Institutionen för kommunikation och information
Examensarbete i datavetenskap 30hp
C-nivå
Vårterminen 2009

# Lokal sökalgoritm för initiering av den genetiska populationen i ett praktiskt ”vehicle routing”-problem

**Lars Persson**

Datum: 090607
Handledare: Anna Syberfeldt
Examinator: Henrik Gustavsson

**Lokal sökalgoritm för initiering av den genetiska populationen i ett praktiskt ”vehicle routing”-problem**

Examensrapport inlämnad av Lars Persson till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Anna Syberfeldt.

**2009-06-07**

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

**Lokal sökalgoritm för initiering av den genetiska populationen i ett praktiskt ”vehicle routing”-problem**

**Lars Persson**

# Sammanfattning

Befintliga studier har påvisat att genetiska algoritmer presterar bättre om de ges en bra startpopulation. I denna rapport presenteras en lokal sökalgoritm för att skapa en population med fokus på ”vehicle routing”-problem. Algoritmen använder sig av heuristik i en blandning av simulated annealing och tabu search för att skapa individerna till populationen. Utvärderingar av algoritmen på ett praktiskt problem visar att den ger en bra start jämfört med en slumpmässig startpopulation, vilket är vanligt att använda. Resultaten av utvärderingen visar också att algoritmen ger bäst resultat vid mer komplexa problem, medan den har mindre effekt om problemet är enklare.

**Nyckelord:** Simulated annealing, Tabu search, Genetiska algoritmer , ”vehicle routing”-problem.

# Innehållsförteckning

**1 INTRODUKTION ..... 1**

**2 BAKGRUND ..... 3**

2.1 VEHICLE ROUTING PROBLEM ..... 3
2.2 GENETISKA ALGORITMER ..... 4
*2.2.1 Evolutionär process ..... 4*
*2.2.2 Beståndsdelar ..... 5*
2.3 LOKALA SÖKALGORITMER MED HEURISTIK ..... 8

**3 PROBLEMBESKRIVNING ..... 10**

3.1 PROBLEMOMRÅDESBESKRIVNING ..... 10
3.2 PROBLEMPRECISERING ..... 11
3.3 SYFTE OCH MÅL ..... 11
3.4 DELSTEG OCH METOD ..... 12

***4* GENOMFÖRANDE ..... 13**

4.1 PRAKTISKT PROBLEM ..... 13
4.2 OPTIMERINGSMÅL ..... 14
4.3 DESIGN ..... 15
4.4 IMPLEMENTATION ..... 18

**5 UTVÄRDERING ..... 24**

5.1 RESULTAT ..... 25
5.2 ANALYS ..... 30

**6 SLUTSATSER ..... 32**

6.1 SAMMANFATTNING ..... 32
6.2 DISKUSSION ..... 32
6.3 FRAMTIDA ARBETE ..... 33

# 1 Introduktion

”Traveling sales person” är ett klassiskt problem som går ut på att hitta den kortaste vägen mellan ett antal noder. Problemet låter simpelt men är klassat som ”NP-hard” i sin enklaste form (Santos, 2008). Detta innebär att tiden det tar att lösa problemet växer exponentiellt med problemstorleken. En variant av ”traveling sales person”-problemet är ”vehicle routing”-problemet där ett antal lastbilar ska besöka sina kunder genom att köra så kort sträcka som möjligt. I grunden är dessa två problem teoretiska men ju fler parametrar som läggs till ju mer liknar de verkligheten och ju svårare blir det att lösa dem. Ett riktigt logistikföretag har till exempel ett visst antal lastbilar där varje lastbil har en viss kapacitet som vikt och volym, de har ofta speciella arbetstider och olika lastbilar åker olika snabbt och har olika driftskostnad.

Vikten av att lösa problemet med ruttoptimering ligger i att det kostar mycket att inte åka den optimala rutten. Det kostar både pengar i form av slitage på lastbil, bränsle och lön till chauffören. Det kostar även i miljönhänseende. Enligt Dablanc (2007) står godstransport för mellan 16-50 % av alla luftföroreningar från fordon i städer, beroende på vilka ämnen man räknar. Det går att hitta den optimala rutten med hjälp av en dator men det kan ta väldigt lång tid eftersom alla tänkbara rutter måste utvärderas för att hitta den bästa. Även för små problem finns en mycket stor mängd möjliga rutter; för ett problem med bara 15 städer att besöka så är antalet möjliga rutter 6 227 020 800 stycken. På grund av komplexiteten hos problemet så använder man ofta tekniker som inte garanterat hittar den optimala lösningen men som hittar en tillräckligt bra lösning på mycket kortare tid.

Genetiska algoritmer är en av många tekniker som kan användas för att lösa ruttoptimeringsproblem (Jozefowiez, Semet och Talbi, 2008). Genetiska algoritmer baseras på Darwins idéer om att den som är bäst anpassad överlever. I en population får de som har bäst egenskaper para sig och få barn som för generna vidare. De bra generna gör att de överlever längre och populationen blir bättre och bättre för varje generation. Genetiska algoritmer har tre viktiga operatorer enligt Chinneck (2006). Den första av dessa är selektionsoperatorn som väljer ut vilka individer som får para sig. Ofta är det en fördel för de individer som har bättre egenskaper, så kallad fitness. Den andra operatorn är parningsoperatorn som är den funktion som utför själva parningen, dvs. bestämmer hur individerna för sina gener vidare. Efter parningen finns det en liten chans att avkomman blir muterad, vilket görs med den tredje operatorn som är mutationsopteratorn. Mutationsoperatorn är den funktion som gör en liten förändring på individen. Dessa operatorer används på varje generation för att skapa en bättre och bättre population av lösningar.

I den enklaste varianten av genetiska algoritmer slumpas den första populationen fram. Det ger ofta en bra spridning på populationen men kan göra att det tar onödigt lång tid innan en tillräckligt bra lösning har evolverat fram. Chinneck (2006) skriver att algoritmen presterar mycket bättre med en bättre startpopulation. Svårigheten blir då att behålla mångfalden bland individerna. Om några individer sticker ut och är mycket bättre

än alla andra kommer hela populationen snabbt att likna varandra och lätt fastna i ett lokalt maxima.

Det här arbetet hittar ett nytt sätt att använda lokala sökalgoritmer med heuristik för att skapa den första populationen i en genetisk algoritm med flera mål för att lösa en variant av vehicle routing problemet. Målen med arbetet är att hitta ett praktiskt problem att skapa en första population till, identifiera vilka mål som det praktiska problemet strävar mot för att sedan designa och implementera algoritmen som genererar population till den genetiska algoritmen. Efter implementationen ska testning och utvärdering ske.

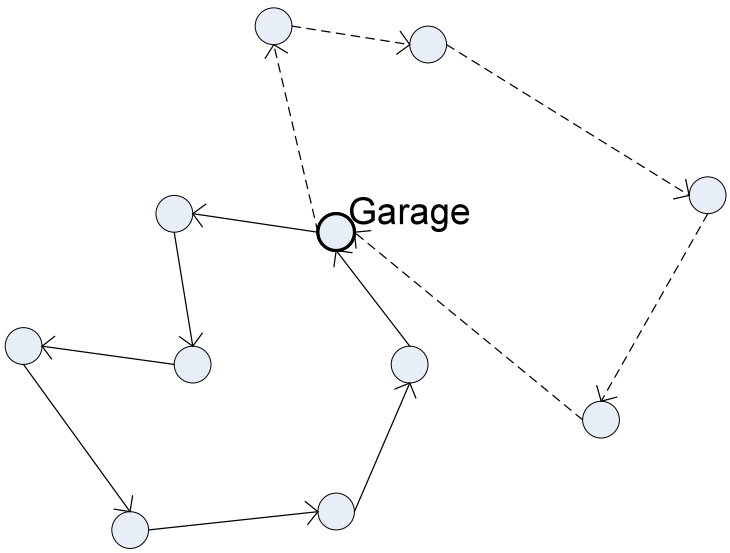
# 2 Bakgrund

Detta kapitel är uppdelat i tre delar. Delkapitel 2.1 förklarar vad vehicle routing problem är för något. Delkapitel 2.2 beskriver hur genetiska algoritmer fungerar och delkapitel 2.3 beskriver två söktekniker som använder heuristik.

## 2.1 Vehicle routing problem

”Traveling sales person” (TSP) är ett problem som går ut på att hitta den kortaste vägen mellan alla noder. Det går att jämföra med en handelsresande som ska resa runt till alla städer och sälja sina produkter. Den handelsresande börjar hemma i sin egen stad och vill resa till alla andra städer och tillbaka till sin egen igen. Resan ska vara den kortast möjliga för att förtjänsten av försäljningen inte ska ätas upp av allt resande. Problemet låter simpelt men har som nämts tidigare hög tidskomplexitet.

”Vehicle routing problem” (VRP) är en variant på TSP där det går ut på att en flotta av fordon ska besöka alla kunder för så låg kostnad som möjligt (se Figur 1). Kostnaden är ofta tid det tar att köra eller distansen som måste köras. Det kan även finnas flera krav på designen av rutterna (Kang, Lee, Lee och Lee 2008). Till exempel kan fordonen ha en maximal lastkapacitet eller maximal arbetstid. Kunderna kan ha begränsningar i form av att bara vara tillgänglig mellan vissa tider eller att efterfrågan varierar. VRP är liksom TSP klassat som ”NP-hard” (Kang et al. 2008). Eftersom det tar mycket lång tid att lösa TSP och VRP optimalt för problem som är lite större används tekniker som går fortare men inte garanterar att hitta den optimala lösningen. Tekniker som användas är bland andra genetiska algoritmer, tabu search och simulated annealing. Dessa tekniker beskrivs i följande delkapitel.




