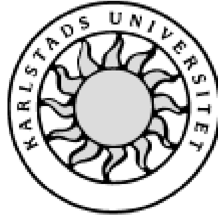


Datavetenskap

Mathias Johansson, Mikael Lindberg

Evolutionär beräkning kontra klassisk optimering

Examensarbete, C-nivå
2004:01

Evolutionär beräkning kontra klassisk optimering

Mathias Johansson, Mikael Lindberg

© 2004 Mathias Johansson, Mikael Lindberg och Karlstads universitet

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Mathias Johansson, Mikael Lindberg

Godkänd, 2004-01-15

Handledare: Reine Lundin

Examinator: Stefan Lindskog

Sammanfattning

Människan har i alla tider strävat efter att lösa de problem som hon ställs inför. Då forntidens problem kanske ofta handlade om att skapa verktyg för att underlätta livet, handlar det idag ofta om att lösa olika typer av beräkningsproblem eller ställa olika typer av diagnoser. Medan antalet påverkande faktorer ökar linjärt i ett problem tenderar antalet möjliga lösningar ofta växa exponentiellt. Detta implicerar i sin tur att det ofta inte ens är möjligt att lösa ett problem exakt inom en rimlig tidsram och det är här optimering kommer in i bilden. Även ett till synes enkelt problem kan resultera i en sökrymd så stor att det kommer ta flera miljarder år att ens genomsöka en bråkdel av den med hjälp av traditionella metoder. För att garantera en optimal lösning finns det nämligen inget annat möjligt tillvägagångssätt än att genomsöka tillståndsrymden i dess helhet. Denna uppsats syftar till att ge en introduktion till vanliga tekniker för optimering med hjälp av heuristik och genetiska algoritmer. Vi kommer att visa att det vid användandet av någon av dessa tekniker är möjligt att hitta en nära optimal lösning inom loppet av några sekunder, beroende på problemets komplexitet. Användandet av någon av dessa tekniker kan i många fall vara helt nödvändigt för att ens kunna hitta en lösning inom en rimlig tidsram.

Optimering är ett väldigt stort område som inte bara berör datavetenskap, och det går inte att täcka hela detta område i en uppsats utan att endast skrapa på ytan. Denna uppsats syftar till att öppna dörren och ta första steget in till denna intressanta värld som många gånger styrs med hjälp av slumpen.

Evolutionary computing vs classical optimization

Abstract

The human being has since the beginning of time always strived to solve the problems she encounters. While the earlier problems often were about creating tools to make everyday life easier, the problems of today tends to be to solve different kinds of calculations or make different kinds of diagnosises. When the number of factors affecting the problem increases linearly, the number of possible solutions tends to increase exponentially. This in turn implies that in many cases it isn't even possible to solve a problem exactly within a reasonable time frame, and that's why the use of optimization comes into the picture. Even a problem that appears to be fairly simple might lead to a search space so large that it will take several billions of years to search through even a portion of it using traditional methods. To guarantee an optimal solution there is simply no other way than to search through the search space in its entirety. This dissertation aims to give an introduction to common heuristic and genetic optimization techniques. We are going to show that by using any of these techniques makes it possible to find a near optimal solution within a few seconds, depending on the complexity of the problem. The use of these techniques can in many cases be a necessity for finding a solution at all within a reasonable time frame.

Optimization in general is a very large area that not only involves computer science, and it is not possible to cover all of it in a dissertation like this. This dissertation aims to open the door and take the first step into this interesting world that many times is ruled by randomness.

Innehållsförteckning

1 Inledning **1**

2 En kort historisk introduktion **3**

3 Problemlösning med sökning **4**

3.1 Blinda sökningar 5
3.1.1 Depth-first
3.1.2 Breadth-first
3.1.3 Iterative deepening

3.2 Sökning med heuristik 9
3.2.1 Heuristisk funktion
3.2.2 Evalueringsfunktion
3.2.3 Greedy best-first
3.2.4 A*

4 Klassiska problem **16**

4.1 The Traveling Salesman Problem 16

4.2 8-queen problem 18

4.3 Satisfiability problem 19

5 Optimering med heuristik **21**

5.1 Hill-climbing 22
5.1.1 Local beam
5.1.2 Stochastic beam

5.2 Simulated annealing 24

5.3 Tabu search 26

6 Optimering med genetiska algoritmer **29**

6.1 Representation 30

6.2 Population 32

6.3 Lämplighetsfunktion 32

6.4 Urvalsoperatorer 33
6.4.1 Slumpmässigt
6.4.2 Proportionellt
6.4.3 Turnering
6.4.4 Rankbaserat
6.4.5 Elitiskt

6.5 Reproduktionsoperatorer 36
6.5.1 Cross-over
6.5.2 Mutation
6.6 Exempel 39
7 Experiment 43
7.1 Problemdefinition 43
7.2 Implementation 44
7.2.1 Simulated Annealing
7.2.2 Genetisk algoritm
7.3 Experimentutförande 49
7.4 Resultat 50
7.5 Diskussion 54
8 Sammanfattande kommentarer 56
Referenser 57

Figurförteckning

Figur 3.1: Hitta en väg från A till D. 4

Figur 3.2: Pre-order depth-first search traversering. 6

Figur 3.3: In-order depth-first search traversering. 6

Figur 3.4: Post-order depth-first search traversering...7

Figur 3.5: Breadth-first traversering..7

Figur 3.6: Iterative deepening search med pre-order evaluering. 9

Figur 3.7: 8-puzzle startposition. 11

Figur 3.8: 8-puzzle målposition. 11

Figur 3.9: Förenklad vägkarta över en del av nordöstra Mongoliet, sträckorna är uppskattade och angivna i km. 12

Figur 3.10: Greedy best-first applicerat på problemet att hitta en väg från Uuldza till Dashbalbar (se karta i Figur 3.9) 13

Figur 3.11: A* applicerat på problemet att hitta en väg från Uuldza till Dashbalbar (se karta i Figur 3.9) 15

Figur 4.1: TSP med fem städer. Den kortaste (optimala) vägen är markerad med feta linjer. 18

Figur 4.2: En av de 92 lösningarna till 8-drottningsproblemet. 19

Figur 5.1: Exempel på en en-dimensionell tillståndsrymd där höjden motsvarar värden på den objektiva funktionen. Globalt maxima innebär den optimala lösningen. 22

Figur 5.2: Sannolikheten att välja ett sämre tillstånd som en funktion av tiden. 25

Figur 5.3: Tabu search exempel på SAT-problemet med 5 variabler och med en tabulängd på tre iterationer. 27

Figur 6.1: En jämförelse av hamming distance mellan binary coding och gray coding. ..31

Figur 6.2: Exempel på one point cross-over, där den slumpgenererade positionen är tre.37

Figur 6.3: Exempel på two point cross-over, där de slumpgenererade positionerna är två respektive fem...38

Figur 6.4: Exempel på uniform cross-over. 38
Figur 6.5: Random mutation med tre mutationer. 39
Figur 6.6: Inorder mutation med intervall från 3 till 5 och 2 mutationer. 39
Figur 6.7: Representation av 8-queen problem som vi kommer att använda nedan i vårt exempel. 40
Figur 6.8: Ett exempel på ett generationsskifte i en Genetisk Algoritm. 42
Figur 7.1: Tidsmätning av Simulated Annealing. 51
Figur 7.2: Sannolikhet för hängningar med Simulated Annealing. 51
Figur 7.3: Tidsmätning av Genetisk Algoritm. 52
Figur 7.4: Sannolikhet för omstart med genetisk algoritm. 52
Figur 7.5: Ett snitt av tiden vid 0.6% mutationschans för genetisk algoritm. 53
Figur 7.6: Ett snitt av sannolikheten för omstart vid 0.6% mutationschans för genetisk algoritm. 53

Tabellförteckning

Tabell 3.1: Optimala lösningen är den vägen med lägst totala kostnad. 5
Tabell 3.2: Fågelvägsavstånd till Dashbalbar, används av h(n)..13
Tabell 4.1: Illustration av hur snabbt antalet möjliga vägar ökar i förhållande till antalet städer då alla städer står i förbindelse med varandra. 17
Tabell 4.2: Sanningstabell för F($\bar{x}$). Som synes finns två olika lösningar som satisfierar F($\bar{x}$), en när x_1=x_2=x_3=F och en när x_1=x_3=F och x_2=S. 20
Tabell 6.1: Binary och gray coding. 31
Tabell 7.1: Parameterinställningar för experimentet...49

1 Inledning

Forskningen kring artificiell intelligens (AI) startade efter andra världskriget och räknas idag till ett av våra nyare forskningsområden. Idag används tekniker tillhörande AI överallt, i allt ifrån posthanteringssystem, operativsystem såsom Windows till rymdfarkoster. En fundamental del inom AI har allt sedan dess födelse varit problemlösning av olika slag. En anledning till det stora intresset för problemlösning är att flertalet problem tenderar att kräva en enorm beräkningskapacitet för att lösas. En problemlösares uppgift är att finna en sekvens av händelser som leder fram till ett önskat mål, och varje sekvens av händelser som leder fram till önskat mål är en lösning på problemet. Ett problem har således ofta flera lösningar, vilka kan variera i kvalité. Ofta är det inte möjligt att hitta en optimal lösning, dvs den bästa lösningen utifrån någon form av bedömning, på ett problem inom en rimlig tidsram med de problemlösningsmetoder som finns. Således kan det ibland vara en nödvändighet att nöja sig med en nära optimal lösning vilket kan vara nog så krävande beräkningsmässigt. Ett intressant område inom problemlösning som bygger på detta är optimering, där mycket resurser lagts under senare halvan av 1900-talet.

Varje problem kan anses ha ett antal tillstånd, där ett tillstånd beskriver statusen av lösningen som tillståndet representerar. Exempelvis är en uppställning av schackpjäser i ett parti schack ett tillstånd. Vidare kan en lösning beskrivas som en sekvens av tillstånd för att nå det slutgiltiga tillståndet, målet eller i schack tillståndet schack matt. De algoritmer vi använder för att lösa ett givet problem opererar på mängden av problemets alla tillstånd, den så kallade tillståndsrymden för problemet. Då de olika tillstånden är av varierande kvalitet med avseende på en viss optimal lösning, så försöker algoritmerna optimera med avseende på tillståndens kvalitet. När vi hädanefter i uppsatsen nämner optimering är det med syftning på resonemanget ovan och inte med avseende på optimeringsproblem som man normalt syftar på inom matematiken.

Flertalet algoritmer har utvecklats med syftet att finna nära optimala lösningar samtidigt som beräkningstiderna försöks minimeras. Två stora grupper av dessa algoritmer, så kallade optimeringsalgoritmer, är idag de som bygger på heuristik respektive evolution. Vi skall i denna uppsats behandla dessa båda grupper och utföra ett experiment där vi ställer dessa mot varandra för att utröna eventuella skillnader i effektivitet.

Vi har valt att inleda vår uppsats med en snabb historisk översikt över utvecklingen inom AI och problemlösning i kapitel 2, varefter kapitel 3 är en genomgång av problemlösning med hjälp av sökning. Detta kapitel ger en inblick i svårigheterna med problemlösning och hur man med hjälp av heuristik kan effektivisera lösningen av svåra problem. En god förståelse för principerna med heuristisk sökning är nödvändigt då vi senare i uppsatsen tar upp optimering med heuristik.

Innan vi går in på teknikerna för optimering, tar vi i kapitel 4 upp några klassiska problem vilka kan lösas med optimeringstekniker. Dessa problem kommer senare i uppsatsen användas i diverse exempel och även experimentet bygger på ett av dessa. Vidare är en inblick i hur svårlösta många problem kan vara nödvändigt för att förstå behovet av någon form av optimering.

Kapitel 5 tar upp den första typen av optimering, optimering med heuristik, och kapitel 6 den andra, optimering med hjälp av genetiska algoritmer. Slutligen kommer vi i kapitel 7 jämföra algoritmer ur dessa två grupper genom implementation och applicering på ett känt problem, Traveling Salesman Problem.

