Wiley, New York, 1979

[4] Gass, Saul I. & Harris, Carl M. (editors): Encyclopedia of operations research and management science, 2nd ed. (Centennial edition), Kluwer academic publishers, Boston, 2001

[5] Heath, T. Michael: Scientific Computing: An introductory survey, second edition, McGraw-Hill, Boston, 2002

[6] Laarhoven, P.J.M. van & Aarts, E.H.L.: Simulated annealing: Theory and applications, Kluwer Academic, 1987

[7] L'Ecuyer, P.: Efficient and portable combined random number generators, Communications of the ACM, Volume 31, Issue 6 (June 1988), pp. 742 - 751, ACM Press, New York, 1988, ISSN 0001-0782

[8] Lundgren, Jan & Rönnqvist, Mikael & Värbrand, Peter: Optimeringslära, Studentlitteratur, Lund, 2003

[9] Metropolis, N. & Rosenbluth, A. W. & Rosenbluth, M. N. & Teller, A. H. & Teller, E.: Equation of state calculation by fast computing machines, Journal of Chemical Physic, 1953, 21, pp. 1087–1092

[10] Michalewicz, Zbigniew: Fogel, David B: How to solve it: Modern heuristics, Springer, New York, 2000

[11] Migdalas, Athanasios & Lundgren, Maud G.: Kombinatorisk Optimering: Problem och algoritmer, Linköpings universitet, 1999

[12] Condor Engineering Inc.: MIL-STD-1553 Tutorial, http://www.condoreng.com/support/downloads/tutorials/MIL-STD-1553Tutorial.PDF, senast uppdaterad: 2003-08-05, senast besökt: 2004-04-05

[13] Nõu, andreas: Large scale combinatorial Optimization with transportation science applications, Doktorsavhandling, Tekniska högskolan i Stockholm, Institutionen för matematik, Stockholm, 1997, Trita-MAT. OS, ISSN: 1401-2294 :97:15, Nr-beteckn. KTH/OPT SYST/DA–97/15–SE

[14] Press, William H. & Teukolsky, Saul A. & Vetterling, William T. & Flannery, Brian P.: Numerical recipes in C: The art of scientific computing, Second edition: Cambridge University press, Cambridge, 1999

[15] Reeves, Colin R. (editor): Modern heuristic techniques for combinatorial optimization problems, Blackwell, Oxford, 1993

[16] Kirkpatrick, S. & Gelatt, C. D. & Vecchi, M. P.: Optimization by Simulated Annealing, Science, no. 4598, Vol. 220, 13 May 1983, pp. 671-680

[17] Stoica, Anca-Juliana & Lanshammar, Håkan & Sandblad, Bengt: AFR-kompendium: Selected topics on Optimization Techniques and Applications: An introduction, Avfallsforskningsrådet AFR (Systems and Control Group, Uppsala University), AFN/Naturvårdsverket, Stockholm, 1994, ISSN: 1400-0210 ;3

[18] Winston, Wayne L.: Operations Research: Applications and Algorithms, Third edition: Duxbury Press, Belmont, 1994

[19] Baker, Kenneth R.: Introduction to sequencing and scheduling, Wiley & Sons 1974