**Figur 1- VRP med två bilar (heldragen respektive streckad linje) som besöker ett antal noder.**

## 2.2 Genetiska algoritmer

Genetiska algoritmer baseras på Darwins lära om de bäst anpassades överlevnad, så kallad ”survival of the fittest”. Varje individ eller lösning representeras av ett genom. Genomet är i sin enklaste form en lista med ettor och nollor, alltså i binär form. Men det går också att använda decimaltal eller hela dataobjekt i en lista beroende på problemets form. I Figur 2 visas exempel på två genom, den första av dessa har en binär representation, medan det andra har en decimal representation.

| | | | | | | |
|---|---|---|---|---|---|---|
| Binärt genom | 0 | 0 | 1 | 1 | 1 | 0 |
| Decimalt genom | 1,2 | 2,5 | 5,2 | 3,2 | 8,7 | 7,3 |

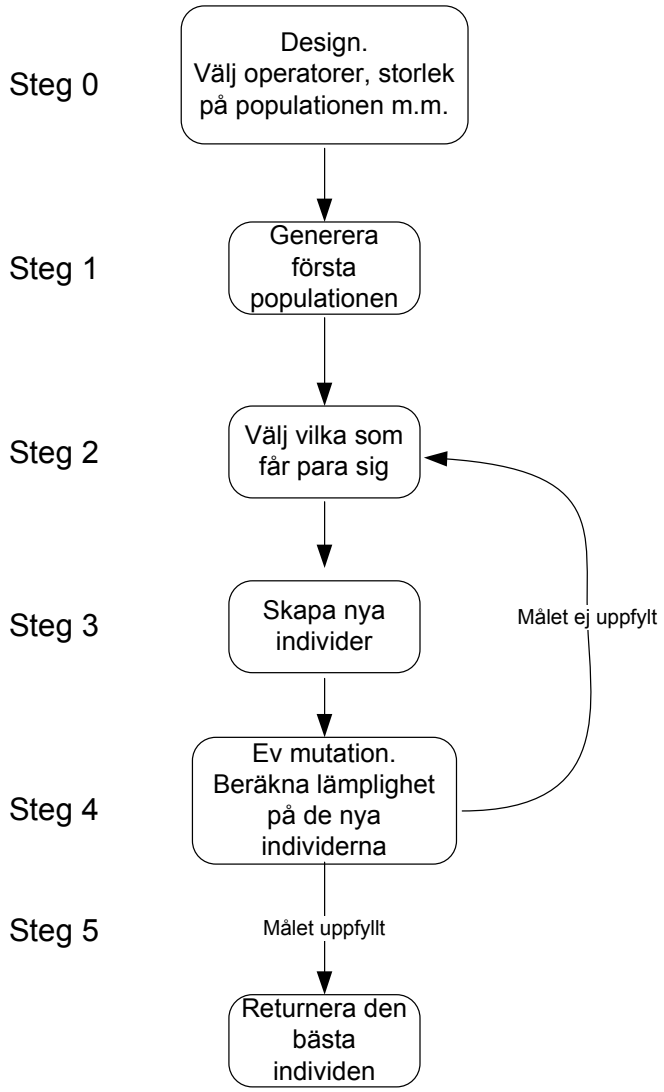
**Figur 2 - Exempel på genomrepresentationer.**

En bra lösning hittas genom att en mängd lösningar skapas och utvärderas. Nya individer skapas genom att blanda de bästa individerna med varandra så att de nya individerna får delar från de gamla. De nya individerna får förhoppningsvis bra delar från båda sina föräldrar och blir då ännu bättre än sina föräldrar. För att inte lösningarna ska bli väldigt lika varandra så finns det en liten chans att varje ny individ blir muterad. Detta betyder att en individ får en liten förändring i sin uppbyggnad oberoende av sina föräldrar. En mutation i binär form kan vara att en nolla blir till en etta eller tvärtom. Varje generation är lite bättre än sin tidigare generation eftersom de minst lämpliga individerna gallras bort.

### 2.2.1 Evolutionär process

Chinnek (2006) beskriver genetiska algoritmer med följande steg (se även Figur 3):

**0.** Designa algoritmen. Välj storlek på populationen och mutationschans. Välj operatorer, det vill säga selektionssoperator, parningsoperator och muteringsoperator. Välj slutkrav som säger när algoritmen ska avslutas.
**1.** Generera den första populationen, beräkna alla dess individers lämplighet.
**2.** Använd selektionssoperatorn på den nuvarande populationen för att generera en grupp som får para sig.
**3.** Använd parningsoperatorn på gruppen som blev utvald i steg 2 för att skapa en ny generation av individer.
**4.** Använd muteringsoperatorn på den nya generationen. Beräkna sedan lämplighet på alla individer i den nya generationen. Spara undan den som är bäst lämpad.
**5.** Stanna om slutkravet är uppnått, gå annars till steg 2.



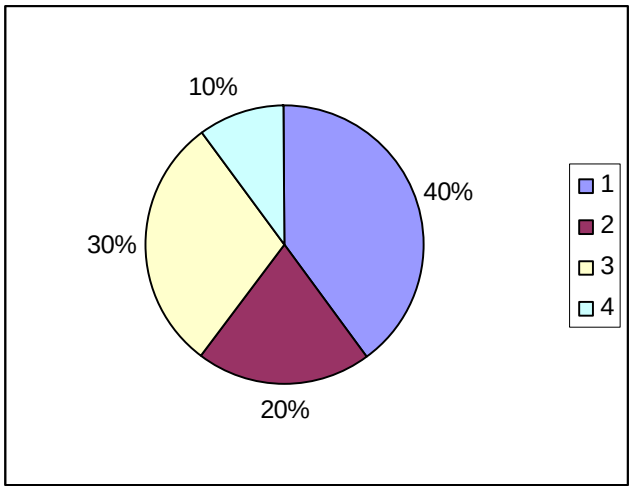

**Figur 3 – Bild av flödet i en genetisk algoritm.**

### 2.2.2 Beståndsdelar

För att resultatet ska bli så bra som möjligt måste alla delar av algoritmen vara så bra som möjligt. Gällande populationens storlek, som bestäms i steg 0 i algoritmen (Figur 3), är detta något som får testas fram beroende på problemets storlek och komplexitet. Populationen får inte vara för liten för då kommer den ha väldigt svårt att ta sig förbi lokala optima eftersom det finns få varianter av individer. Populationen får heller inte vara för stor för då tar det onödigt lång tid för populationen att resultera i bra individer. Om den första populationen som skapas i steg 1 (Figur 3) inte är helt slumpmässig utan har utformats för att lösa målet så kommer evolutionen att få bättre resultat (Chinneck 2006). Svårigheten är dock lite samma som för en för liten population, om populationen är för ensidig kommer den att få problem med lokala optima. För att hjälpa utvecklingen på traven så går det att skapa den första populationen med lite hjälp av heuristik i stället för att bara slumpa fram den. Chinneck (2006) skriver att genetiska algoritmer presterar bättre om den första generationen har högre lämplighet. Problemet är då att populationen inte ska bli för ensidig så de direkt fastnar i ett lokalt optimum.

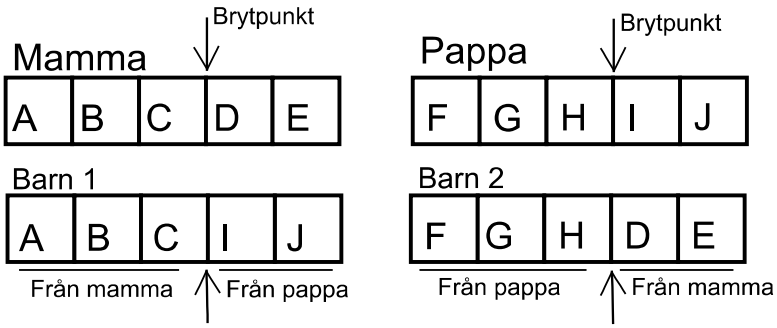
Selektionsoperatorn som används i steg 2 av algoritmen (Figur 3) är till för att välja ut de individer som får para sig genom att gallra bort de sämre individerna. Gruppen med individer som får para sig ska ha lika många individer som hela populationen (Deb 2001). Operatorn väljer alltså flera individer flera gånger. Två varianter av selektionsoperatorn är tournament selection och roulette-wheel selection (Deb 2001). Tournament selection går ut på att välja ut två individer som får mötas i en duell. Vinnaren är den som har bäst lämplighet och är den som får läggas till i gruppen som får para sig. Sedan väljs två nya individer och en ny duell sker. Detta upprepas tills rätt antal som ska få para sig är valda. För att öka mångfalden i populationen går det se till att alla ska vara med i turneringen två gånger. Då kommer den med bäst lämplighet vinna två turneringar och den som har sämst lämplighet förlora två turneringar. Det är inte helt strikt att de som har en lämplighet under snittet i populationen försvinner eftersom de två minst lämpade kan råka mötas och då vinner den näst minst lämpade och får vara med i parningen.

Roulette-wheel selection går ut på att varje individ får en chans att väljas som är baserad på dess lämplighet. Det går att se det som ett routlettehjul som är indelat i lika många ”fack” som det finns individer. Den individ som har störst lämplighet har också störst del av hjulet medan den som har minst lämplighet har minst del (se exempel i Figur 4). Om den totala lämpligheten hos alla individer är 50 och den bästa individen har en lämplighet på 20 så har den individen en sannolikhet på, 20/50 alltså, 40 % att väljas. Detta går att implementera genom att slumpa fram ett tal mellan noll och summan av alla individers lämplighet. Individ ett i detta fall representerar 1 - 20, individ 2 representerar 21 - 30 och så vidare.




**Figur 4 – Ett roulette-hjul med fyra individer med olika chans att väljas.**

Parningsoperatorn som används i steg 3 av algoritmen (Figur 3) är den funktion som ser till att nya individer skapas. Den kombinerar två individer för att skapa två nya. Det finns olika metoder att använda beroende på hur problemet ser ut och hur genomen är uppbyggda. Två vanliga metoder är en-punkts parning (Figur 5) och två-punkts parning (Deb 2001). I en-punkts parningen slumpas en punkt fram i genomet. Ett nytt genom skapas genom att ta pappans genom fram till den slumpade punkten och resten från mamman. Det andra genomet skapas sedan genom att ta första delen från mamman och den andra från pappan. I två-punkts parning väljs två punkter ut. Barnen får då två delar från den ena föräldern och en del från den andra. Dessa metoder fungerar dock inte på alla problem. Ibland får det inte finnas dubletter av gener i genomet. Om A ligger i början av pappan och i slutet av mamman kan barnet få två av samma gen vilket är ogiltigt. Parningsoperatorn måste då anpassas så att den bara resulterar i giltiga barn.




**Figur 5 – En-punkts parning där två föräldrar skapar två barn.**

I steg 4 av algoritmen (Figur 3) används mutationsoperatorn. Används inte mutering försvinner lätt gener från populationen när individer gallras bort. Generna kanske inte passade in på just den individen som försvann men i en annan individ kanske de skulle gjort nytta och hjälpt till att hitta den optimala lösningen. För att hjälpa populationen ha kvar sin mångfald så kan muteringsoperatorn se till att slumpartade förändringar sker i nya individer. Det kan vara att byta ut en gen mot en helt annan eller att byta plats på två gener i samma genom. Vilken mutering som passar bäst är beroende av problemet och genomets uppbyggnad. Blir det för mycket muteringar i populationen är risken att de bra framavlade individerna muteras sönder och varje generation blir sämre och sämre. Hur mycket mutering som passar i varje problem får testas fram.

Efter eventuell mutering beräknas lämpligheten hos individen. Hur den beräknas beror på problemet. Om det är ett vehicle-routing problem som beräknas kan det vara den totala sträckan som körs som avgör. Ofta i praktiken så är det flera saker som avgör hur bra en individ är då det finns flera mål att sträva mot. Då kan varje mål resultera i poäng som adderas ihop och bildar ett så kallat fitnessvärde hos individen, ju högre fitnessvärde ju bättre individ. Denna teknik för att beräkna lämplighet kallas också för en viktadsumma.

Processen med parning och mutering fortsätter i oändlighet om inget slutkrav finns. Kravet kan vara att iterera ett visst antal generationer eller avsluta när populationen inte förbättras längre. Chinneck (2006) föreslår att man ska titta på hur mycket den sämsta individen förbättras. Har inte den sämsta individen förbättrats på flera generationer har troligtvis inte den bästa heller förbättrats.