2 En kort historisk introduktion

Artificiell intelligens (AI) är en term som många av oss någon gång har kommit i kontakt med men vars innebörd få egentligen känner till. Många tänker direkt på robotar eller andra system som kan tänka och agera självständigt men faktum är att AI är ett brett område med många olika grenar varav de ovannämnda endast är en delmängd. Det första steget mot AI togs redan år 384-322 f.K. av Aristotle då han formulerade ett sätt att resonera vilket han kallade syllogism [1]. Själva termen AI myntades dock inte förrän 1956 av John McCarthy och fram till dess användes flertalet andra namn[1]. Det tidigaste arbetet som idag kan härledas till AI gjordes av Warren McCulloch och Walter Pitts år 1943 och behandlade artificiella neuroner [2]. Flertalet av de tidiga programmen som använde sig av AI var däremot av problemlösningskaraktär vilken grundar sig på sökning. Användandet av heuristik för effektivisering av sökning introducerades år 1958 av Simon och Newell vilket har haft stort inflytande på problemlösning, men det skulle dröja några år innan termen heuristisk sökning kom till. Grunden för genetiska algoritmer för optimeringsproblem lades av Box och Friedman på slutet av 1950-talet [2]. De första genetiska algoritmerna uppfanns på 1960-talet av flertalet oberoende forskargrupper men implementerades först på 1970-talet av John Holland och hans studenter [3], [4].

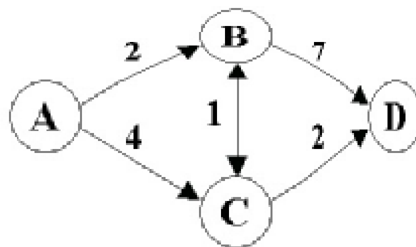
Vi ämnar ta upp mer detaljerad historik under respektive område.

[1] Complex information processing, machine intelligence, heuristic programming och cognology är några av dessa.

3 Problemlösning med sökning

Människan har i alla tider strävat efter att lösa de problem som hon ställs inför. Då forntidens problem kanske ofta handlade om att skapa verktyg för att underlätta livet, handlar det idag ofta om att lösa olika typer av beräkningsproblem eller ställa olika typer av diagnoser. För en läkare innebär problemlösning att systematiskt utvärdera alla möjliga orsaker och utifrån detta ställa en diagnos. För en datormotståndare i ett datorspel handlar det om att i förväg utvärdera olika typer av agerande samt vad dessa skulle leda till, och utifrån detta göra ett val. Det kan kanske vid första anblicken verka väldigt trivialt; upprätta ett sökträd över alla möjliga fall och traversera tills det att en önskad lösning är funnen. Men då problemet ökar linjärt i storlek kommer ofta tiden för att lösa problemet öka exponentiellt, en tid som inte alltid är tillgänglig. Som ett exempel på stora problem har det gjorts en uppskattning av antalet olika tillstånd för brädet i schack till 10^{120} vilket är större än antalet nanosekunder som passerat sedan big bang [5]. En grundläggande del inom artificiell intelligens har alltid varit problemlösning, där det ofta handlar om att förenkla och i många fall möjliggöra lösningen av komplexa problem.

En problemlösares uppgift är att finna en sekvens av händelser som leder fram till önskat mål [2]. För en läkare kan målet vara att ställa en korrekt diagnos, medan för en datormotståndare i ett spel är det att vinna spelet eller partiet. En sökalgoritm tar emot ett problem som input och returnerar en lösning i form av en sekvens av händelser [2]. Denna sekvens av händelser kan betraktas som en väg från start till mål. En optimal lösning är den lösning på ett problem som har lägst, eller högst om det vore ett maximeringsproblem, totala kostnad, där kostnad är en generell beteckning på arbetsbördan att ta sig från en nod till en annan. I många fall kan det nämligen finnas flera alternativa vägar fram till samma mål.



Figur 3.1: Hitta en väg från A till D.

Lösning	Total kostnad
A-B-C-D	5
A-B-D	9
A-C-B-D	12
A-C-D	6

Tabell 3.1: Optimala lösningen är den vägen med lägst totala kostnad.

I detta kapitel kommer vi att gå igenom några vanliga problemlösningsalgoritmer. Alla dessa används för att traversera sökträd för att hitta en lösning på det aktuella problemet. Vi har valt att börja med blinda sökalgoritmer eftersom kunskap om dessa är nödvändig för att få en helhetsbild över området, och då övriga problemlösningsalgoritmer bygger vidare på dessa. Förhoppningen är att läsaren i och med detta kapitel tydligt ska se fördelarna med heuristiska sökmetoder gentemot blinda sökmetoder.

3.1 Blinda sökningar

Inom artificiell intelligens kallas en sökning i ett träd eller en graf där algoritmen inte har någon information förutom själva problemdefinitionen för en blind sökning, alternativt oinformerad sökning [2]. Blinda sökalgoritmer för problemlösning är en central del av klassik datavetenskap. Dessa algoritmer är relativt okomplicerade jämfört med heuristiska (se 3.2) sökmetoder, då det enda de kan göra är att avgöra om den nuvarande noden är en sökt målnod eller inte. Då dessa algoritmer inte använder sig av heuristik vid vägval utan följer ett bestämt mönster varje gång, leder det till att de ofta är väldigt ineffektiva och har hög tidsåtgång för mer komplexa problem. Många större problem är rent av olösbara med blinda sökningar.

I följande delkapitel kommer vi att ta upp tre vanliga blinda sökmetoder – depth-first, breadth-first och interative deepening – och illustrera dessa med hjälp av mindre exempel.

3.1.1 Depth-first

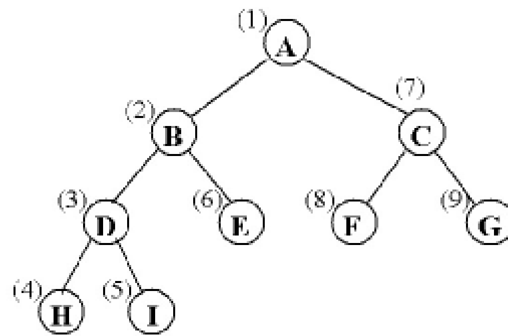
Depth-first, även kallad djupet först, är en så kallad blind sökmetod vilken alltid strävar efter att traversera ett träd på djupet, det vill säga den undersöker en nods barn[2] innan nodens

[2] Om en nod *n* ligger på en viss nivå *p,* är ett barn *b* till *n* en nod som ligger på nivå *p+1* och har en direkt länk till *n*. Noden *n* är förälder till *b*.

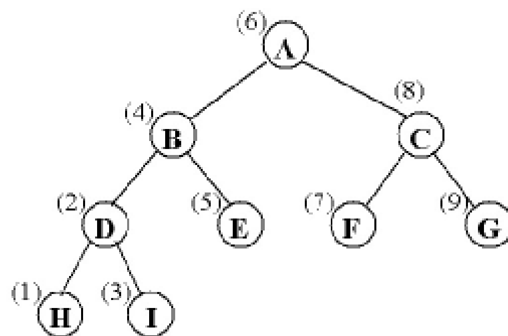
syskon[3] [5]. Algoritmer av typen depth-first delas vanligtvis in i tre olika kategorier efter dess traverseringsordning; pre-order, in-order och post-order. Traverseringsordningen avgör i vilken ordning noden och dess barn kommer att evalueras. Figur 3.2 illustrerar ett binärt träd vilket traverseras med depth-first i pre-order, där noden själv evalueras före dess barn. I motsats till detta evalueras nodens barn innan noden själv i post-order, vilket illustreras i Figur 3.4. In-order följer samma mönster vid ett binärt träd, där noden själv evalueras mellan dess barn, men vid träd innehållande noder med fler än två barn är inte ordningen entydigt definierad.

En klar nackdel med depth-first search är att evalueringsordningen kan bidra till ineffektivitet vid felaktiga vägval. Antag exempelvis att om målnoden i Figur 3.3 är C så tvingas ändå hela vänstra subträdet att evalueras.

Ur implementationssynvinkel kan depth-first enkelt implementeras med rekursion eller iterativt med hjälp av en stack.

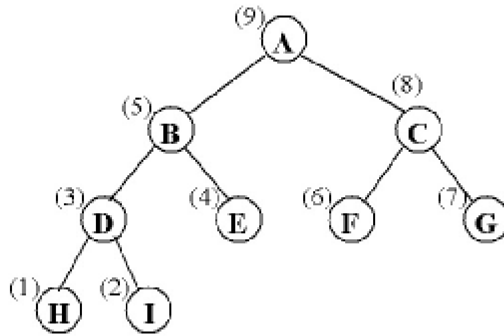


Figur 3.2: Pre-order depth-first search traversering.



Figur 3.3: In-order depth-first search traversering.

[3] Alla noder som har samma förälder är syskon.



Figur 3.4: Post-order depth-first search traversering.

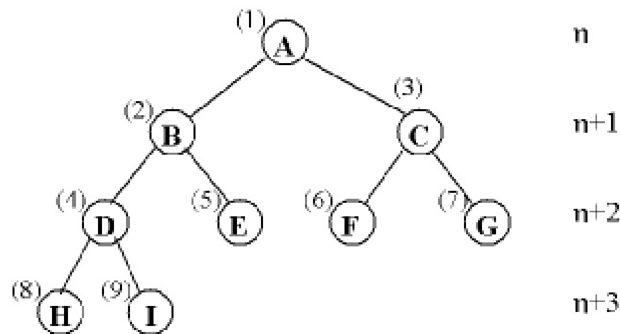
3.1.2 Breadth-first

Breadth-first, även kallad bredden först, evaluerar alla noder på nivå η innan noder på nästa nivå, $\eta+1$, evalueras. I de fall kanterna inte är förknippade med olika vikter och då en lösning existerar resulterar breadth-first search alltid i den optimala vägen. Figur 3.5 illustrerar ordningen i vilken breadth-first evaluerar ett träds noder och vi ser tydligt hur nivå efter nivå systematisk traverseras. Detta implicerar att antalet noder N som i värsta fall behöver genereras är, borträknat rotnoden,

$$N = a + a^2 + ... + a^b + (a^{b+1} - a) = \sum_{i=0}^{b} a^{i+1} - a$$

där α är det maximala antalet barn per nod och β är den nivå där lösningen återfinns eller totala antalet nivåer då lösning saknas [2].

Vid implementation av breadth-first search används en kö för temporär lagring av noder.



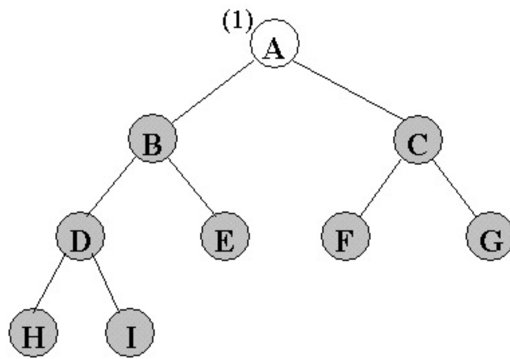
Figur 3.5: Breadth-first traversering.