## 2.3 Lokala sökalgoritmer med heuristik

Lokala sökalgoritmer kan beskrivas som att de utgår från en lösning och undersöker dess grannar (Aarts och Lenstra, 1997). Den väljer den bästa av grannarna och fortsätter iterativt tills ett lokalt optimum har hittats. En granne är en närliggande lösning och ofta kan heuristik användas för att avgöra vilken granne som är bra att välja. Heuristik är enligt Hillam (2003) sunt förnuft eller antaganden som brukar stämma utifrån den fakta som finns tillgänglig. När det gäller att hitta kortaste vägen så brukar det vara bra att gå raka vägen mot målet. Det är inte säkert att det är kortaste vägen, det kan komma en flod som gör att man tvingas ta en omväg över en bro.

Två lokala sökalgoritmer som använder heuristik för att lösa problem är simulated annealing och tabu search (Michalewicz och Fogel, 2000). Simulated annealing går till på följande sätt. Först väljs en slumpmässig lösning ut som utvärderas. Sedan väljs en granne till lösningen, dvs. en lösning som är ganska lik den första lösningen. Till exempel kan lösningarna ha samma genuppsättning men skilja sig åt vad gäller ordningen på några av generna. Är grannen bättre än den första lösningen så sparas grannen i stället för den tidigare lösningen. Skulle grannen vara sämre så finns det en chans att denna sparas ändå. Om den sparas eller ej bestäms av temperaturen T och hur stor skillnad det var mellan de två lösningarnas lämplighet. Är temperaturen hög så är chansen lika hög att den dåliga lösningen sparas som att den bra lösningen sparas. Är temperaturen låg är chansen att sparas större för den bra lösningen. Efter att en av lösningarna är sparad så sänks temeperaturen lite och en ny granne tas fram och jämförs med den lösning som sparades i förra iterationen. Detta håller på tills slutmålet är nått. Målet kan vara att temperaturen är tillräckligt låg eller att ingen förbättring har skett på länge. Eftersom temeraturen sjunker i varje iteration betyder det att lösningarna i början är väldigt slumpmässiga och att de efterhand bara byts om den nya lösningen är bättre.

Michalewicz och Fogel (2000) beskriver simulated annealing för ett maximeringsproblem med följande pseudokod (se Figur 6):

```
Procedure Simulated-annealing
Begin
    t ← 0
    initialize T
    select a current point v_c at random
    evaluate v_c
    repeat
        repeat
            select a new point v_n in the neighborhood of v_c
            if eval(v_c) < eval(v_n)
                then v_c ← v_n
            else if random[0,1) < e^((eval(v_n) − eval(v_c))/T)
                then v_c ← v_n
        until(termination-condition)
        T ← g(T, t)
        t ← t + 1
    until(halting-criterion)
end
```

**Figur 6 - Pseudokod för simulated annealing.**

I proceduren är $v_c$ är ett lösningsförlag och $v_n$ är ett lösningsförslag som är en granne till $v_c$. T är temperaturen, t är temperaturförändringen. *eval* är en funktion som beräknar lämpligheten på ett lösningsförslag. Lämpligheten kan beräknas med hjälp av heuristik. Funktionen *random* returnerar ett slumpmässigt värde mellan 0 och 1 i detta fall. Funktionen *g* gör en sänkning av T beroende på t. Hur lämplig en lösning är beror på vilket problem den löser. Är problemet att hitta en rutt så kan den vara bra om sträckan är kort. Om det handlar om att hitta den högsta bergstoppen i en bergskedja är det bra att vara så högt över vattnet som möjligt.

Tabu search är en annan lokal sökalgoritm som använder heuristik. Heuristiken används i detta fall för att jämföra grannar och välja den granne som är bäst, likt simulated annealing. Det som gör tabu search speciellt är att de senaste lösningsutvärderingarna sparas i ett minne (Michalewicz och Fogel, 2000). Minnet kan exempelvis vara hur längesedan varje del av genomet förändrades, till exempel hur längesedan en bit ändrades om en binär lösningsrepresentation används. För att behålla spridningen kan man till exempel ha som krav att samma bit inte får ändras inom fem iterationer.

# 3 Problembeskrivning

Detta kapitel beskriver det övergripande problemområdet för arbetet och det specifika problemet inom detta område som studeras. Syfte och mål med arbetet beskrivs också.

## 3.1 Problemområdesbeskrivning

Som tidigare beskrivits i kapitel 2 är det första steget i en genetisk algoritm är att skapa lösningarna för den initiala populationen. Den lösning som ursprungligen föreslagits för att göra detta är att slumpa fram lösningarna (Holland, 1992), och än idag är det vanliga sättet för att skapa den initiala populationen att göra det slumpmässigt (Chinneck, 2006). Att slumpa fram lösningarna är enkelt, men det är inte speciellt effektivt med avseende på algoritmens prestanda. Slumpmässiga lösningar är ofta dåliga och att förfina och förbättra lösningarna kvalitet tar lång tid för algoritmen då detta måste göras över flera generationer. Vilket resultat algoritmen kan uppnå beror bland annat på den kvalitén som lösningarna i den initiala populationen har, det vill säga vilken utgångspunkt som algoritmen har (Chinneck, 2006).

Det finns flera olika sätt som har förslagits för att intiera den första populationen, även om detta problem har ägnats förhållandevis lite uppmärksammhet och de flesta använder sig av slumpmässiga lösningar (Johnson, 1990 sid. 270). Ett förslag är att använda experters domänkunskap för att skapa en eller flera lösningar som man kan utgå ifrån (Johnson, 1990). Nackdelen med detta förslag är att det kräver att det finns experter med tillräcklig domänkunskap och att problemet inte är alltför komplext så att det går att ta fram lösningar manuellt. Ett annat förslag är att använda lösningar från tidigare optimeringar (Louis och Li, 2000). Nackdelen med detta förslag är att det kräver att tidigare optimeringar har körts under samma (eller åtminstone) liknande omständigheter, vilket kräver en mer eller mindre statisk domän.

Ett tredje förslag är att använda heuristik för att skapa lösningar till den första populationen. Heuristisk är ett antagande som baseras på den information som finns tillgänglig. Tillexempel så kan heuristik för ett problem där det gäller att hitta den kortaste vägen mellan ett antal noder uppskatta längden för den kortaste vägen mellan noderna. Heuristik garanterar inte korrekta eller optimala lösningar, men de kan ofta ge hyfsade lösningar. Heuristik används av lokala sökalgoritmer såsom simulated annealing och tabu search för att guida sökningen.

Idén med att använda heuristik för att initiera den första populationen istället för att skapa den slumpmässigt uppmärksammades av Grefenstette redan år 1987 (Grefenstette, 1987). Grefenstette spekulerade kring att om värdefullt genetiskt materiel kunde tillföras den genetiska algoritmen redan från start så kunde algoritmen uppnå bättre resultat.

## 3.2 Problemprecisering

De lösningar som har tagits fram i tidigare studier kring att använda heuristik för TSP och liknande problem såsom VRP har generellt två begränsningar. För det första så är de framtagna för och utvärderade enbart på ett teoretiskt problem, och för det andra så tar de bara hänsyn till en enda parameter, nämligen avståndet mellan städer. I verkliga problem som liknar TSP så finns det vanligtvis många fler parametrar att ta hänsyn till (t.ex. last, tidsåtgång, miljöpåverkan, etc.) och komplexiteten hos problemet är betydligt större, exempelvis som en följd av begränsningar som finns (t.ex. maximal last, lagstadgade raster, etc.). Johnson (1990, sid 270) menar att det, speciellt för verkliga problem, är troligt att det finns effektivare sätt att initiera populationen på än de som tidigare föreslagits.

Att använda heuristik för att initiera den första populationen är dock inte helt trivialt. Grefenstette (1987) belyser ett viktigt problem med att använda heuristik, nämligen att algoritmen snabbt kan fastna i ett lokalt optimum om det finns ett fåtal lösningar i populationen som är överlägsna de andra. Dessa lösningar kommer då snabbt att ta över populationen och mångfalden bland generna i populationen går då förlorad. När lösningarna blir väldigt lika varandra så kommer sökningen att avstanna och inte komma vidare.

En annan viktig aspekt vid användandet av heuristik för att initiera den första populationen är enligt Johnson (1990) lösningsrepresentationen. Lösningarna i den initiala populationen blir bara användbara om de använder samma genetiska representation som den genetiska algoritmen (Johnson, 1990, sid. 274). Därför måste lösningsrepresentationen för heuristiken och för den genetiska algoritmen synkroniseras.

## 3.3 Syfte och mål

Syftet med detta arbete är att ta fram och utvärdera en populationsgenereringsalgoritm (PGA) som använder lokal sökning med heuristik för att initiera populationen för genetiska algoritmer som appliceras på verkliga problem liknande VRP. Med hjälp av denna PGA ska den genetiska algoritmen få ett försprång för att uppnå bättre resultat.

Den PGA som tas fram måste alltså utformas på ett sätt så att för tidig konvergens undviks, dvs. att lösningarna inte blir alltför lika varandra och algoritmen på grund av detta fastnar i ett lokalt optimum. Vidare så måste PGAn ta hänsyn till den lösningsrepresentation som den genetiska algoritmen använder.

## 3.4 Delsteg och metod

Arbetet läggs upp som fem steg, vilka beskrivs nedan.

**Steg 1**
Identifiera ett praktiskt VRP som kan användas för att utveckla och testa en populationsgenereringsalgoritm för att initiera en population. Detta görs genom att kontakta ett företag som sysslar med transporter. För att ha tillräcklig komplexitet och vara relevant för arbetet har följande kriterier satts upp för det praktiska problemet:

- Problemet ska innebära att minst 15 noder ska besökas (vilket ger en sökrymd som omfattar minst 6227020800 möjliga lösningar).
- Fordonen ska ha en begränsad maxkapacitet.
- Fordonens förare ska ha en begränsad arbetstid.

**Steg 2**
Identifiera vilka optimeringsmål den lokala sökalgoritmen ska sträva mot när den skapar populationen och vilka restriktioner som måste beaktas. Detta sker genom att kolla vilka mål och restriktioner som finns för det praktiska problemet som identifierades i föregående delsteg.

**Steg 3**
Designa och implementera en populationsgenereringsalgoritm som består av en lokal sökalgoritm med heuristisk utifrån resultaten från steg 1-3. Designen sker med hjälp av flödesdiagram och pseudokod. Implementationen sker i C#.

**Steg 4**
Testa och utvärdera PGAn genom att använda det praktiska problem som identifierades i steg 2. Testningen sker genom att använda PGAn i olika testfall. Testfallen är beroende av det praktiska problemet men skulle kunna vara att mängden noder varierar och fordonen varierar. Utvärdering sker genom att jämföra resultaten där PGAn användes mot resultaten där en slumpmässig förstapopulation användes på samma testfall. Här är fitness det viktiga utvärdeingskriteriet.

# *4* Genomförande

I detta kapitel beskrivs det praktiska problem som har använts för att utveckla en populationsgenereringsalgoritm (delkapitel 4.1). Vidare så beskrivs optimeringsmålen som algoritmen strävar mot (delkapitel 4.2). Algoritmens design och implementation beskrivs också (delkapitel 4.3 och delkapitel 4.4).

## 4.1 Praktiskt problem

Det praktiska problem som valts är ett problem som Avfallshantering Östra Skaraborg (AÖS) har gällande tömning av sopcontainrar. AÖS ansvarar för sophämtningen i bland andra Skövde kommun där de har ca 180 sopcontainrar. Varje container har en eller flera fasta dagar i ett 14-dagars schema då den töms. Reglerna anger att 14 dagar är en maxgräns för hur länge soporna får ligga innan avfallet börjar lukta för mycket och kan locka till sig ohyra m.m. De flesta containrarna innehåller hushållsavfall som ska till Värmeverket för förbränning och ett mindre antal av containrarna innehåller andra avfallstyper som kompost eller wellpapp. Olika typer av avfall får inte blandas med varandra i sopbilen. Sopbilen har en begränsad kapacitet vad gäller både vikt och volym och när denna kapacitet är uppnådd måste tömning ske. Var tömningen sker beror på avfallstypen.

Containertömningen planeras idag av en transportplanerare som gör en uppskattning på hur ofta varje container ska tömmas och manuellt lägger upp en tömningslista för en 14-dagarsperiod. Ibland kan det förekomma att en container måste tömmas på en annan dag än den ordinarie tömningsdagen. Det kan tillexempel bero på att en storstädning ska ske i ett bostadskvarter och containern då måste vara tom innan detta påbörjas för att inte bli överfull. Transportplaneraren lägger då till den containern till dagens lista med containrar som ska tömmas. Chauffören får sedan listan för dagen och får planera ordningen och körvägen själv. Sådant arbete är svårt att göra manuellt. Dels är det svårt att uppskatta hur ofta eller när en container behöver tömmas, vilket gör att containrarna ofta bara är halvfulla när lastbilen kommer eller att de är överfulla. Det är också svårt att veta var i en rutt containern ska ligga för att inte behöva åka onödigt långt. Detta gör att chauffören troligtvis inte åker en optimal väg eftersom det är svårt för en människa att utföra en ruttoptimering. Är chauffören erfaren kan den utifrån erfarenhet lära sig vilka upplägg som är bättre eller sämre, men är det en ny chaufför så blir allt mycket svårare. Ännu större svårighet skapas av de tidsrestriktioner som måste tas hänsyn till och som innebär att vissa containrar inte får tömmas under vissa tider. Tillexempel får containrar vid dagis inte tömmas när barnen är ute och leker på förmiddagen.

En erfaren chaufför hinner tömma ca 25 containrar under en dag. När en ny eller ovan chaufför kör så hinner han inte med lika många. Det beror på att chauffören bara får en lista i handen med en mängd adresser där han ska tömma containrar. Kommer den nya chauffören inte från trakten säger troligtvis adresserna inte mycket och han måste leta upp varje adress i en kartbok eller liknande om inte någon åker med och visar var alla containrar står. Även om han skulle känna till en del adresser är det svårt att veta vilken ordning som är bra med avseende på tid och sträcka. Chaufförerna har en arbetstid som brukar vara mellan kl. 06:46-16:00 med en lunch kl. 11:45-12:15.

Med bakgrund av de svårigheter som finns så vill AÖS ändra sin sophämtningsrutin så att en ny tömningsrutt kan skapas med hjälp av ett datorprogram varje morgon. Rutten ska överföras till en GPS så chauffören enkelt kan köra rutten optimalt utan att ha kört den förut.

Följande önskemål om den genererade rutten har uttryckts av AÖS:

- Rutten ska tömma så många containrar som möjligt
- Rutten ska innehålla så fulla containrar som möjligt
- Tidsrestriktioner ska beaktas
- Rutten ska vara anpassad för chaufförens arbetsdag dvs. start, slut och raster
- Möjligt att tvinga med en container för tömmning

I kapitel 3.4 (steg 1) ställdes tre krav upp för det praktiska problem som arbetet baseras på: minst 15 noder som ska besökas, begränsad kapacitet hos fordonet och begränsad arbetstid för föraren. Som framgår ovan uppfylls samtliga dessa krav av containertömningsproblemet hos AÖS.

## 4.2 Optimeringsmål

Som beskrevs i föregående delkapitel handlar det praktiska problemet om att en sopbil ska tömma så många och så fulla containrar som möjligt på sin rutt under dagen utan att åka onödigt långa sträckor och med hänsyn till de begräsningar och restriktioner som finns. Containrar kan också bli tvingade att tömmas även om de inte är fulla, vilket som nämnts tidigare till exempel kan bero på att det kommer att ske en storstädning i ett bostadskvarter. Det kan också vara så att containern stått i 14 dagar och då måste tömmas av hygieniska skäl.

Baserat på dessa fakta om problemet har följande mål som populationsgenereringsalgoritmen (PGAn) ska sträva mot identifierats:

- Alla containrar som måste tömmas för dagen ska finnas med i varje lösning.
- Varje rutt ska hinna med så många containrar som möjligt under arbetsdagen
- Varje rutt ska ha så fulla containrar som möjligt
- Rutterna får inte vara för lika varandra. Detta för att den genetiska algoritmen inte ska fastna för lätt i ett lokalt optimum.

Restriktioner som PGAn ska ta hänsyn till är:

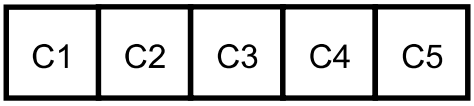
- Rutterna får inte ta längre tid än arbetsdagen att åka
- Olika avfallstyper får inte blandas i sopbilen

För att uppnå målet med att alla containrar som måste tömmas för dagen ska finnas med i varje lösning väljs de containrarna ut i början av PGAns arbete, de får sedan bli basen för varje lösning. För att varje rutt ska bestå av så fulla containrar som möjligt men ändå innehålla variation så slumpas vilka containrar som ska ingå i rutten med en fördel för fullare containrar. Rutten sorteras för att ge en kortare körsträcka vilket gör att fler containrar hinns med. I sortering grupperas även avfallstyperna i rutten eftersom olika avfallstyper inte får blandas i sopbilen.

Arbetet har avgränsats så att PGAn inte tar hänsyn till tidsrestriktioner som finns vid tillexempel dagis eller lunchrestauranger då en bra heuristik för detta är mycket svår att finna. Alternativet skulle i så fall vara att kassera lösningen om den skulle innehålla en bruten tidsrestriktion och generera en ny. Det skulle kunna fungera om det inte finns så många restriktioner att ta hänsyn till, skulle det dock finnas många så skulle PGAn få svårt att hitta giltiga lösningar. Därför överlåter PGAn problemet med tidsrestriktioner till den genetiska algoritmen. Även om lösningen skulle innehålla en bruten tidsrestriktion så kan lösningen innehålla stora delar som är bra och användbara för den genetiska algoritmen.

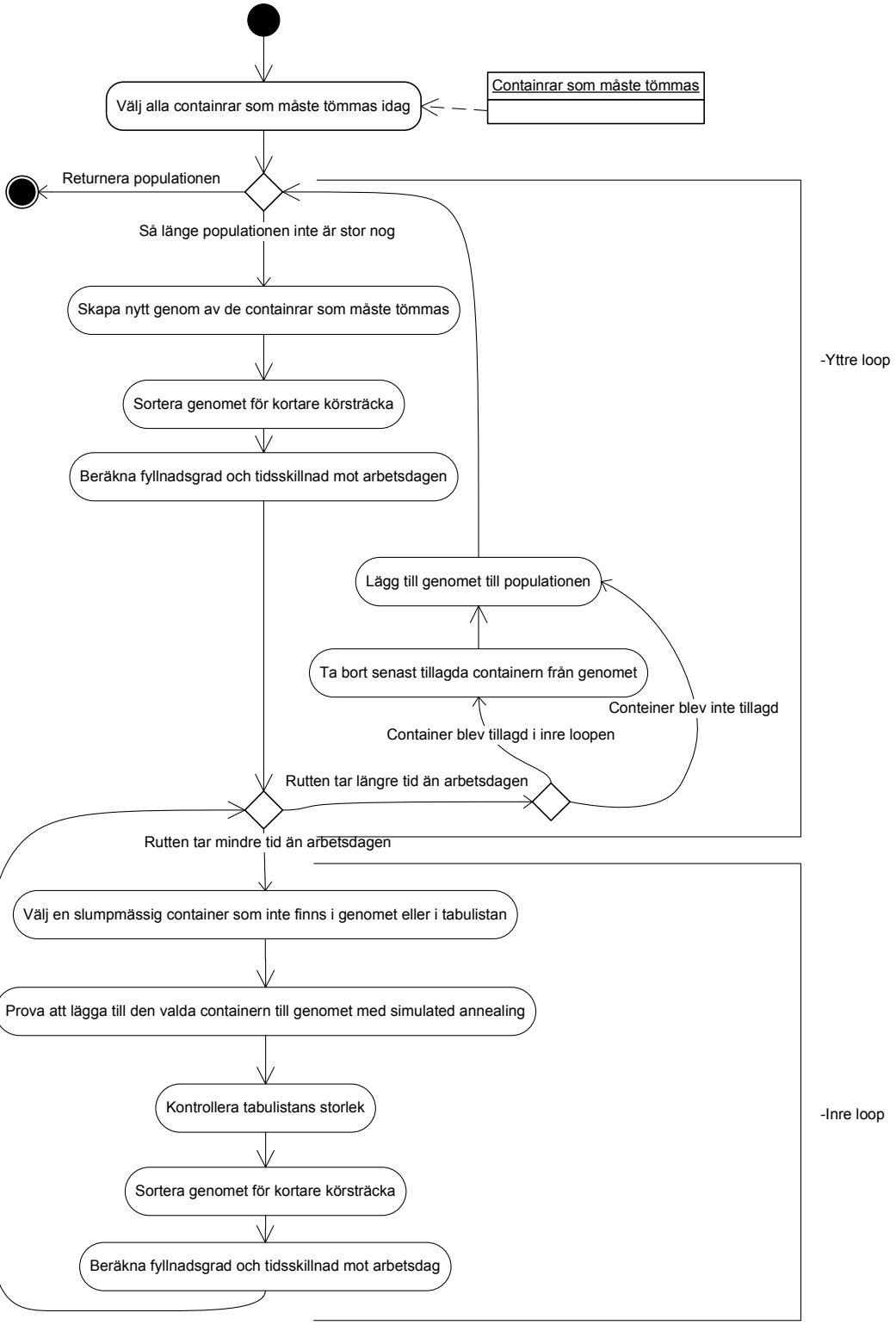
## 4.3 Design

PGAn producerar en population som den genetiska algoritmen kan använda för sina beräkningar. En population består av en mängd lösningar på problemet, i detta fall är varje lösning en rutt. Varje lösning kallas i den genetiska algoritmen för individ eller genom. Nedan används individ och genom likvärdigt för att beskriva en lösning i PGAn. Varje individ i PGAn består av en lista med containrar (se Figur 7). Containrarna ska tömmas i den ordning som de förekommer i lisan. Tömningsplatser som Värmeverket finns inte med i genomet utan läggs in i beräkningarna där de behövs. Varje container består av bland annat en position i longitud och latitud, volym, fyllnadsgrad och avfallstyp.