3.1.3 Iterative deepening

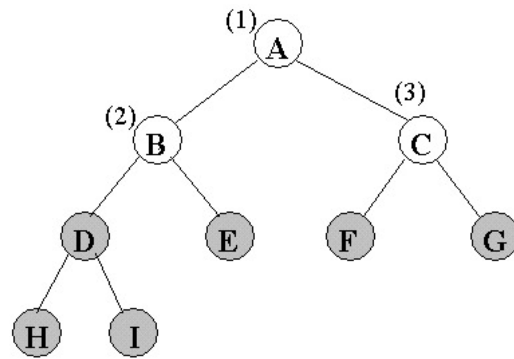
Iterative deepening, även kallad iterative deepening depth-first, är överlag den sökmetod att föredra bland de blinda sökmetoderna när sökrymden är stor och djupet okänt [2]. Iterative deepening kan likställas med en upprepad variant av depth-first search. Skillnaden mellan iterative deepening och vanlig depth-first är att i iterative deepening upprepas en något modifierad version av depth-first algoritmen η gånger, där η är antalet nivåer i sökträdet, eller tills en lösning hittas. Modifikationen av depth-first search är att den avbryter sökningen efter att ha nått ett angivet djup. Under första iterationen av depth-first genomsöks endast första nivån av sökträdet och därefter utvidgas djupet med en nivå för varje iteration. Algoritmen kan vid första anblick förefalla ineffektiv men då den drar fördel av både depth-first search låga minnesanvändning samt breadth-first search förmåga att finna den optimala lösningen visar den sig trots allt vara effektiv [2]. Det totala antalet noder N som behöver genereras är, borträknat rotnoden,

$$N = (b)a + (b-1)a^2 + \ldots + (1)a^b = \sum_{i=0}^{b-1} (b-i)a^{(i+1)}$$

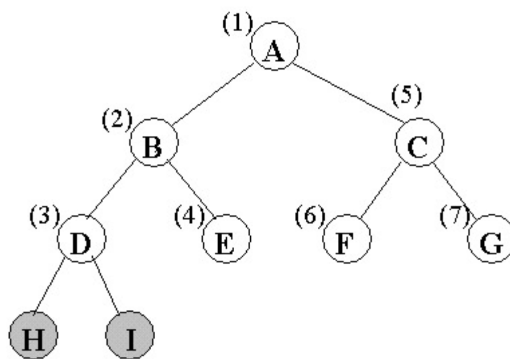
där α är det maximala antalet barn per nod och β är den nivå där lösningen återfinns eller totala antalet nivåer då lösning saknas [2]. Figur 3.6 illustrerar iterative deepening search på ett träd med fyra nivåer, gråmarkerade noder berörs ej av den aktuella sökningen.



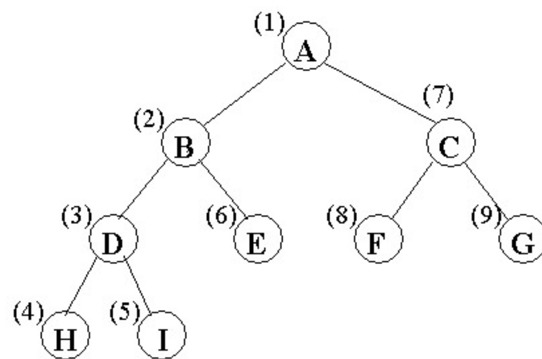
Första iterationen av depth-first search.
Endast den första nivån genomsöks.



Andra iterationen av depth-first search.
Endast första och andra nivån genomsöks.



Tredje iterationen av depth-first search.
Alla nivåer förutom sista genomsöks.



Fjärde iterationen av depth-first search.
Alla nivåer genomsöks, sökningen fulländad.

Figur 3.6: Iterative deepening search med pre-order evaluering.

3.2 Sökning med heuristik

Där vi i kapitel 3.1 behandlade blinda sökmetoder, vilka enbart bygger på själva problemdefinitionen, kommer vi i detta kapitel istället rikta in oss mot sökmetoder vilka även använder sig av problemspecifik kunskap för att ytterligare försöka effektivisera problemlösningen. Själva ordet heuristik härstammar från det grekiska ordet eurisco, vilket i sin tur betyder ”jag upptäcker”. Vid sökning med heuristik väljs inte traverseringsordningen systematiskt på samma sätt som med de blinda sökningarna utan en slags ”bäst först” funktion, evalueringsfunktion, tillämpas vid traverseringen [3]. Med hjälp av evalueringsfunktionen är det möjligt att välja den väg som förefaller leda till en lösning på problemet, istället för att traversera alla vägar en efter en. Människan fungerar på ett liknande sätt, vi väljer inte alltid det första valet utan utvärderar de val som finns och utifrån det väljer

vi det som förefaller bäst. Observera att något som förefaller vara bäst inte alls behöver vara den optimala lösningen, den behöver faktiskt inte alls leda till en lösning. På liknande sätt fungerar sökning med heuristik, den garanterar inte någon lösning och effektiviteten beror till stor del på kvalitén hos evalueringsfunktionen.

Vid implementation används ofta en form av priotiteringskö där de mest lämpade valen får högre prioritet än de mindre lämpade valen [2].

Heuristisk sökning kan med fördel tillämpas då sökrymden för ett problem helt enkelt är för stor för att kunna genomsökas i sin helhet inom en rimlig tidsram. Schack är ett typiskt exempel på detta där sökrymden blir oöverskådlig. Heuristik används även i de fall där problemet saknar en entydig lösning, det vill säga i de fall där exakt lösning saknas eller inte är tydlig, för att försöka ge en så bra lösning som möjligt [5].

3.2.1 Heuristisk funktion

En heuristisk funktion, *h(n),* uppskattar avståndet/kostnaden mellan en nod *n* och lösningen av det givna problemet (målnoden). För att kunna göra denna uppskattning krävs det att den heuristiska funktionen har extra information om det specifika problemet. Detta innebär att den heuristiska funktionen blir problemspecifik, och inte kan appliceras på andra problem. Hur väl genomtänkt en heuristisk funktion är avspeglar sig direkt på den aktuella sökmetodens kvalité. I många fall kan den heuristiska funktionen hämtas nästan direkt utifrån lösningen, som i fallet med TSP (The Traveling Salesman Problem, se kapitel 4.1) där totala kostnaden för att besöka ett antal städer ska minimeras. I vissa speciella fall är dock detta inte möjligt, och det blir då svårare att hitta en lämplig heuristisk funktion som passar problemet.

För att öka förståelsen ytterligare skall vi nedan ge ett exempel på två olika sätt att implementera en heuristisk funktion för 8-puzzle problemet. 8-puzzle går ut på att utifrån en slumpad start position, Figur 3.7, flytta ett nummer i taget tills det att målpositionen, Figur 3.8, uppnåtts. En förflyttning kan endast ske till den tomma rutan från en intilliggande ruta. Vidare får en förflyttning endast ske i vertikalt respektive horisontellt led.

En enkel typ av heuristisk funktion är att summera antalet rutor vilka inte befinner sig på sin målposition. En sådan heuristisk funktion skulle i Figur 3.7 returnera värdet sju. En annan teknik är att summera antalet steg mellan varje ruta och dess målruta, vilket i Figur 3.7 skulle returnera ett värde av åtta.

2	3	5
1		8
6	4	7

Figur 3.7: 8-puzzle startposition.

1	2	3
4		5
6	7	8

Figur 3.8: 8-puzzle målposition.

3.2.2 Evalueringsfunktion

Evalueringsfunktionen, *f(n)*, undersöker hur lämpliga en eller flera specifika vägar är för att nå en lösning. Genom att jämföra flertalet vägar med hjälp av evalueringsfunktionen kan man på så sätt avgöra vilken väg som förefaller mest lämplig. Till sin hjälp använder sig evalueringsfunktionen av den problemspecifika heuristiska funktionen och eventuellt ytterligare information för att maximera tillförlitligheten. Det finns huvudsakligen två varianter av evalueringsfunktioner, ordnade respektive numeriska [3]. De ordnade tar emot flertalet vägar och returnerar dessa ordnade efter lämplighet. De numeriska tar emot en väg och returnerar ett värde på dess lämplighet.

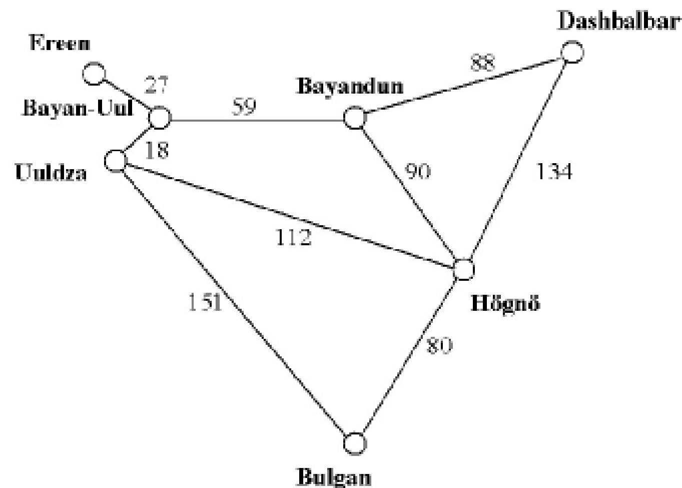
3.2.3 Greedy best-first

Greedy best-first, i vanligt tal även refererad till som bäst-först sökning, är den enklaste typen av heuristisk sökning. Evalueringsfunktionen för greedy best-first använder sig inte av någon ytterligare information förutom den heuristiska funktionen.

$$f(n) = h(n)$$

Eftersom inte någon ytterligare information används kommer algoritmen alltid att välja den väg vilken för stunden förefaller mest lämplig. Detta leder oundvikligen till ett beteende som kan liknas med depth-first. Likt depth-first försöker Greedy best-first följa en enskild väg från start till mål och överväger andra vägar först då en återvändsgränd stöts på vilket i sin tur implicerar att greedy best-first search inte är optimal. Skillnaden gentemot depth-first är dess traverseringsordning. Då traversering i depth-first sker systematiskt efter samma mönster varje gång väljer i stället greedy best-first alltid den väg som förefaller bäst. I värsta fall leder detta till en komplexitet av $O(b^m)$ där b är antalet barn per nod (b £ α) och m är det maximala djupet av sökträdet [2]. Komplexiteten hos greedy best-first kan dock sänkas avsevärt med hjälp av en bra heuristisk funktion.

Figur 3.9 visar en vägkarta som ligger till grund för de kommande exemplen på både greedy best-first och senare A* search (se 3.2.4). Målet med exemplen är att finna en väg från Uuldza till Dashbalbar. Tabell 3.2 anger fågelvägsavstånd mellan angiven stad och vår målstad, Dashbalbar, och motsvarar vår heuristiska funktions uppskattade värden.



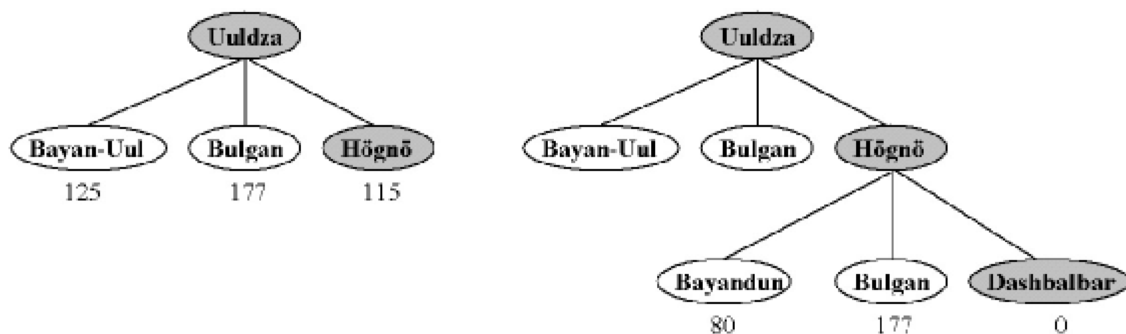
Figur 3.9: Förenklad vägkarta över en del av nordöstra Mongoliet, sträckorna är uppskattade och angivna i km.

Stad	Avstånd
Bayandun	80
Bayan-Uul	125
Bulgan	177
Dashbalbar	0
Ereen	142
Högnö	115
Uuldza	140

Tabell 3.2: Fågelvägsavstånd till Dashbalbar, används av h(n).

Figur 3.10 illustrerar ett exempel på greedy best-first search där siffrorna under trädets noder motsvarar de uppskattade värdena givna av den heuristiska funktionen. Dessa värden är helt enkelt fågelvägsavståndet från nuvarande stad (eller nod) till målet, Dashbalbar, vilket innebär att algoritmen alltid kommer att välja den stad som är närmast målet. Det här arbetssättet med att alltid välja den väg som förefaller mest lämpad har lett till att algoritmen klassas som girig, därav namnet.

Från Figur 3.9 kan vi också se att den rekommenderade vägen av greedy best-first search i det här fallet är en giltig lösning på problemet, men inte den optimala.



Figur 3.10: Greedy best-first applicerat på problemet att hitta en väg från Uuldza till Dashbalbar (se karta i Figur 3.9)

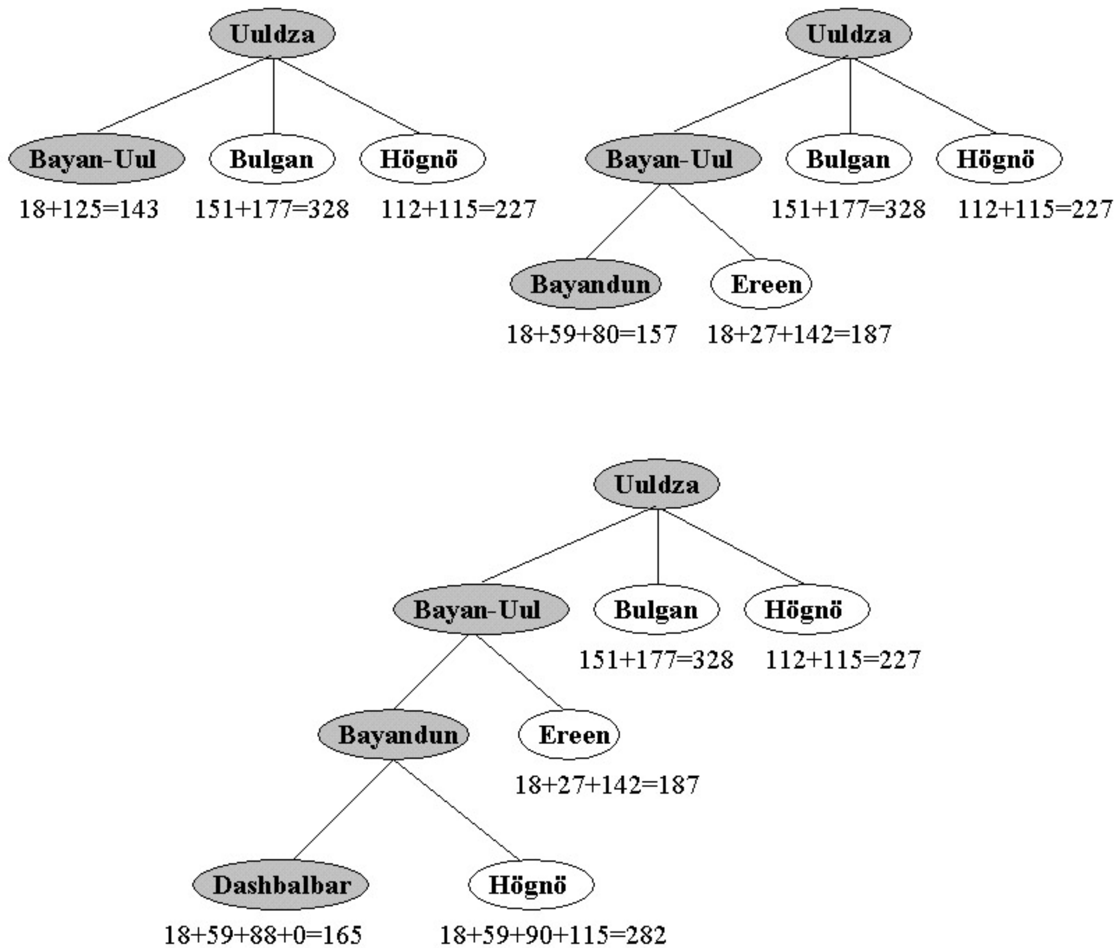
3.2.4 A*

Att alltid välja den för stunden bästa vägen kan kanske förefalla logiskt, men som vi sett tidigare är detta inte alltid det mest optimala arbetssättet. På grund av detta gjordes det år 1968 en vidareutveckling av greedy best-first som fick namnet A*. Denna vidareutveckling gjordes av Hart, Nilsson och Raphael för att istället ta hänsyn till den totala kostnaden för en lösning. För att möjliggöra detta använder sig evalueringsfunktionen av ytterligare en funktion, $g(n)$, vilken returnerar den exakta kostnaden från startnod till den nuvarande noden, n. Genom att kombinera $g(n)$ med den heuristiska funktionen, $h(n)$, är det möjligt att få ett uppskattat värde på kostnaden för hela lösningen.

$$f(n) = g(n) + h(n)$$

Det går att visa att A* är optimal så länge den heuristiska funktionen inte överskattar kostnaden av vägen från den nuvarande noden till målnoden. Ett annat krav för detta är att ingen kostnad för någon kant i sökträdet är negativ [1]. Vidare är A* optimalt effektiv för en godtycklig heuristisk funktion. Detta innebär att det inte existerar någon annan algoritm som är optimal samt kan garantera generering av färre noder än A* vid användandet av samma heuristiska funktion [2]. Precis som med ovanstående algoritmer, kommer A* alltid att finna en lösning såvida tids- och minnesåtgång inte är begränsande faktorer.

I Figur 3.11 illustreras samma problem som i 3.2.3, fast med hjälp av A*. Då A*s evalueringsfunktion ser ut som den gör, innebär siffrorna under en nod helt enkelt summan av det exakta färdavståndet från start till nuvarande nod, $g(n)$, och fågelavståndet från nuvarande nod till målnoden, $h(n)$. Den uppmärksamme läsaren kanske redan har noterat att vägen som förespråkas av A* även är den optimala lösningen på problemet. Det är möjligt för A* att hitta den optimala lösningen, eftersom den heuristiska funktionen returnerar fågelvägsavståndet och detta avstånd aldrig kan vara en överskattning av en sträcka.



Figur 3.11: A applicerat på problemet att hitta en väg från Uuldza till Dashbalbar (se karta i Figur 3.9)*

Eftersom A* lagrar alla besökta noder och den bästa vägen till varje nod, kommer minnesåtgången bli betydlig. Detta implicerar att A* är i det närmaste obrukbar för stora problem. På grund av detta har det utvecklats flertalet förbättringar av A*, som syftar till att minska tids- och minnesåtgången. Vi tänker inte gå in på dessa, men den intresserade läsaren kan hitta information om dessa i flertalet böcker, bland annat i *Artificial Intelligence: A Modern Approach* av Stuart Russell och Peter Norvig.

4 Klassiska problem

I kapitel 3 tog vi upp problemlösning där lösningen utgjordes av en väg. Vägen är en uppsättning sekventiella steg som måste tas för att komma från utgångsläget till målet. I det här kapitlet skall vi i stället ta upp ett par problem vilka kan betraktas som optimeringsproblem, där själva stegen fram till målet saknar betydelse utan där enbart målet är lösningen. Eftersom vägen fram till lösningen inte är av intresse får detta till följd att de algoritmer som används för att lösa denna typ av problem inte behöver lagra information om tidigare utförda steg. Detta gör i sin tur dessa algoritmer minnessnåla [2]. Nedan går vi igenom ett par problem med fokus på The Traveling Salesman Problem eftersom det kommer att ligga till grund för vårt experiment senare i uppsatsen.

4.1 The Traveling Salesman Problem

Problemet med den resande försäljaren, mest känt som The Traveling Salesman Problem eller förkortat TSP, har länge intresserat forskare världen över. Namnet ”Traveling Salesman Problem” tros ha uppkommit i USA år 1955 i en artikel av Morton och Land. I artikeln beskrevs problemet som en försäljare vilken skulle besöka de 48 delstaterna samt Washington D.C en gång vardera i en sådan ordning att den totala färdsträckan minimerades. Sedan dess har flertalet varianter, exempelvis Clustered TSP, uppkommit för att spegla verkliga problem som dykt upp [3]. Problemet kan liknas med att söka efter den kortaste Hamilton cykeln[4] i en graf vilket matematiker intresserade sig för redan på 1800-talet [6]. Det dröjde ända fram tills år 1972 innan någon bevisade att TSP var NP-hard [2], det vill säga dess komplexitet kan inte bevisas kräva mindre än exponentiell tid för att lösas utan hjälp av heuristik [5].

Generellt kan sägas att TSP går ut på att en försäljare ska besöka n städer en gång var med kortast möjliga färdsträcka. Given mängden av alla Hamilton cykler i en graf, söker vi alltså ur en matematisk synvinkel den Hamilton cykel vilken har lägst totala kostnad där kostnaden är summan av vikterna hos samtliga kanter i cykeln. Problemet går alltså ut på att hitta den väg som passerar alla noder i en graf en gång, och med så liten kostnad som möjligt. Det är

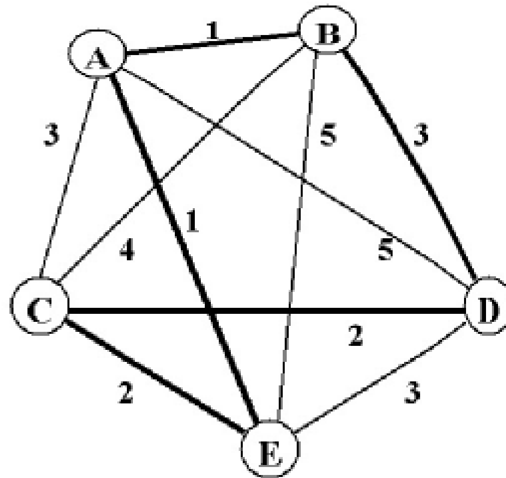
[4] En oriktad graf anses innehålla en Hamilton cykel då det existerar en cykel sådan att den förbinder samtliga grafens noder (vertices). En cykel är en väg (path) där startnoden är samma som slutnoden. Vidare säger definitionen av en väg att ingen nod får upprepas. [6]

lätt att inse att om det finns en direktväg från varje nod till alla andra noder, kommer det att finnas *n!* antal möjliga vägar. Om man bortser från i vilken nod färden startar, kan antalet möjliga vägar reduceras till *(n-1)!*, då det finns *n* versioner av varje väg där endast start- och målnoden är olika. I fallet med symmetrisk TSP, där kostnaden att ta sig från en stad till en annan är oberoende av färdriktningen, kan antalet möjliga vägar halveras, vilket ger oss *(n-1)!/2* möjliga vägar. Då *n* växer linjärt kommer antalet möjliga vägar öka dramatiskt, så redan vid relativt små värden på *n* kommer antalet vägar vara oerhört stort. Detta resulterar i att det är ohållbart att undersöka alla möjliga kombinationer av noder och därför får vi i många fall även nöja oss med endast en nära optimal lösning.

n	***(n-1)!/2***
3	1
5	12
7	360
10	181440
12	19 958 400
15	43 589 145 600
20	60 822 550 204 416 000
25	310 224 200 866 620 000 000 000
30	4 420 880 996 869 850 000 000 000 000 000

Tabell 4.1: Illustration av hur snabbt antalet möjliga vägar ökar i förhållande till antalet städer då alla städer står i förbindelse med varandra.

I Figur 4.1 illustreras ett enkelt exempel på TSP med fem noder. Den markerade vägen uppfyller kravet för en Hamilton cykel, då den besöker samtliga noder endast en gång vardera och inte utnyttjar någon kant mer än en gång. Vidare kan vi enkelt resonera oss fram till att den markerade vägen är en optimal lösning då summan av de fem kortaste kanternas vikter ger värdet nio. Detta implicerar i sin tur att det inte kan existera någon väg med lägre kostnad än nio. Då lösningen i Figur 4.1 har kostnaden av just nio kan vi alltså dra slutsatsen att vi har funnit en optimal väg.



Figur 4.1: TSP med fem städer. Den kortaste (optimala) vägen är markerad med feta linjer.

4.2 8-queen problem

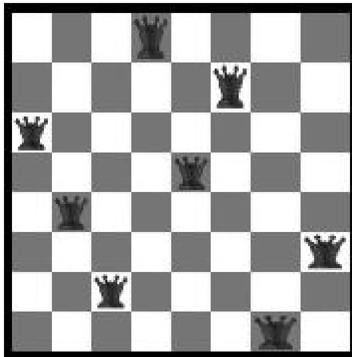
Första kända förekomsten av 8-drottningsproblemet är en publicering i en tysk schacktidning år 1848 och det dröjde två år innan alla 92 möjliga lösningarna på problemet var kartlagda. Problemet generaliserades i början av 1900-talet till ett *n*-drottningproblem och med hjälp av dagens algoritmer, exempelvis random-restart hill-climbing vilket tas upp i kapitel 5.1, har en lösning funnits då *n* är lika med tre miljoner på under en minut. [2]

Själva problemet går ut på att placera *n* drottningar på ett *n* * *n* stort schackbräde på ett sådant sätt att ingen drottning hotar någon annan drottning. En drottning hotar en annan pjäs placerad på samma rad, kolumn eller diagonal.