**Figur 7 - En lista med fem containrar som representerar en lösning.**

Översiktligt så består PGAn av två loopar. En yttre loop som bygger upp populationen med ett varv per genom. Den inre loppen bygger upp genomen genom att försöka lägga till en container per varv. Innan den yttre loopen körs så hämtas alla containrar som måste tömmas för dagen. Eftersom de containrar som måste tömmas ska finnas med i alla rutter kommer de bli basen för alla genom. Se Figur 8 för ett flödesdiagram för PGAn.




**Figur 8 - Flödesdiagram för PGAn.**

När ett nytt genom skapas sorteras det för att få en kortare körsträcka. Eftersom sopbilen inte får blanda olika soptyper är det dumt om soptyperna ligger blandade i rutten eftersom det resulterar i att sopbilen måste åka onödigt många gånger till värmeverket och tippen för att dumpa avfallet. För att spara ytterliggare körsträcka sorteras genomet också så att närliggande containrar töms i samma veva i stället för att åka till nästan samma ställe flera gånger under dagen. Denna sortering sker naivt genom att börja på den först containern och sedan åka till den container som ligger närmast. Den tredje containern blir den som ligger närmast den andra osv. inom varje soptyp. Detta ger nödvändigtvis inte en optimal rutt men gör att de containrar som ligger nära varandra geografiskt hamnar nära varandra i rutten.

När sorteringen är genomförd skickas genomet till en simulator som beräknar fyllnadsgraden på genomet, vilket är snittet på fyllnadsgraden hos de containrar som ingår i rutten. Simulatorn beräknar också skillnaden mellan sluttiden på rutten och sluttiden på arbetsdagen. Skulle rutten sluta redan kl. 14:00 och arbetsdagen slutar kl. 16:00 blir resultatet -120 minuter. Dessa beräkningar sker genom att simulatorn simulerar att en sopbil åker och tömmer de containrar som finns i genomet. Simulatorn beräknar att sopbilen börjar åka från garaget när arbetsdagen börjar och åker och tömmer container efter container tills alla containrar som finns i genomet är tömda. Simulatorn räknar också med resor till Värmeverket och tippen vid behov och resa till och från garaget för lunch. Skulle rutten ta mindre tid än arbetsdagen så går PGAn in i den inre loopen för att lägga till en container, annars läggs genomet till i populationen.

Den inre loopen (se Figur 8) försöker lägga till en container till genomet varje varv tills genomet tar längre tid att köra än arbetsdagen. Först väljs en slumpmässig container ut ur mängden av alla containrar. Det finns två krav på den valda containern. Det första är att den inte redan får finnas i rutten eftersom det är onödigt att tömma samma container två gånger samma dag. Det andra kravet är att den inte får finnas i den så kallade tabulistan, vilket är en lista med containrar som nyligen valts ut för att läggas till. Ligger den i tabulistan betyder det att containern nyss provades och att det är stor chans att den redan ligger i ett genom som nyligen skapats. Tanken med tabulistan är att öka variationen mellan genomen så inte alla genom innehåller samma containrar.

När en ny container är vald jämförs fyllnadsgraden på containern med snittet på genomet. Är den nya containerns fyllnadsgrad högre än snittet läggs containern till i genomet och genomets snitt stiger. Även om genomets snitt skulle vara högre än den valda containerns fyllnadsgrad finns det en chans att containern läggs till, fast den skulle sänka snittet. Om den läggs till eller inte avgörs med simulated annealing. I simulated annealing finns en variabel som kallas för temperatur. Om temperaturen är hög är chansen lika stor att containern läggs till som att den inte gör det. Är temperaturen låg är det lägre chans att en container med låg fyllnadsgrad läggs till. Temperaturen sjunker lite efter varje försök att lägga till en container vilket gör att det blir svårare och svårare att lägga till en container med låg fyllnadsgrad som försämrar fyllnadsgraden i genomet.

Efter ett försök att lägga till en container kontrolleras tabulistans storlek så den inte är för stor. Är tabulistan för stor tar det lång tid att hitta en ny container och skulle den bli så stor att alla containrar får plats i tabulistan går det inte välja någon ny container

överhuvudtaget. Är listan för stor plockas den container bort som legat längst i listan. Hur stor listan får vara är upp till användaren att definiera.

När en ny container lagts till genomet så sorteras det igen för att hitta en bra placering för den nya containern. Tidsskillnaden mellan ruttens sluttid och arbetsdagens sluttid beräknas liksom snittet på fyllnadsgraden med hjälp av simulatorn. Den inre loopen har nu gått ett varv. Är det så att genomet tar mindre tid än en arbetsdag tar loopen ett varv till annars avbryter den och den senast tillagda containern plockas bort innan genomet läggs till i populationen. Den yttre loopen har då också gått ett varv. När populationen är stor nog avbryts genereringen och den färdiga populationen returneras.

## 4.4 Implementation

PGAn följer stegen i Figur 9 där steg 1.0 är ett första steg där alla containrar som måste tömmas för dagen väljs ut från mängden av alla containrar som rutten kan skapas av. De som väljs ut i detta steg blir basen för dagens rutt. Implementationen är skriven i C#.net 3.5 med hjälp av Microsoft Visual Studio 2008

**1.0** Välj alla containrar som måste tömmas idag.
**2.0** Så länge populationen inte är stor nog.
- **2.1** Skapa ett nytt genom av de utvalda från 1.0.
- **2.2** Sortera genomet för kortare rutt
- **2.3** Beräkna snittfyllnadsgraden i genomet
- **2.4** Beräkna hur lång tid rutten tar att köra, inklusive ev. raster m.m. Beräkna sedan skillnaden mellan ruttens slut och arbetsdagens slut.
- **2.5** Så länge som rutten tar mindre tid än en arbetsdag att köra
  - **2.5.1** Välj en slumpmässig container som inte finns i genomet eller i tabulistan. Lägg till den valda containern till tabulistan.
  - **2.5.2** Prova att lägga till den genom simulated annealing baserad på fyllnadsgraden.
  - **2.5.3** Kontrollera tabulistans storlek så den inte är för stor
  - **2.5.4** Sortera genomet så den tar kortare tid att köra
  - **2.5.5** Beräkna fyllnadsgrad på genomet
  - **2.5.6** Beräkna tidskillnaden mellan ruttens och arbetsdagens slut och gå sedan till 2.5
- **2.6** Om en container blev tillagd inne i 2.5-loopen tas den senast tillagda containern bort.
- **2.7** Lägg till genomet till populationen och gå sedan till 2.0

**3.0** Populationen är stor nog och är klar att returneras till den genetiska algoritmen.

**Figur 9 - Beskrivning av designen.**

Steg 2.0 är den yttre loopen som håller på så länge populationen inte är stor nog (se Kod 1). Steg 2.1 är det första steget för att skapa ett nytt genom där de containrar som plockats ut i steg 1.0 skapar ett nytt genom.

```
Population GeneratePopulation(int popSize)
{
  Population pop = new Population();
  Queue tabuList = new Queue();

  //steg 1.0
  Genome baseGenome = GetAllContainersThatNeedCollectionToday()

  //steg 2.0
  while(pop.size < popSize)
  {
    Genome genome = baseGenome;
    //...
    pop.Add(genome);
  }
  return pop;
}
```

**Kod 1 - Yttre loopen.**

PGAn använder en naivsortering som sorterar containrarna inom soptyperna så att de containrar som ligger nära varandra geografiskt hamnar nära varandra i listan (se Kod 2). I denna implementation hämtas avståndet mellan containrarna från en databas, i annat fall skulle euklidiskt avstånd kunna beräknas utifrån containrarnas position. Sorteringen sker genom att den första containern behåller sin plats i genomet. Den container som hamnar på plats numer två är den container som har kortast geografiskt avstånd från den som ligger på plats ett. Den container som hamnar på plats tre i genomet är i sin tur den som ligger närmast den på plats två osv. Efter sorteringen används simulering för att beräkna snittfyllnadsgraden i genomet och tidsskillnaden mellan ruttens slut och arbetsdagens slut (steg 2.3 och 2.4 i Figur 9).

```
Genome NaiveSort(Genome genome)
{
  Genome newGenome = new Genome();
  Container from = genome[0];
  newGenome.Add(from);
  genome.Remove(from);
  int genomeSize = genome.Size;

  for(int i = 0; i < genomeSize; i++)
  {
    //väljer den container i genome som ligger närmast from
    from = PickClosestContainer(genome, from);
    newGenome.Add(from);
    genome.Remove(from);
  }
  return newGenome;
}
```

**Kod 2 - Naiv sortering.**

Steg 2.5 i Figur 9 (Kod 3) är en loop som snurrar så länge som rutten som skapas tar mindre tid att köra än arbetsdagen. Det är denna loop som bygger upp rutten. Först (steg 2.5.1 i Figur 9) väljs en container ut bland mängden av alla containrar som rutten ska byggas av. Har den valda containern högre fyllnadsgrad än snittet i genomet läggs den till genomet, har den lägre fyllnadsgrad än snittet i genomet används simulated annealing för att se om den läggs till ändå (Kod 4).

```
//...
//steg 2.5-inre loopen
//så länge rutten tar mindre tid än arbetsdagen,
//lägg till en container
while(timeDiff < 0)
{
  //steg 2.5.1
  //hämtar en slumpmässig container som
  //inte finns i genome eller i tabuQueue
  Container container = GetRandomContainer(genome, tabuQueue);
  tabuQueue.Enqueue(container);

  //steg 2.5.2
  //prova att lägga till container till genome
  //...

   //steg 2.5.3 kontrollera tabulistans storlek
  CheckSize(tabuQueue);

  //steg 2.5.4 sortera genomet
  Sort(genome);

  //steg 2.5.5
  //...
}
```

**Kod 3 - Inre loopen.**

```
//...
//steg 2.5.2
//prova att lägga till container till genome
if(container.Fullness > genome.AverageFullness)
{
  genome.Add(container);
  containerWasAdded = true;
}
else
{
  double rdouble = random.NextDouble();
  double diff = container.averageFullness - container.Fullness;
  double p = Math.Exp(diff / T);
  p=1.0 / (p+1.0);
  if(rdouble < p)
  {
    genome.Add(container);
    containerWasAdded = true;
  }
  T = T*r;
}
//...
```

**Kod 4 - Den inre lopen försöker lägga till en container till genomet.**

I steg 2.5.4 i Figur 9 sorteras genomet igen. Det är samma sorts sortering som sker i steg 2.2. Efter sorteringen beräknas snittet på fyllnadsgraden och skillnaden på ruttens sluttid och arbetsdagens sluttid (steg 2.5.5 och 2.5.6 i Figur 9) precis som i steg 2.3 och 2.4. Är tidskillnaden negativ så betyder det alltså att rutten är slut före arbetsdagens slut och då går loopen till steg 2.5 igen för att lägga till en ny container. Skulle rutten ta längre tid än arbetsdagen avbryts den inre loopen och den senast tillagda containern tas bort från genomet innan genomet lägg till i populationen som i Kod 5. Det kan vara så att genomet är för långt redan med de containrar som måste tömmas för dagen. Då tas inte någon container bort eftersom alla måste tömmas. Finns det sedan tillräckligt med individer avbryts PGAn och populationen returneras. Finns inte tillräckligt med individer hoppar flödet till 2.0 igen för att skapa en ny individ.

```
//...
//steg 2.6
//tabort den senast tillagda containern från genomet
if(containerWasAdded)
{
  Container latest = tabuQueue.Last;
  genome.Remove(latest);
  Sort(genome);
}

//steg 2.7
population.Add(Genome);
//...
```

**Kod 5 - Borttagning av den senast tillagda containern från genomet.**

# 5 Utvärdering

Detta kapitel beskriver hur populationsgenereringslagoritmen (PGAn) som presenterades i föregående kapitel har utvärderats. Utvärderingen har skett med hjälp av det praktiska problem hos Avfallshantering Östra Skaraborg (AÖS) som presenterades i kapitel 4.1. För utvärderingen har tre mängder med containrar valts innehållande 39, 133 och 180 containrar. De två första mängderna har valts utifrån riktiga distrikt som AÖS använder och den tredje mängden är samtliga containrar som AÖS har i Skövde.

PGAn har använts för att skapa startpopulationer till en genetisk algoritm, som sedan körts på de tre mängderna för att optimera rutter. För varje generation av den genetiska algoritmen har de bästa och sämsta värdena på intressanta parametrar sparats, samt snittet av alla lösningar. Intressanta parametrar inkluderar antal containrar i rutten, fyllnadsgraden, tidskillnaden mot arbetsdagen och fitnessvärdet på genomet. För att slumpen inte ska avgöra resultatet så har den genetiska algoritmen körts 25 gånger på varje mängd containrar och snittet av dessa körningar har sedan använts i utvärderingen.

Eftersom fyllnadsgraderna på containrarna spelar roll för resultatet har ytterligare 25 gånger har körts på varje mängd containrar där containrarna har fått andra fyllnadsgrader. I båda fallen har containrarna fått slumpmässiga fyllnadsgrader mellan 0 och 1 där 0 betyder att containern är tom och 1 att den är full. De båda fallen har jämförts med varandra och tagits med i utvärderingen.

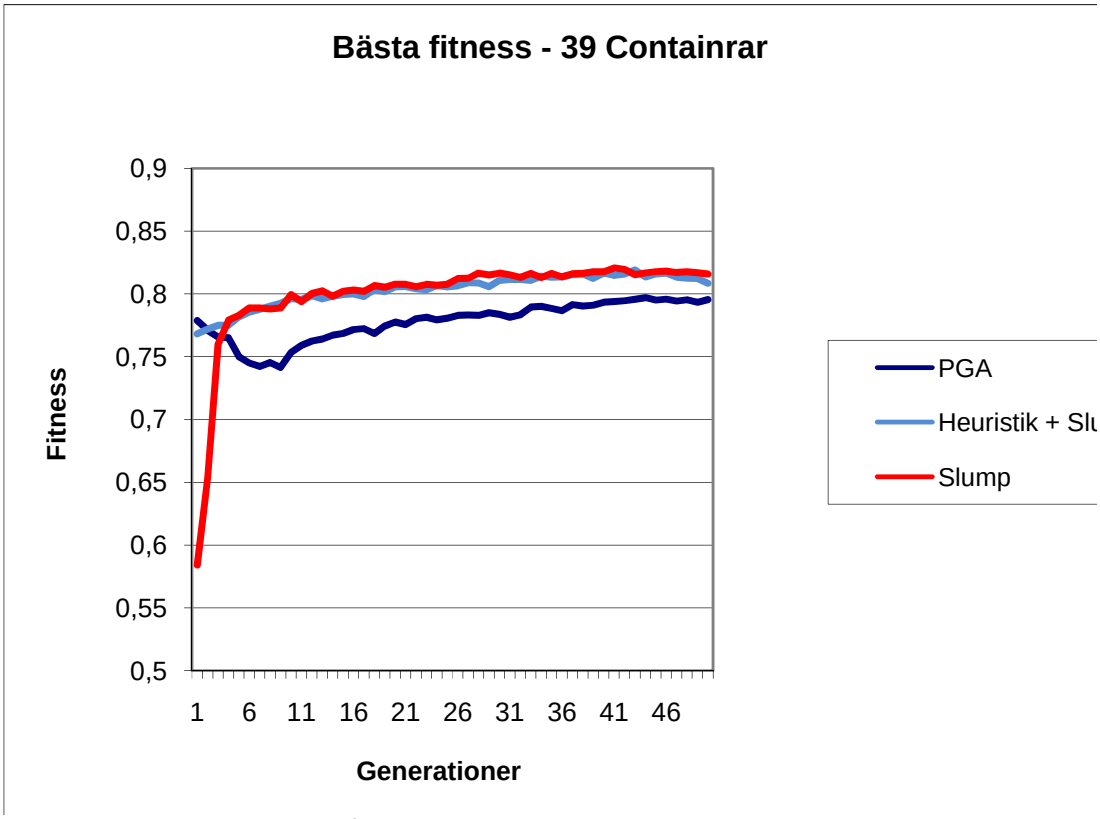
Den genetiska algoritmen har testats med tre olika sorters startpopulationer. Den första populationen är skapad av PGAn som har som målsättning att skapa en så bra första population som möjligt. Den andra populationen är skapad helt slumpmässigt. Detta har gjorts genom att först slumpa hur många gener genomet ska ha och sedan slumpa varje container som ska läggas till. Den tredje varianten av startpopulation är skapad med en blandning av heuristik och slump, mestadels det sistnämnda. För den population bestäms längden på genomet av hur lång arbetsdagen är. En slumpmässig container läggs till i genomet tills rutten tar längre tid än en arbetsdag. Då tas den senast tillagda containern bort och genomet är klart.

Prestandan hos PGAn avgörs genom att studera slutresultatet för den optimerade rutt som den genetiska algoritmen har tagit fram. Den genetiska algoritmen evolverar fram en optimerad rutt genom att beräkna en viktad summa av de olika optimeringsmålen. Desto högre summa, så kallad fitness, som genomet får desto bättre rutt. I beräkning av fitness räknar den genetiska algoritmen in antal containrar i rutten, fyllnadsgraden för rutten och brott mot tidsrestriktioner. Desto längre tid rutten tar jämfört med arbetsdagen desto hårdare straffas rutten. En optimal rutt enligt den genetiska algoritmen har ett fitness på 1.0 och en riktigt dålig rutt har en fitness på 0.0. I utvärderingen har den genetiska algoritmen använt sig av 100 individer och kört 50 generationer.

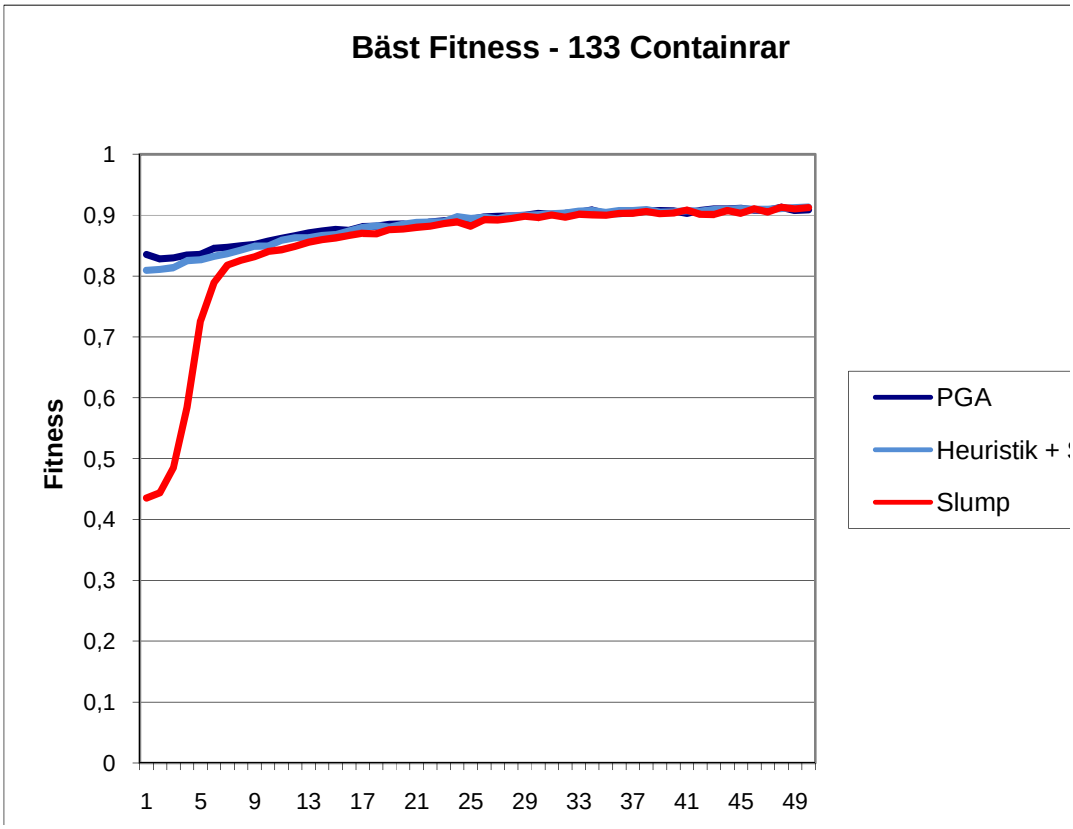
## 5.1 Resultat

I Figur 10 - Figur 12 visas resultat från utvärderingen med de tre olika containermängderna. Resultaten visar att PGAn gör mest nytta för stora mängder med containrar och får svårare i mindre mängder. Exempel kan ses i Figur 10 där en rutt skapas bland 39 containrar. Där får PGAn en bra start men har svårt att förbättras. Den helt slumpmässiga populationen får en dålig start men jobbar snabbt ikapp och går om PGAn. Bäst i detta fall blev den population som kombinerade heuristik och slump.