Även detta problem växer till ofantlig storlek då *n* växer. Vid åtta drottningar finns det totalt

$$\binom{8^2}{8} \approx 1{,}78 * 10^{14}$$

möjliga kombinationer. Genom att sätta ut en drottning i taget på sådant sätt att ingen drottning hotar någon annan har antalet kombinationer minskat till 2057 st vilket är klart överkomligt. Om vi däremot lät *n* gå mot 100 skulle det totala antalet möjliga kombinationer bli ungefär 10^{400} och det reducerade antalet cirka 10^{52} [2]. Även det mindre antalet av dessa kombinationer resulterar i en sökrymd så stor att vi kan bortse från de blinda sökmetoderna.



Figur 4.2: En av de 92 lösningarna till 8-drottningsproblemet.

4.3 Satisfiability problem

Satisfiability (SAT) problemet är ett fundamentalt problem inom matematisk logik. Problemet går helt enkelt ut på att finna en satisfierbar lösning till ett sammansatt uttryck. Liksom de tidigare problemen vi tagit upp i detta kapitel växer sökrymden exponentiellt med antalet variabler i uttrycket. Inom logiken kan en variabel endast ha två värden, sant eller falskt, men trots detta blir antalet kombinationer snabbt ohanterligt. Exempelvis skulle ett SAT problem med 100 variabler resultera i sökrymd innehållande $2^{100} \approx 10^{30}$ tillstånd.

Uttrycket som ska satisfieras är ett booliskt uttryck, som bara kan anta värdena sant eller falskt. Ett exempel på ett sådant uttryck med 3 variabler skulle kunna vara

$$F(\vec{x}) = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3)$$

x_1	x_2	x_3	$F(\vec{x})$
F	F	F	S
F	F	S	F
F	S	F	S
F	S	S	F
S	F	F	F
S	F	S	F
S	S	F	F
S	S	S	F

Tabell 4.2: Sanningstabell för $F(\vec{x})$. Som synes finns två olika lösningar som satisfierar $F(\vec{x})$, en när $x_1=x_2=x_3=F$ och en när $x_1=x_3=F$ och $x_2=S$.

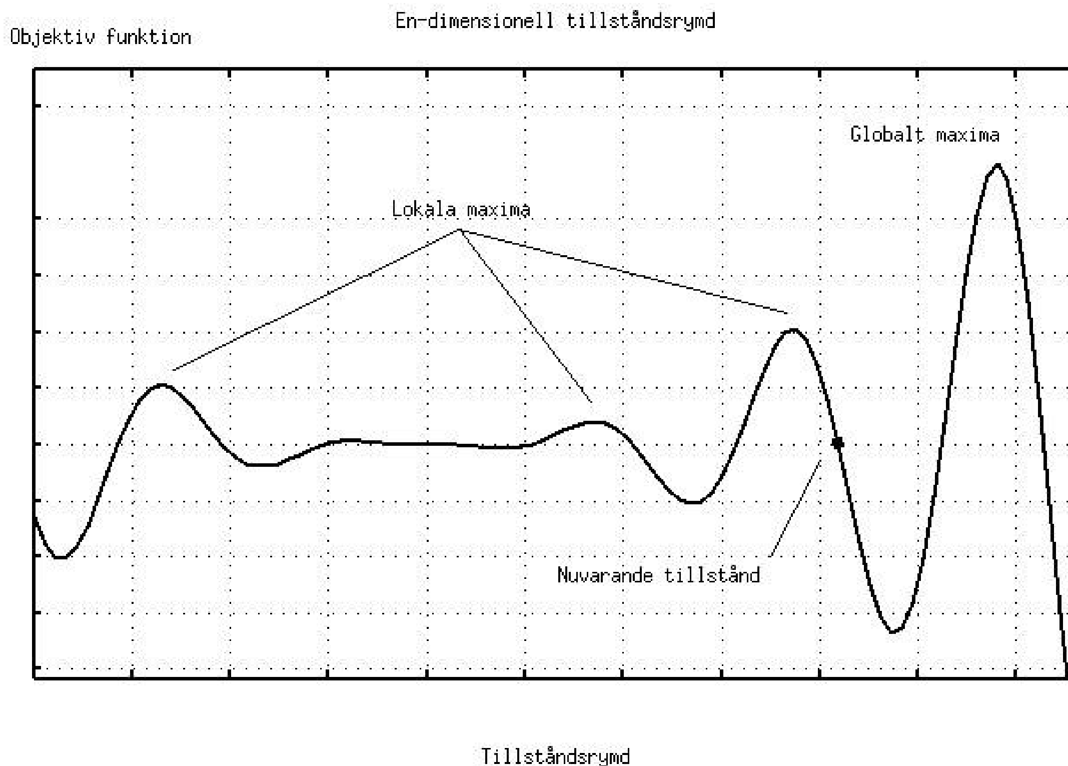
Vad som skiljer SAT från de ovannämnda problemen, är att valet av heuristisk funktion inte alls är uppenbart. Då uttrycket som ska satisfieras i sig endast kan returnera sant eller falskt, innebär det att alla tillstånd som inte satisfierar uttrycket är likvärdiga, och det går inte att enkelt skilja på ”bättre” och ”sämre” tillstånd. På grund av detta kräver skapandet av den heuristiska funktionen ett annorlunda tankesätt gentemot exempelvis TSP, vilket ligger utanför denna uppsats.

5 Optimering med heuristik

I kapitel 3.2 (Sökning med heuristik) behandlade vi algoritmer vilka systematiskt traverserade ett sökträd och utifrån varje nod tog ett beslut angående fortsatt färdriktning. Under hela färden mot lösningen lagrades alla noder på den aktuella vägen genom sökträdet och när målnoden funnits motsvarar den tillfälligt lagrade vägen vår sökta lösning. I de fall vi inte är intresserade av vägen till en sökt lösning utan enbart lösningen själv behöver inte den aktuella vägen lagras, vilket gör de algoritmer som baseras på detta väldigt minnessnåla. Algoritmer inom denna genre kallas ofta lokala sökmetoder (Local search) och de rör sig, precis som heuristiska problemlösningsalgoritmer (Kapitel 3.2), generellt sett endast till intilliggande tillstånd[5].

I Figur 5.1 illustreras tillståndsrymden, det vill säga alla möjliga tillstånd, för ett problem i horisontell led och i vertikal led evalueras tillståndens lämplighet i förhållande till målet med hjälp av en objektiv funktion. Vid lokala sökmetoder försöker vi nå det globala maximet, eller i vissa fall det globala minimet, då det representerar den optimala lösningen. Tillståndsrymden nedan är synnerligen enkel och speglar få verkliga problem då varje tillstånd här endast har två intilliggande tillstånd, men exemplet fungerar bra för att illustrera principen. Samtliga optimeringsalgoritmer i detta kapitel kan jämföras med en förflyttning längs funktionskurvan med syftet att finna det globala maximet. Dock vill vi än en gång poängtera att vårt exempel är väldigt förenklat, då verkliga tillstånd i de flesta problem har fler än två intilliggande tillstånd och det kan existera cykler bland dem. Antalet riktningar och dimensioner ökar då snabbt vilket gör att en sådan tillståndrymd med tillhörande objektiv funktion kräver illustrationer med hjälp av hyperytor.

[5] Ett tillstånd *T* är i fallet med 8-queen problemet en viss uppställning av de åtta drottningarna på schackbrädet. Ett intilliggande tillstånd till *T* har samma uppställning men där en drottning flyttats på något sätt. Noteras bör att tillstånd är problemspecifika, och det är således svårt att ge en allmän definition av ett tillstånd.



Figur 5.1: Exempel på en en-dimensionell tillståndsrymd där höjden motsvarar värden på den objektiva funktionen. Globalt maxima innebär den optimala lösningen.

I det här kapitlet kommer vi att ta upp några vanliga lokala sökalgoritmer för optimering – hill-climbing, simulated annealing och tabu search – och några av deras varianter. Vi kommer att dra paralleller till Figur 5.1 för att illustrera algoritmernas arbetssätt så lättförståeligt som möjligt.

5.1 Hill-climbing

Hill-climbing påminner mycket om greedy best-first, som vi behandlade i kapitel 3.2.3, i det avseendet att algoritmen alltid strävar efter att nå ett bättre tillstånd i jämförelse med det nuvarande. Skillnaden är att då local search algoritmer inte håller reda på vägen fram till nuvarande tillstånd saknar hill-climbing möjligheten att gå tillbaka och pröva alternativa vägar. Denna svaghet resulterar i att hill-climbing liksom flertalet applikationer för numerisk optimering inom industrin lätt fastnar i lokala maxima [3].

Namnet hill-climbing implicerar ett maximeringsproblem, men det kan likaväl tillämpas på minimeringsproblem, antingen genom att negera den objektiva funktionen eller genom att

söka det globala minimet. Detta skulle kunna kallas hill-descending, men det allmänna namnet är hill-climbing och vi kommer således att hålla oss till detta.

Om vi applicerade hill-climbing på tillståndsrymden tillhörande Figur 5.1 och det markerade nuvarande tillståndet, skulle algoritmen ha två vägar att välja mellan – den kan gå upp till vänster mot ett högre värde på den objektiva funktionen, eller till höger mot ett lägre värde. Med dessa två färdriktningar kommer hill-climbing att röra sig mot det lokala maximet då det för stunden ser mest lovande ut. Algoritmen kommer således att köra fast i det lokala maximet och terminera.

Om vi fortsätter att dra paralleller till Figur 5.1, så kan vi enkelt dra slutsatsen att det erhållna maximet endast beror på vilket starttillstånd som valts. Det är dessutom ofta väldigt svårt att veta vilket starttillstånd som bör väljas för att algoritmen inte ska fastna i ett lokalt maxima. På grund av detta finns flertalet metoder för val av starttillstånd, allt ifrån helt slumpmässiga till mer problemspecifika. Dessutom tillämpas algoritmen ofta med flera olika starttillstånd med syftet att få spridning över hela tillståndsrymden eller om möjligt ringa in det globala maximet.

Random restart hill-climbing är en itererad variant av hill-climbing, som vid varje iteration slumpar ut ett nytt starttillstånd. Till skillnad från hill-climbing är random restart hill-climbing komplett, av den enkla anledningen att alla möjliga starttillstånd så småningom kommer att ha genererats. Det kan förefalla som ett ineffektivt sätt, men för vissa optimeringsproblem såsom 8-queen har det visat sig ytterst effektivt. Även då antalet drottningar är så högt som tre miljoner finner random restart hill-climbing en lösning inom en liten tidsram. Algoritmens effektivitet beror dock till stor del på tillståndsrymdens utseende och fungerar som bäst då antalet lokala maxima är litet. [2]

Stochastic hill-climbing och first-choice hill-climbing är ytterligare två varianter av hill-climbing vilka lämpar sig i vissa specifika fall. Stochastic hill-climbing väljer slumpmässigt en av de vägar som leder till ett bättre tillstånd. I vissa fall kan sannolikheten för ett val baseras på hur mycket bättre ett val är i förhållande till de övriga. First-choice hill-climbing är en påbyggnad av stochastoc hill-climbing där de närliggande tillstånden evalueras i slumpmässig ordning tills det att ett tillstånd som är bättre än det nuvarande erhålls. Då antalet intilliggande tillstånd är stort är denna metod en bra strategi [2].

5.1.1 Local beam

Local beam search håller reda på *k* olika tillstånd, istället för bara ett. Vid varje steg av algoritmen genereras alla intilliggande tillstånd till alla *k* nuvarande tillstånd. Om ett av de genererade tillstånden är ett mål så avbryts algoritmen, annars väljs de *k* bästa tillstånden ut från mängden av alla intilliggande tillstånd och sedan upprepas allt. Detta kan liknas vid *k* parallella hill-climb algoritmer vilka samarbetar genom att koncentrera sina ansträngningar där mest framsteg görs. Tyvärr resulterar detta ofta i att algoritmen snabbt överger en stor sökrymd för att helt och hållet koncentrera sig på en mindre sökrymd. Detta gör i slutändan ofta local beam till en något effektivare version av hill-climbing fast till en kostnad av mer minnesanvändning.

5.1.2 Stochastic beam

Stochastic beam search är i första hand en vidareutveckling av local beam search som bygger på stochastic hill-climbing fast med en liten möjlighet att även välja ett sämre tillstånd. Istället som med local beam välja de *k* bästa intilliggande tillstånden, väljs nu tillstånd slumpmässigt där sannolikheten för att välja ett specifikt tillstånd är en funktion av tillståndets lämplighet. Detta arbetssätt påminner lite om det naturliga urvalet genetiska algoritmer tillämpar vilket vi kommer att ta upp mer om i kapitel 6.

5.2 Simulated annealing

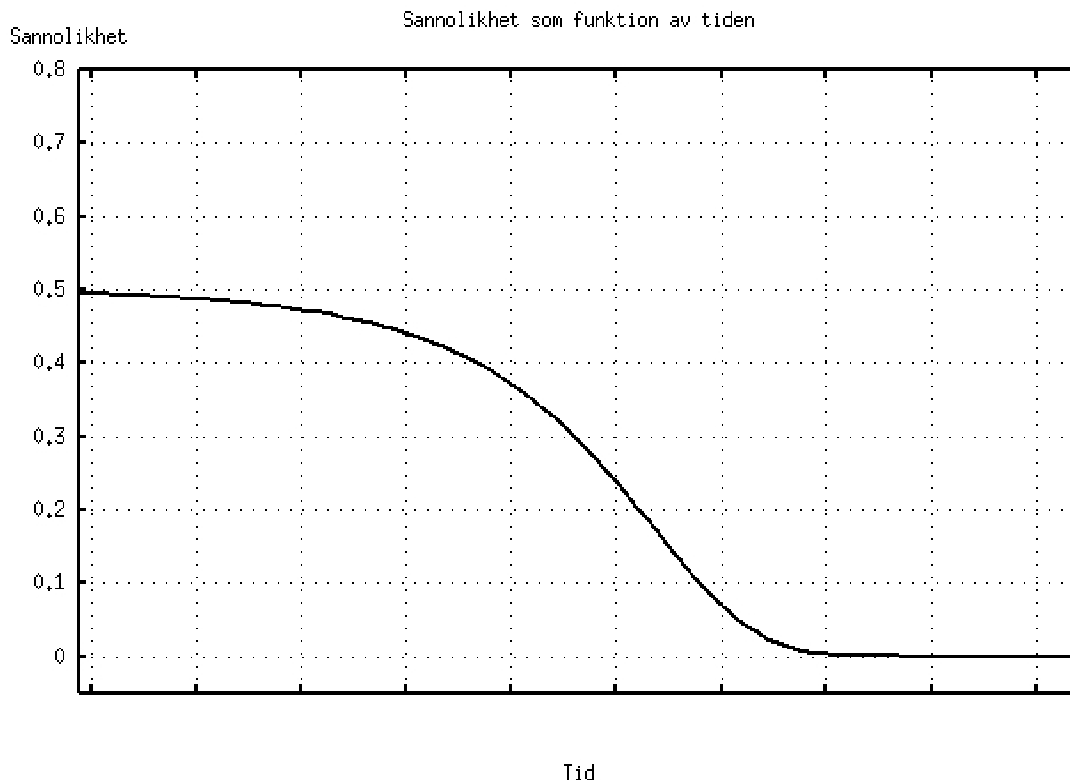
Den första beskrivningen av Simulated annealing[6], SA, gjordes av Kirkpatrick år 1983. Sedan dess har intresset ökat och idag ses det som ett eget forskningsområde med hundratals artiklar publicerade varje år [2].

SA liknar stochastic hill-climbing i det avseendet att algoritmen använder sig av sannolikheter. Algoritmen kan även liknas med en kombination av vanlig hill-climbing och s.k. random walk, där random walk innebär helt slumpvis traversering. I SA evalueras de intilliggande tillstånden ett i taget utan någon speciell ordning. Det nya med denna algoritm är att ett sämre tillstånd kan accepteras, dock med en viss sannolikhet *p* som beror på tiden som passerat och hur pass mycket sämre tillståndet är,

$$p = \frac{1}{1 + e^{\frac{eval(v_c) - eval(v_n)}{T}}}$$

[6] Monte Carlo annealing, statistical cooling, probabilistic hill-climbing, stochastic relaxation och probabilistic exchange algorithm är andra förekommande namn.

där *eval(v)* är en evalueringsfunktion för ett tillstånd v, v_c är det nuvarande tillståndet, v_n är det intilliggande tillstånd som bearbetas för stunden och T är en viss temperatur [3]. Vid algoritmens start är T som högst, och sannolikheten att välja ett sämre tillstånd än det nuvarande är då som högst. T avtar därefter med tiden för att gradvis minska denna sannolikhet till förmån för de bättre tillstånden. I Figur 5.2 illustreras ett exempel på hur sannolikheten för att göra ett sämre val skulle kunna ändras i förhållande till tiden.



Figur 5.2: Sannolikheten att välja ett sämre tillstånd som en funktion av tiden.

Idén bakom SA härstammar från metallurgin, där annealing är en process där en metall hettas upp för att sedan låtas avsvalna med en viss hastighet. Processen avser härda metallen och återfinns även inom termodynamiken vid skapandet av kristaller. Avsvalningens hastighet

är en kritisk faktor för hur mycket hårdare metallen blir i slutändan vilket även gäller i SA för den funktion som har till uppgift att sänka T.

Ur implementationssynvinkel kräver simulated annealing mer arbete gentemot hill-climbing då den även kräver en avsvalningsfunktion. Utseendet på avsvalningsfunktionen är viktigt, då det går att visa att om temperaturen sänks tillräckligt sakta, kommer SA hitta ett globalt maxima med en sannolikhet som närmar sig 1 [2]. Samtidigt får temperaturen inte sjunka för sakta, då algoritmen förlorar en del av sin effektivitet. En initialtemperatur T måste även bestämmas tillsammans med avsvalningsfunktionen för att maximera algoritmens effektivitet.

5.3 Tabu search

En av de nyare lokala sökmetoderna är tabu search som presenterades för första gången år 1989 av Fred Glover och sedan dess har intresset för algoritmen växt [2]. Idén bakom tabu search är precis som simulated annealing att försöka undfly lokala optimum i sökandet efter det globala optimumet. Tillvägagångssättet skiljer sig däremot mycket då tabu search använder sig av en så kallad tabulista där nyliga tillståndsförändringar lagras. Precis som simulated annealing väljs nya tillstånd från de närliggande, däremot finns ingen slump med i bilden i standardutförandet och tabulistan blockerar vissa tillstånd som inte får användas.

Ur en abstrakt synvinkel fungerar tabulistan som en slags lista över de tillstånd som algoritmen inte får använda för tillfället. Ur implementationsmässig synvinkel är det snarare så att tabulistan lagrar de värden i beskrivningen av tillståndet som har ändrats nyligen och inte får ändras igen på ett tag. Det är ”tabu” på dessa tillstånd eller förändringar. Vidare raderas gamla inlägg i tabulistan efter ett bestämt antal iterationer i algoritmen, eller steg i sökningen beroende på betraktarens abstraktionsnivå.

Då det är svårt att beskriva tabu search med ord har vi valt att ge ett enkelt exempel som visar på algoritmens arbetssätt. Som problem har vi i detta exempel valt SAT (Satisfiability problem, se kapitel 4.3) då det är ett enkelt problem att dra en parallell till. Värdena på evalueringsfunktionen för de olika närliggande tillstånden är fiktiva då vårt mål endast är att illustrera algoritmens arbetssätt och inte ett exempel taget från verkligheten. I Figur 5.3 visas ett exempel med ett slumpgenererat starttillstånd där en 1:a motsvarar sant och en 0:a motsvarar falsk. Varje evalueringsvärde motsvarar det närliggande tillstånd som ges vid förändring av ovanstående bit i det nuvarande tillståndet. Varje tillstånd har fem närliggande sådana där var och ett skiljer sig från nuvarande tillstånd med en bit. Det tillstånd vilket har

det högsta evalueringsvärdet väljs till nytt nuvarande tillstånd inför nästa iteration och markeras i tabulistan. Alla värden i tabulistan som skiljer sig från noll minskas med ett för varje iteration. Vi har valt tabulängden tre, vilket innebär att när en bit ändras blir den otillgänglig under de nästkommande tre iterationerna. Notera att tabulängden är avgörande för hur effektiv algoritmen är, vid en längd av fem i vårt exempel skulle algoritmen avbrytas efter fem iterationer och vid en längd av fyra skulle den fastna i en cykel. Vidare skulle en tabulängd på ett resultera i ett beteende som är snarlikt hill-climbing.

Iteration	Närliggande tillståndsrymd							Tabulista				
1	Starttillstånd	1	0	1	1	0		0	0	0	0	0
	Evaluering	4	6	3	6	**7**						
2	Tillstånd	1	0	1	1	1		0	0	0	0	3
	Evaluering	5	4	**9**	3	-						
3	Tillstånd	1	0	0	1	1		0	0	3	0	2
	Evaluering	4	**10**	-	7	-						
4	Tillstånd	1	1	0	1	1		0	3	2	0	1
	Evaluering	**11**	-	-	3	-						
5	Tillstånd	0	1	0	1	1		3	2	1	0	0
	Evaluering	-	-	-	7	**14**						
6	Lösning	0	1	0	1	0		2	1	0	0	3

Figur 5.3: Tabu search exempel på SAT-problemet med 5 variabler och med en tabulängd på tre iterationer.

Som många andra algoritmer har det gjorts en del varianter för att försöka öka dess effektivitet. Ett vanligt tillägg är evaluering av samtliga närliggande tillstånd inklusive de som är förbjudna enligt tabulistan då ett av dem skulle kunna resultera i ett exceptionellt bra val. Med ett exceptionellt bra val syftar vi på val vilka är långt bättre än de övriga och i dessa specifika fall ignorerar algoritmen tabulistans blockering. Detta implementeras med ett speciellt aspireringskriteria, som när det uppfylls leder till att ett i normala fall förbjudet val

görs. Det här arbetssättet påminner om hur vi som människor fungerar i verkliga livet, om vi ser en möjlighet är det lätt att glömma bort principerna vilka vi vanligtvis lever efter. [3]

Ett annat vanligt tillägg för att försöka få en bredare sökrymd är så kallat frekvensbaserat minne. Frekvensbaserat minne vilket kan implementeras genom ett långtidsminne som håller reda på hur många gånger en specifik bit har flippats i fallet med SAT. Denna frekvens viktas och kombineras med evalueringsfunktionens värde vid varje iteration för att på så sätt gynna de minst använda bitarna och öka spridningen. Ofta används denna metod bara under speciella omständigheter, såsom där alla möjliga val leder till ett sämre tillstånd. [3]

6 Optimering med genetiska algoritmer

Charles Darwin presenterade år 1859 sin teori om evolution utan vetskap om hur organismers egenskaper förs vidare mellan generationer. I slutet av 1800-talet presenterade James Baldwin en ytligt sett liknande teori där han menade på att en organisms anpassning till miljön kunde påverka både evolutionshastighet och -riktning. Det dröjde dock ända fram till mitten av 1900-talet innan Watson och Crick identifierade DNA-molekylens struktur. Vi vet idag att det är DNA-molekylerna som utgör våra gener och att det är en kombination av föräldrarnas DNA som avgör nästa generations DNA. På detta sätt förs våra gener vidare från generation till generation och har så gjorts sedan tidernas begynnelse. I enlighet med Darwins teori finns vidare något som heter naturligt urval, vilket helt enkelt innebär att de starkaste överlever. De organismer vilka har störst chans att överleva, möjligtvis tack vare gynnsamma egenskaper, har också störst chans att föra vidare sina gener till kommande generationer. Varje organism kan sägas ha en viss ”lämplighet” för reproduktion, och egenskaperna hos organismer med högre lämplighet kommer också att ha större genomslagskraft i nästa generation. Mutation är ett annat viktigt begrepp inom evolutionsteori, då uppkomsten av nya egenskaper hos en organism i regel sker i samband med en mutation. Mutation kan sägas vara kopieringsfel vid kopiering av DNA-strängar, exempelvis vid fortplanting eller normal celldelning [4].