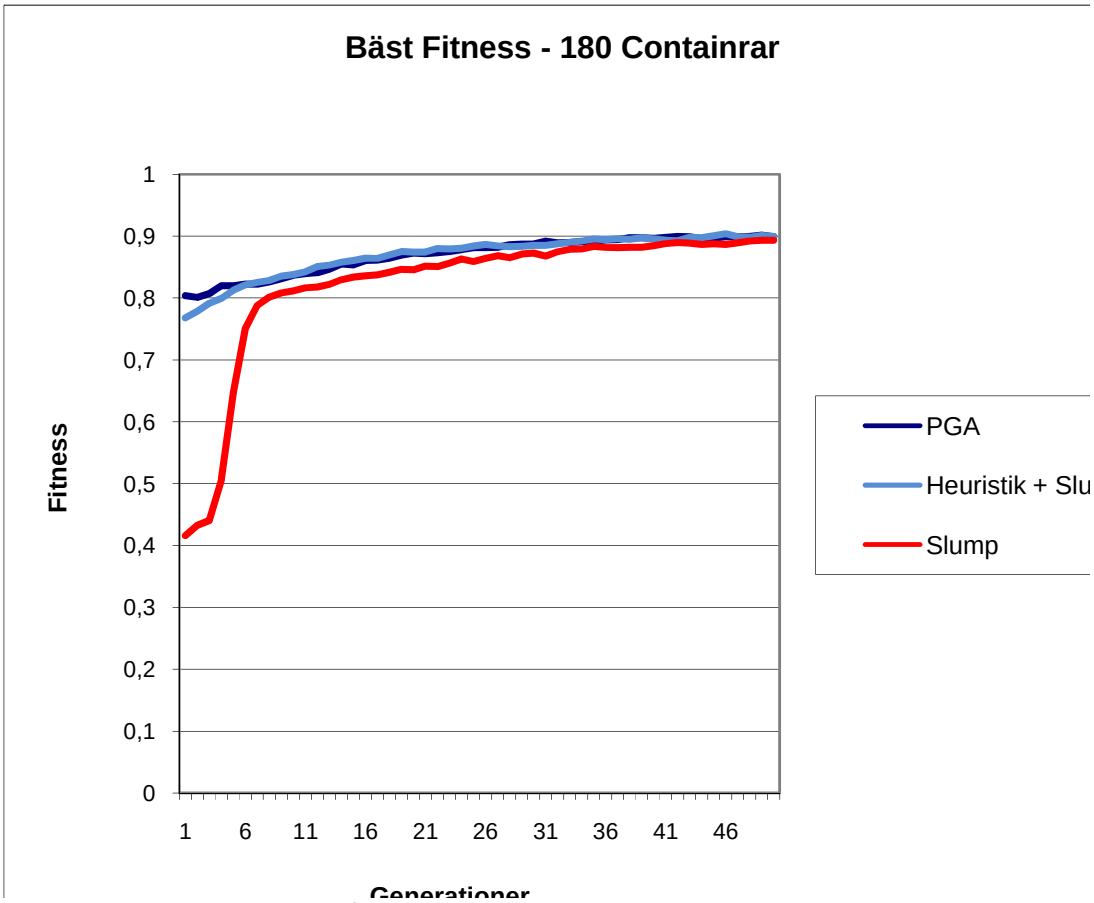
I Figur 12 syns att PGAn presterar bra när det är en större mängd containrar som rutten kan bestå av. Då får den en bättre start än både den helt slumpade populationen och den som kombinerar slump och heuristik. Efter några generationer kommer dock sistnämnda i kapp och slåss om förstaplatsen.




**Figur 10 – Generationens bästa fitness i mängden med 39 containrar.**

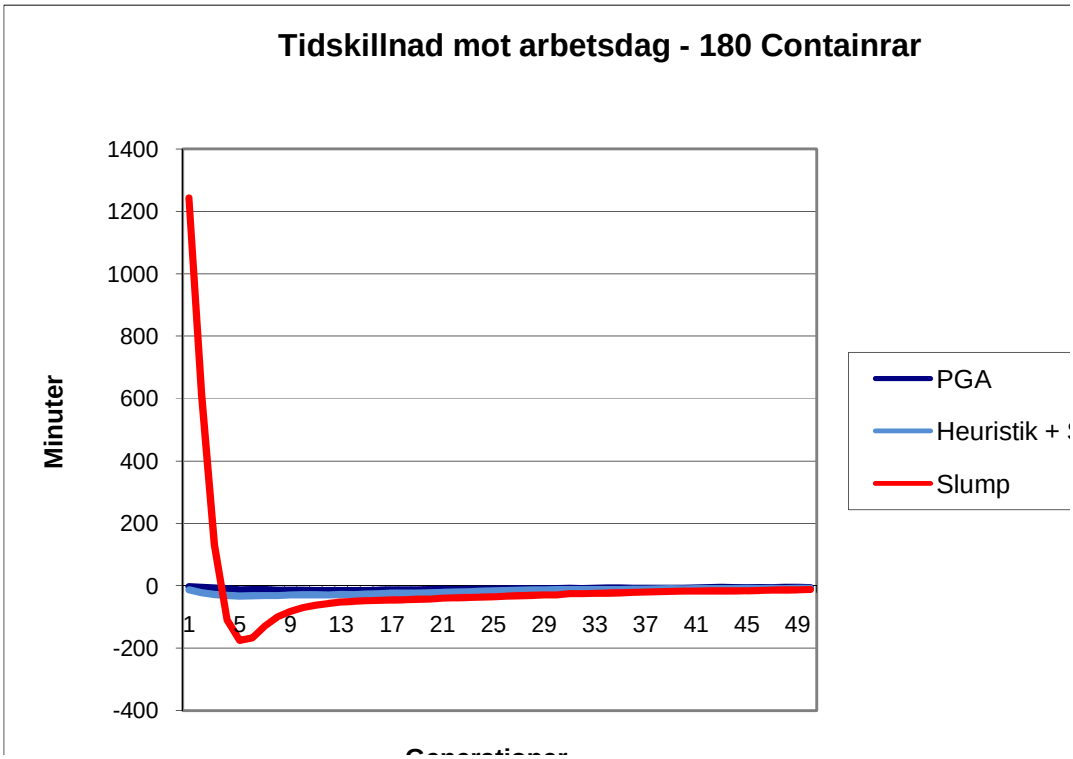



**Figur 11 – Generationens bästa fitness i mängden med 133 cotnainrar**



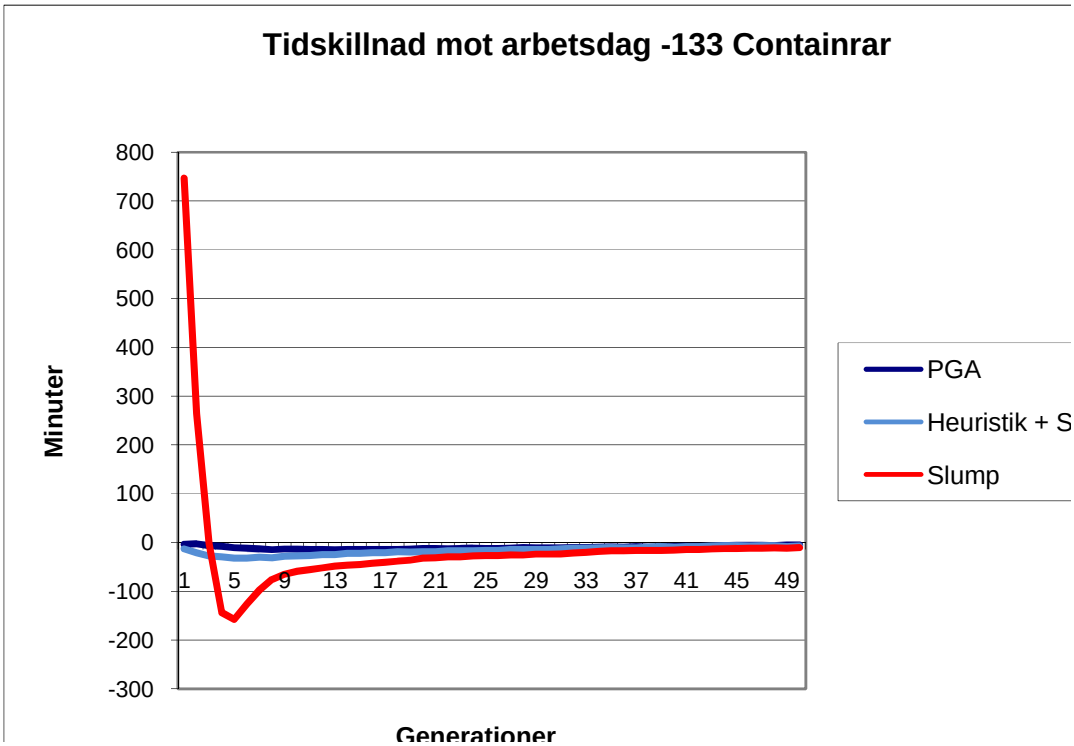

**Figur 12 – Generationens bästa fitness i mängden med 180 containrar.**

Det som gör att den helt slumpade populationen får en dålig start vid stora mängder containrar är att genomet kan ha samma längd som antalet containrar i mängden. Om det finns 180 containrar i mängden kan genomet ha en storlek på 180 containrar. Genom med så många containrar får dålig fitness eftersom arbetstiden då blir mycket längre än den ska. Den population som skapas med PGAn och den som blandar heuristik och slump tar bort de mest extrema genomlängderna och hamnar mycket närmare verkliga rutters längd. I Figur 13 visas hur många minuter en rutt i snitt är fel jämfört mot arbetsdagen för mängden med 180 containrar. En rutt som tar nästa ett dygn för lång tid att köra ger dålig fitness.

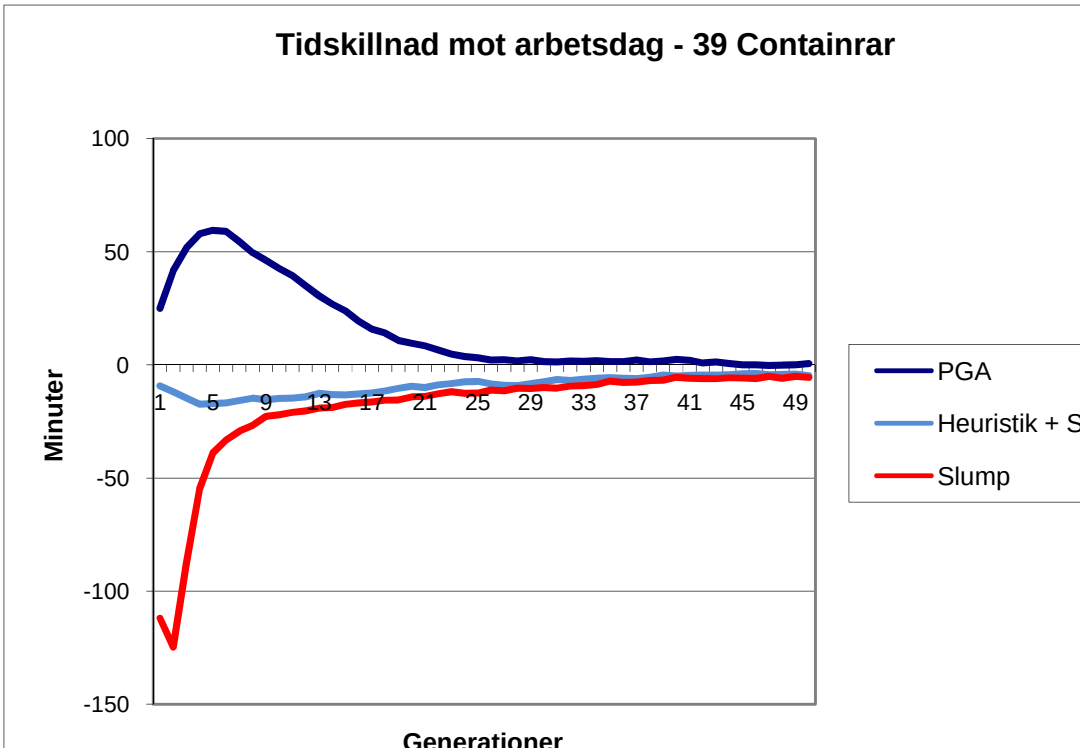



**Figur 13 - Generationernas snitt av tidskillnaden mot arbetsdagen, 180 containrar.**

När det är mindre antal containrar i mängden så kan rutterna inte bli lika långa och risken för dålig fintess minskar, men fenomenet finns kvar. I Figur 14 blir det lite bättre för den helt slumpade generationen men i Figur 15 ses hur tidskillnaden hamnar under 0. Det beror på att risken att genomet blir för kort har ökat. I Figur 15 ses också hur populationen som PGAn har skapat har svårt att komma inom arbetsdagen. Det bör påpekas att Figur 13 visar snittet på alla rutter i populationen, så det är troligt att det finns flertalet rutter som matchar arbetsdagen väl, men snittet ligger lite ovanför.