Allt detta som vi har nämnt ovan kommer igen inom evolutionär beräkning. Evolutionär beräkning, eller Evolutionary Computing (EC), är efterliknandet av det naturliga urvalet, som Darwin presenterade i och med sin teori om evolution, på en sökmetod [7]. Genetiska algoritmer (GA), som vi ska ta upp i detalj i det här kapitlet, är en klass av algoritmer tillhörande EC vilka modellerar genetisk evolution.

GA skapades på 1960-talet av flertalet oberoende forskargrupper men implementerades först på 1970-talet av John Holland [3], [4]. John Hollands primära mål var att studera en simulerad organisms anpassning till förändringar i miljön. Sedan dess har GA utvecklats för att lösa olika typer av problem och själva begreppet GA skiljer sig idag starkt från John Hollands originalidé.

I detta kapitel kommer vi att ta upp de olika delarna av GA – representation, population, lämplighet, urvals- och reproduktionsoperatorer – dels genom enkla exempel men också genom att dra paralleller till evolutionsteorin, för att sedan knyta ihop allt i ett stort exempel i

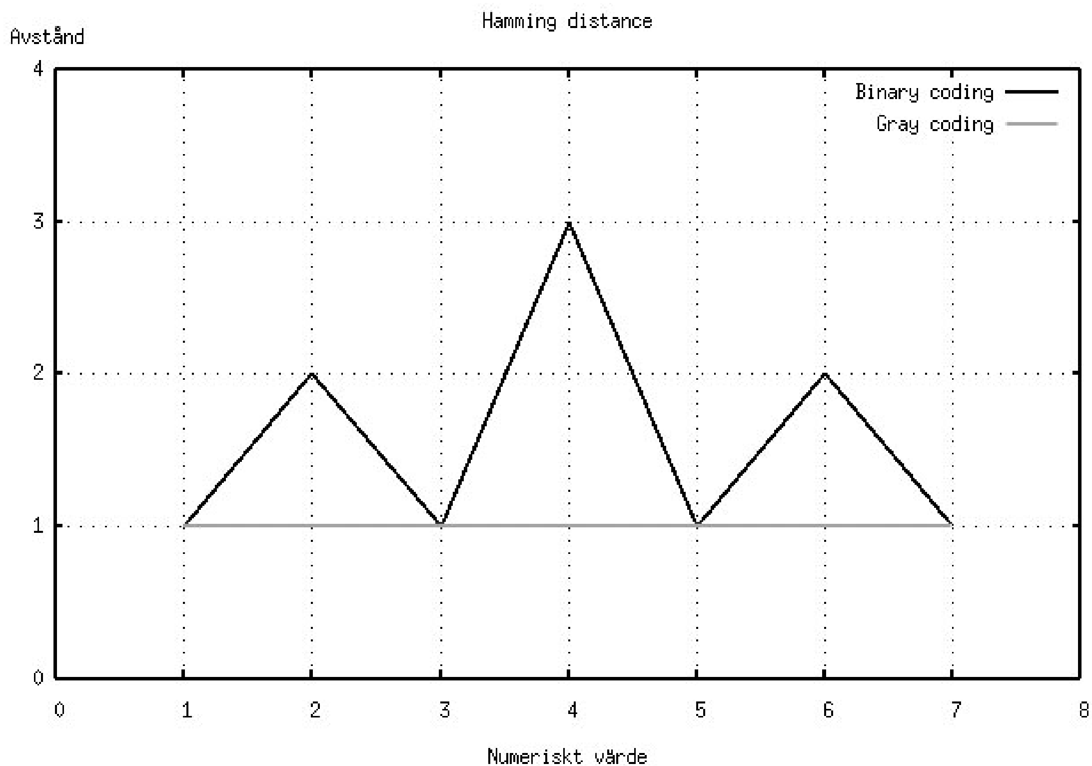
slutet av kapitlet. Detta sista stora exempel syftar till att skapa förståelse för hur alla beståndsdelar hos en genetisk algoritm samverkar.

6.1 Representation

Med representation inom GA syftar man till den datastruktur vilken används för att representera en tänkbar lösning och mappningen av tillstånd i tillståndsrymden till denna datastruktur. Antalet olika representationer är lika många som vi kan komma på och då valet av representation är en viktig faktor, kanske den viktigaste, för effektiviteten hos den genetiska algoritmen skall vi i detta kapitel presentera två vanliga representationssätt [4]. Själva representationen av en lösning kan ses som en individs kromosom, individens samlade egenskaper, som i sin tur är uppbyggd av ett antal gener där en gen representerar ett specifikt karaktärsdrag hos individen [7]. Vanligtvis representeras tänkbara lösningar genom en bitsträng av ett fixt antal bitar med en bestämd ordning, men på senare tid har det även experimenterats med andra typer av representationsscheman.

Binär kodning, eller binary coding, är kanske det första och mest naturliga representationssätt för många, och det är också det vanligaste av flera anledningar. Den ursprungliga anledningen är att John Holland använde denna typ av representation vilket fått till följd att många har gjort likadant [4]. Vidare klarar binär kodning av att utnyttja hela kapaciteten hos den bitsträng som representerar en lösning. Trots detta finns det ofta bättre representationssätt för många problem. Problemet med binär kodning är att en liten förändring hos en kromosom kan få stort utslag för dess lämplighet. [8]

Gray coding minimerar problemet med binär kodning då en liten förändring hos kromosomen avspeglar sig som en liten lämplighetsförändring. Tanken bakom gray coding är att minimera det s.k. hamming distance, det vill säga att minimera antalet bitförändringar hos en bitsträng som ändrar värde till ett av dess närliggande värden. I Figur 6.1 illustreras hamming distance för binary coding och gray coding där vi tydligt se att en liten föränding av det numeriska värdet av gray coding resulterar i en liten förändring i hamming distance vilket är önskvärt.



Figur 6.1: En jämförelse av hamming distance mellan binary coding och gray coding.

För att tydligare påvisa vad som menas med hamming distance har vi i Tabell 6.1 representerat de numeriska värdena noll till och med sju med hjälp av de olika kodningsteknikerna. En fetstilt siffra representerar en förändring gentemot det föregående numeriska värdet, antalet förändringar mellan två värden motsvarar hamming distance mellan de olika värdena vilket vi grafiskt visade i Figur 6.1.

Numeriskt	Binary	Gray
0	000	000
1	00**1**	00**1**
2	0**10**	0**1**1
3	01**1**	01**0**
4	**100**	**1**10
5	10**1**	11**1**
6	1**10**	1**0**1
7	11**1**	10**0**

Tabell 6.1: Binary och gray coding.

6.2 Population

En population består av en mängd individer som genom evolution utvecklas för att på så sätt anpassa sig efter rådande förhållanden. Inom GA utgör en population en mängd tänkbara lösningar som genom evolution är tänkt att utvecklas för att frambringa bättre lösningar. Storleken på populationen är en viktig faktor för effektiviteten, en större population kan täcka av en större del av tillståndsrymden. Samtidigt medför en större population att beräkningstiden för varje generation förlängs vilket i slutändan kan resultera i sämre prestanda. Det är med andra ord en tradeoff mellan beräkningstiden för varje generation samt hur många generationer som kommer att behöva genereras innan en tillräckligt bra lösning erhålles. I de fall populationen är liten kan vi öka chansen för mutation (se kapitel 6.5.2) för att på så sätt tvinga algoritmen att söka av ett större område av sökrymden. [7]

Vid skapandet av initialpopulationen, den population som algoritmen startar med, bör målet i de flesta fallen vara att generera tillstånd med en så stor spridning över den totala tillståndsrymden som möjligt. Ofta slumpas tillstånden ut för att på så sätt försöka uppnå en god spridning och en enhetlig bild av sökrymden. I de fall man har tillgång till en heuristisk funktion kan initialpopulationen riktas in för att redan från början koncentrera sig på områden där goda lösningar verkar ha större chans att förekomma. Nackdelen med detta senare sätt att välja initialpopulation är att det kan lura in algoritmen på fel spår, vilket ofta leder till minskad effektivitet. [7]

6.3 Lämplighetsfunktion

Lämplighetsfunktionen, även kallad fitnessfunktionen, är kanske en av de viktigaste byggstenarna hos en genetisk algoritm då den ligger till grund för själva evolutionsprincipen inom GA. Likt den objektiva funktion som användes vid heuristisk optimering är fitnessfunktionens uppgift att avgöra hur bra ett tillstånd, kromosom, är i förhållande till den optimala lösningen. Fitnessfunktionen är, vilket vi nämnde ovan, en central del i evolutionsmekanismen då den används av urvalsoperatorerna, reproduktionsoperatorerna samt kan även förekomma som en faktor vid mutation. Fitnessfunktionens uppgift är således att mappa en kromosom till ett reellt tal vilket i sin tur används för att bedöma hur bra den specifika kromosomen är som en lösning. [7]

Vid implementation av fitnessfunktionen krävs en djup insikt om den genetiska algoritmens arbetssätt samt en god förståelse för själva problemet. Det är vidare viktigt att funktionen är effektiv i avseende på korrekthet, men även i avseende på exekveringstid. I många fall kommer nämligen den största delen av den genetiska algoritmens exekveringstid spenderas på evaluering av tillstånd med hjälp av fitnessfunktionen. Ett sätt att dra ner på den tid som spenderas på evaluering är att använda så kallad inkomplett evaluering, där evalueringen endast är ett uppskattat eller relativt värde. Enda kravet är i dessa fall att algoritmen fortfarande kan skilja en sämre lösning från en bättre. [3]

I många fall ger fitnessfunktionen för litet utslag vid en förbättring av kromosomens gener. Flera metoder för att skala om fitnessvärdet existerar och de tre vanligaste är linjär skalning, sigma trunkering samt exponentiell skalning. Linjär skalning ger

$$f_i' = a * f_i + b$$

där a och b är konstanter ofta valda på så sätt att det nya fitnessvärdet speglar större förändringar vid nära optimala lösningar. Ett problem är att linjär skalning kan införa negativa fitnessvärden vilket leder till specialfall. Sigma trunkering är en vidareutveckling av linjär skalning där negativa värden undviks, matematisk ger det

$$f_i' = f_i + (\bar{f} - \alpha * \sigma)$$

där α är ett litet heltal, σ är populationens standardavvikelse och $\bar{f}$ är populationens genomsnittliga fitnessvärde. Eventuella negativa fitnessvärden sätts vanligtvis till noll. Den exponentiella skalningen är helt enkel

$$f_i' = f_i^k$$

där k är ett decimaltal större än ett. [8]

6.4 Urvalsoperatorer

Urvalsoperatorer syftar till att välja ut de individer hos en population vilka skall få chansen att föra sina gener vidare till nästa generation. Urvalsoperatorerna kan delas in i två typer,

sannolikhetsbaserad och rationell[7]. Sannolikhetsbaserade medför att även de mindre lämpade kromosomerna får en chans att föra sina gener vidare medan rationella typer endast låter de mest lämpade föra sina gener vidare. Given samma population kommer ett rationellt urval alltid välja samma individer för reproduktion varje gång medan ett sannolikhetsbaserat urval kan välja ut olika individer för reproduktion vid olika tillfällen. Den rationella typen tenderar att ha kortare exekveringstid än den sannolikhetsbaserade. Implementation av de rationella urvalsoperatorerna tenderar dessutom att resultera i snabbare koncentration av populationen inom ett mindre område vilket både kan vara en fördel och nackdel. [3]

Det finns en rad olika tekniker och vi ska i detta kapitel gå igenom de vanligaste och mest förekommande.

6.4.1 Slumpmässigt

Slumpmässigt urval är den enklaste typen där fitnessfunktionen ignoreras helt varpå alla kromosomer får en lika stor chans att väljas ut för reproduktion, bra som dålig.