**Figur 14 - Generationernas snitt av tidskillnaden mot arbetsdagen, 133 containrar.**




**Figur 15 – Generationernas snitt av tidskillnaden mot arbetsdagen, 39 containrar.**

Samtliga resultat visar att PGAn alltid ger en bättre start än den helt slumpmässiga populationen. Dock kan PGAn göra att skillnaden på generna blir för små när mängden containrar är liten, vilket är till nackdel för den genetiska algoritmen, desto större mängd containrar desto mer skillnad gör PGAn. Detta innebär att om utvärderingen av genomet är tidskrävande så att bara några få generationer i den genetiska algoritmen hinns med så gör PGAn stor skillnad, men desto fler generationer som körs desto mindre skillnad gör PGAn. Den population som skapas med hjälp av en kombination av heuristik och slump klarar sig relativt bra i alla testfallen. Inte alltid bäst, men inte heller sämst.

En aspekt som troligtvis påverkar resultaten väsentligt är hur den genetiska algoritmen fungerar, dvs. vilka mutationsoperatorer och parningsoperatorer som den använder. I denna studie har relativt lite fokus lagts på den genetiska algoritmen och bättre resultat går troligtvis att uppnå om den genetiska algoritmen och PGAn anpassas till varandra på ett bättre sätt.

## 5.2 Analys

Utvärderingen av PGAn visar att den ger den genetiska algoritmen en bättre start jämfört med en helt slumpmässig population om varje generations bästa individ beaktas. Dock är utvecklingen hos populationen beroende på hur stor mängden med containrar är. Utvärderingen visade att desto större mängd containrar som genomen tilläts bestå av desto större nytta gjorde PGAn jämfört med en helt slumpmässig population. I den minsta mängden med containrar som testades (39 stycken) resulterade den helt slumpmässiga populationen i bättre slutresultat hos den genetiska algoritmen än populationen från PGAn efter 50 generationer. Vid den största mängden containrar (180 stycken) blev slutresultatet hos den genetiska algoritmen något bättre när den använde en population från PGAn jämfört med en slumpmässig population. I mängden med 133 stycken containrar blev slutresultatet i stort sett lika för alla tre populationerna. Den population som var skapad med slump och heuristik var inte lika bra i starten som populationen från PGAn men den kom ikapp snabbare än den helt slumpmässiga populationen.

Utvärderingens utfall beror troligtvis på att PGAn skapade genom med högre fitness än de två andra metoderna genom att bland annat välja bort containrar med låg fyllnadsgrad. Detta är dock inte enbart en fördel – det kan också bli till en nackdel eftersom variationen blir mindre vilket kan göra att det blir svårare för populationen att ta sig förbi lokala optima. Detta märks tydligare när det är mindre mängder containrar som genomen kan bestå av. När det finns en stor mängd containrar att välja bland så blir variationen stor ändå och slutresultatet blir bättre med PGAn än med en helt slumpad population. Den naiva sorteringen som sker inom rutten gör också att individerna blir lika varandra om få containrar finns i mängden.

Det bör beaktas att den genetiska algoritmen kan ha många olika inställningar såsom exempelvis olika mutering- och parningsoperatorer, vilka kan resultera i ett annat beteende och andra resultat. Det är tänkbart att vissa inställningar kan göra att PGAn ger dåliga förutsättningar medan andra kan göra att den blir bättre. Även antalet individer i populationen spelar troligtvis roll. I utvärderingen användes 100 individer och resultatet visade sig bli sämre när det var fler individer än det fanns containrar. En tänkbar förklaring till detta är att PGAn gör störst nytta när de viktigaste containrarna kan plockas

ut ur en stor mängd contrainrar, men om de containrarna ändå kommer med i de flesta individerna gör PGAn snarast skada genom att minska variationen hos individerna.

# 6 Slutsatser

Detta kapitel innehåller en sammanfattning av arbetet i delkaptiel 6.1. En diskussion kring resultatet finns i delkapitel 6.2 och tankar kring framtida arbete i delkapitel 6.3.

## 6.1 Sammanfattning

Detta arbete har hittat ett nytt sätt att använda lokala sökalgoritmer med heuristik för att skapa den första populationen i en genetisk algoritm med flera mål för att lösa en variant av vehicle routing problemet. Ett praktiskt problem har hittats där en sopbil ska få en optimal rutt med containrar för tömning varje dag. Målen som den genetiska algoritmen strävar mot är att rutten ska ha så många och så fulla containrar som möjligt och att rutten ska hinnas med under arbetsdagen. Implementationen använder sig av simulated annealing och tabu search för att bygga upp individerna (rutterna).

Utvärderingen visar att algoritmen hjälper den genetiska algoritmen att prestera bättre vid komplexa problem. Vid utvärderingens minsta containermängd med 39 stycken containrar, blir slutresultatet bättre när en helt slumpartad population används. Men vid de två andra containermängderna, 133 och 180 stycken, blir resultatet bättre med den presenterade algoritmen.

## 6.2 Diskussion

Vilka inställningar som är optimala för den genetiska algoritmen har inte undersökts i detta arbete. Då inställningar som tabulistans storlek och temperaturen i simulated annealing delen påverkar individernas utformning är detta något som bör beaktas. Under utvärderingen var tabulistans storlek satt till 0.7 gånger mängden av alla containrar. Finns det exempelvis 100 containrar att tömma så får 70 containrar plats i tabulistan samtidigt. Tabulistans storlek är dock ett värde som tagits fram enbart baserat på känsla och enstaka experiment och det är troligt att ett bättre värde skulle kunna hittas. En ökad storlek på tabulistan skulle tillexempel kunna leda till en ännu bättre variation. I simulated annealing-delen bestämmer temperaturen hur stor roll fyllnadsgraden har. Det kan vara så att temperaturen är för hög som det är nu och därför bör sänkas för att få bättre fyllnadsgrad på rutterna. Dock är det så att om temperaturen är *för* låg så kan variationen hos individerna bli för dålig så att den genetiska algoritmen inte presterar lika bra.

Som PGAn har implementerats nu sker en naiv sortering av containrarna hos varje individ. En nackdel med implementationen av denna sortering är att den alltid utgår från den första containern i genomet. Det gör att den första containern inte kommer att byta plats under uppbyggnaden av individen. Detta skapar problem om det finns containrar som är tvingade att tömmas för dagen eftersom de alltid väljs ut först som en bas. Om så är fallet kommer det att bli samma container som hamnar först i alla genomen. En möjlig förbättring av PGAn är att den naiva sorteringen utgår från en slumpmässig container i stället för den som ligger först i genomet.

## 6.3 Framtida arbete

En aspekt som kan vara intressant att undersöka i framtiden är hur den framtagna PGAn fungerar med en genetisk algoritm som använder sig av så kallad Pareto-optimering i stället för en målfunktion bestående av en viktad summa. I detta arbete har fokus enbart varit på det sistnämnda. I en genetisk algoritm som använder sig av Pareto-optimering värderas varje parameter, till exempel antal containrar och fyllnadsgrad, var för sig i stället för att viktas ihop till ett enda värde. När en Pareto-optimering sker strävar den genetiska algoritmen efter att ha en högre grad av variation under alla generationer. För att möta detta behov skulle en variant av PGAn kunna utvecklas som strävar mot de olika målen med olika individer. Nu strävar PGAn mot alla målen samtidigt med alla individer, vilket skapar en något mer homogen population.

En intressant fortsättning av detta arbete är att undersöka hur PGAn presterar i andra praktiska problem. I detta arbete har containertömning på dagsbasis varit mål för problemet, men det är mycket möjligt att PGAn går att använda även i andra problemdomäner. Eftersom vissa parametrar går att variera i PGAn går det styra hur första populationen ska utformas. Om PGAn används för generering av artificiell intelligens i spel skulle PGAn kunna hjälpa till att styra evolveringen av olika personligheter. Om utformningen av en industriellprodukt ska utvecklas kan PGAn hjälpa till att styra mot de olika målen som finns i de fallen (vikt, form, pris osv.). För att kunna applicera PGAn på andra typer av problem än det som presenterades i detta arbete krävs troligen vissa ändringar i implementationen men PGAn kan fungera som en grund för dessa problem.

# Referenser

Aarts, E. H. L., och Lenstra, J. K. (1997) *Local search in combinatorial optimization.* Chichester: John Wiley & sons, Ltd.

Chinneck J. W. (2006) *Practical optimization: a gentle* introduction. Carleton University. Tillgänglig på Internet: http://www.sce.carleton.ca/faculty/chinneck/po.html [hämtad: 09.02.19].

Dablanc, L. (2007) Goods transport in large European cities: difficult to organize, difficult to mordernize. *Transportation Research, 41(A),* 280-285.

Deb, K. (2001) *Multi-objective optimization using evolutionary algorithms.* Chichester: John Wiley & sons, Ltd.

Grefenstette, J. J. (1987). *Genetic Algorithms and Simulated Annealing,* London: Pitman.

Hillam, B. P. (2003) *Introduction to Algorithms: a java based approach*. California state polytechnic university. Tillgänglig på Internet: http://www.csupomona.edu/~bphillam/algotext/ [09.02.25].

Holland, J. H. (1992) *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control and artificial intelligence* (2:a upplagan). Cambridge: MIT Press.

Johnson, D. S., (1990) *Local optimization and the traveling salesman problem.* Proceedings of the 17th International colloquium on automata, languages and programming.

Jozefowiez, N., Semet, F. och Talbi E.G. (2008) Mulit-objective vehicle routing problems. *European journal of operational research, 189,* 293-309.

Kang, K. H., Lee, B. K., Lee, Y. H., och Lee, Y. H. (2008) A heuristic for the routing problem with due times. *Computers & industrial engineering, 58,* 421-431.

Louis, S. J. och Li, G. (2000) Case injected genetic algorithms for traveling salesman problems. Information sciences, 122, 201-225.

Michalewicz, Z. och Fogel, D. B. (2000) *How to solve it: modern heuristics* (2:a upplagan). Berlin: Springer-Verlag.

Santos, L., Coutinho-Rodrigues, J. och Current, J. R. (2008) Implementing a multi-vehicle multi-rout spatial decision support system for efficient trash collection in Portugal. *Transpotation research, 42(A),* 922-934.