6.4.2 Proportionellt

Proportionellt urval var en tidig och vanligen implementerad teknik som sedan dess har minskat i popularitet. Denna urvalsteknik resulterar i att en mer lämpad individ, lösning, får en större chans att föra sina gener vidare gentemot en sämre. Sannolikheten för att en individ skall väljas står i direkt proportion till dess fitness i förhållande till populationens sammanlagda fitnessvärde. Matematiskt uttryckt ger detta

$$P(x_n) = \frac{f(x_n)}{\sum_{i=1}^{N} f(x_i)}$$

där x är en individ i populationen, f är fitnessfunktionen, N är antalet individer i populationen och slutligen är P sannolikheten för att välja en specifik individ för reproduktion. Detta kan resultera i att en eller flera individer helt dominerar urvalet och på grund av detta implementeras ibland en begränsning för hur många gånger en specifik individ får bli vald för reproduktion.

[7] I flertalet böcker kallas dessa två typer för stokastiska (stochastic) respektive deterministiska (deterministic).

6.4.3 Turnering

Turneringsurval är en metod som är tänkt att minska chansen för att mindre lämpade individer ska kunna föra sina gener vidare, utan att någon av de mest lämpade individerna blir för dominant i efterföljande generationer. Urvalet kan gå till på flertalet sätt men vanligen väljs slumpmässigt k (vanligtvis två) individer från populationen varpå dessa tävlar mot varandra om chansen att föra sina gener vidare. Det finns ett flertal varianter på hur denna tävling går till och vi skall här ta upp tre metoder.

Den enklaste metoden för tävling är helt enkelt att låta den individ med högst fitnessvärde vinna [7]. En något mer utvecklad metod är att införa ett gränsvärde j, på låt säga 0.75, varpå vi slumpar ett värde r där $0 \leq r \leq 1$. I de fall $r < j$ väljs den individ med högst lämplighet ut för reproduktion och i övriga fall ges chansen till den mindre lämpade individen [4]. Den tredje och sista metoden vi tänkt ta upp är ursprungligen hämtad från simulated anneling

$$P(m,n) = \frac{1}{1 + e^{\frac{f(m)-f(n)}{T}}}$$

där m och n är de två tävlande individerna, f är fitnessfunktionen, T är den sjunkande temperaturen (se kapitel 5.2) och P är sannolikheten för att välja individ n för reproduktion [8].

6.4.4 Rankbaserat

Rankbaserat urval skiljer sig från de urvalsmetoder vi nämnt ovan då chansen att bli vald inte är direkt kopplat till en individs fitness. Vid rankbaserat urval rangordnas populationen efter individernas individuella fitness, där en hög rank motsvarar ett högt fitnessvärde. Individens rank avgör sedan dess chans att bli vald för reproduktion till skillnad mot föregående metoder vilka direkt baserade sig på individens fitnessvärde. Sannolikheten för att en individ ska bli vald kan således ses som en funktion av dess rank. Den här funktionen kan se ut på en mängd olika sätt, men delas ofta in i grupper om linjära och icke-linjära.

En enkel linjär funktion för detta ändamål kan se ut på följande sätt

$$P(r) = a - (r-1)b$$

där r är en individs rank, α är sannolikheten för att den bästa individen skall väljas, β är en skalfaktor och P är sannolikheten för att en individ med en specifik rank ska väljas. Vidare skulle en icke-linjär funktion kunna se ut på följande vis

$$P(r) = \lambda(1-\lambda)^{r-1}$$

där λ är en faktor som avgör hur snabbt sannolikheten för urval ska avta med sämre rank. [8]

6.4.5 Elitiskt

Elitiskt urval skiljer sig från de övriga metoder då den aldrig används ensam utan som ett komplement till de övriga metoderna för att öka effektiviteten hos den genetiska algoritmen. Idén med elitism är att låta ett antal individer överleva mellan generationer för att på så sätt säkerställa att den eller de bästa individerna hos en population inte försämras genom evolutionsmekanismerna. Till evolutionsmekanismer räknas reproduktionsoperatorer som exempelvis cross-over och mutation (se kapitel 6.5). Detta görs enkelt genom att den bästa eller några av de bästa individerna kopieras direkt till nästa generation.

6.5 Reproduktionsoperatorer

Såhär långt har vi tagit upp representation, population, lämplighet och urval. Men en genetisk algoritm fungerar inte med enbart detta, utan det måste till något som för gener vidare mellan generationer. Nya individer måste generas på något sätt, och detta sker på liknande sätt som i naturen genom parning. En heuristisk algoritm kan på ett enkelt sätt generera en individs barn efter ett fördefinierat och problemspecifikt system. Reproduktionen hos heuristiska algoritmer kan ses som en asexuell reproduktion, där varje barn är en kopia på föräldern med en smärre förändring i någon gen. En genetisk algoritm däremot kan inte givet en specifik individ direkt avgöra vilka dess barn eller efterföljare är. De genetiska algoritmerna använder sig tillskillnad från heuristiska algoritmer vanligtvis av en sexuell reproduktion vilket innebär att föräldrar kombineras för att generera avkomma vilket ger att mängden möjliga barn ökar kraftigt.

Vi ska i detta kapitel ta upp de två vanligaste operatorerna för reproduktion, cross-over och mutation, samt några olika varianter av dessa.

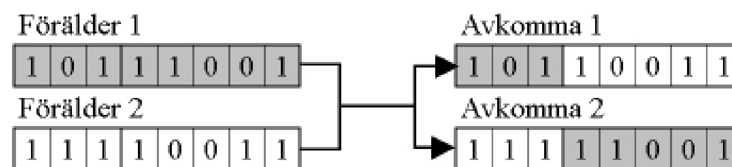
6.5.1 Cross-over

Cross-over är en typ av reproduktionsoperator vilken har till uppgift att kombinera generna från två föräldrar för att skapa nya individer vilka kommer att utgöra nästa generation. Då två

föräldrar valts ut för reproduktion sker cross-over mellan dem med en viss sannolikhet *p*, och i övriga fall förs båda föräldrarna vidare till nästa generation [7]. Vanligtvis resulterar en cross-over med två föräldrar också i två avkommor för att på så sätt garantera att inga gener går förlorade i generationsskiftet. De typer av cross-over vilka vi kommer att ta upp nedan kan inte appliceras på alla typer av problem, exempelvis som i fallet med TSP. I vissa specialfall kommer nämligen dessa traditionella metoder för cross-over bryta mot representationens restriktioner om inte dessa tas i beaktning. I de flesta fallen har nämligen representationen av en kromosom en mängd regler som definierar kromosomernas utseende. Vid exempelvis TSP måste varje stad representeras en och endast en gång i varje kromosom, och en traditionell cross-over ignorerar detta faktum totalt. Användningen av traditionell cross-over på TSP kommer således leda till införandet av ogiltiga kromosomer vid generationsskifte. Vi kommer att ta upp mer om användandet av cross-over vid TSP i kapitel 7.2.2, och riktar här in oss mer på de mer allmänna metoderna.

6.5.1.1 One point

One point cross-over är den enklaste typen av cross-over där en slumpad position vid varje ny reproduktion avgör hur avkommans sammansättning av gener kommer att se ut. Positionen vilken slumpats ut kommer att avgöra en gräns och de gener som kommer efter denna gräns kommer att byta plats med partnerns motsvarande gener för att på så sätt generera två avkommor. Sin enkelhet till trots, används inte one point cross-over särskilt mycket i dagens GA applikationer [4].

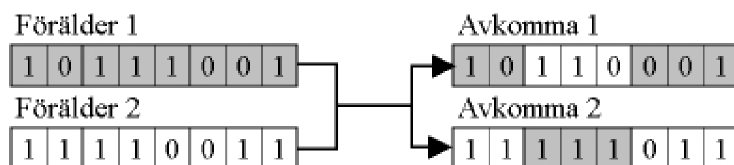


Figur 6.2: Exempel på one point cross-over, där den slumpgenererade positionen är tre.

6.5.1.2 Two point

Two point är, tillsammans med uniform, en av de på senare tid mest använda metoderna för cross-over och är ett sätt att komma runt en del av problemen som one point cross-over dras med [4]. Tekniken som används för two-point är väldigt lik one-point med skillnaden att vi nu

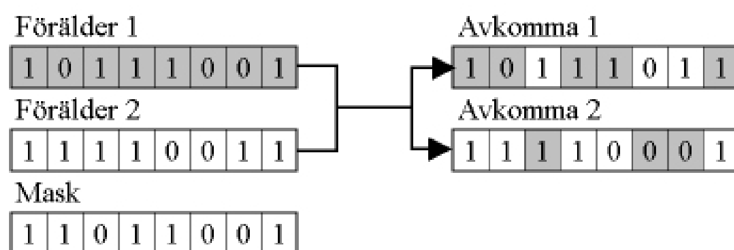
arbetar med två slumpartade positioner. Generna som ligger i substrängen mellan de slumpade positionerna skiftas med partnerns motsvarande gener för att producera avkomma.



Figur 6.3: Exempel på two point cross-over, där de slumpgenererade positionerna är två respektive fem.

6.5.1.3 Uniform

Uniform cross-over skiljer sig från de ovanstående metoderna då den använder sig av en slumpgenererad mask istället för positioner. Masken är en sträng av ettor och nollor, där varje bit motsvarar en gen i kromosomen. Bitarna avgör från vilken förälder en gen skall hämtas, där ettor och nollor representerar de två olika föräldrarna, vilket resulterar i en korsning med slumpmässigt antal korsningspunkter.



Figur 6.4: Exempel på uniform cross-over.

6.5.2 Mutation

Mutation är en nödvändig metod för att tillföra nya genetiska egenskaper hos en population. Utan chans till mutation är vi helt beroende av att initialpopulationen har alla de genetiska egenskaper som krävs för att nå vårt önskade resultat samt att vi inte förlorar någon av dessa under evolutionens gång. Mutation förhindrar dessutom att evolutionen stagnerar med tiden för att på så sätt försäkra oss om möjligheten till fortsatt evolution. Forskning har visat att sannolikheten för mutation bör vara som högst i början av evolutionen för att därefter successivt avta, en liknande stagnering har vi tidigare sett hos temperaturen i simulated annealing [7].

Vi ska i detta kapitel ta upp två vanliga typer av mutation – random och inorder - samt ge konkreta exempel på dessa, det skall dock nämnas att det finns flertalet metoder för mutation som vi inte har för avsikt att ta upp i denna uppsats.

6.5.2.1 Random

Vid random mutation utsätts varje enskild gen i en kromosom för en viss sannolikhet att muteras. Denna sannolikhet är i regel väldigt liten, för att undvika att bra lösningar förstörs i evolutionsprocessen. Dock kan sannolikheten för mutation vara större i början.

Figur 6.5: Random mutation med tre mutationer.

6.5.2.2 Inorder

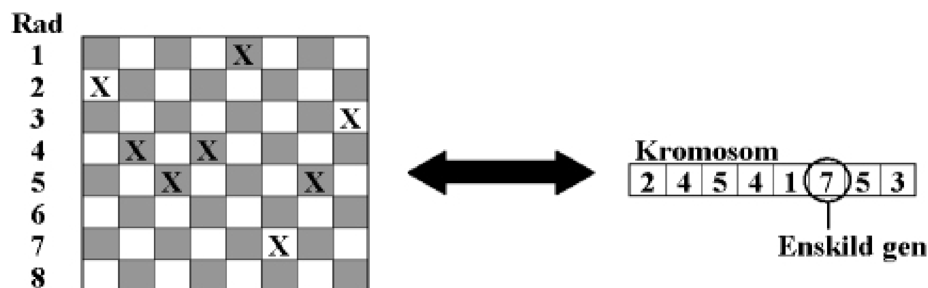
Inorder mutation liknar random mutation till stor del, men mutation kan bara ske inom ett slumpartat slutet intervall i en gen. Två slumpgenererade positioner avgör inom vilka gränser mutation kan ske, och inom detta begränsade intervall tillämpas random mutation.

Figur 6.6: Inorder mutation med intervall från 3 till 5 och 2 mutationer.

6.6 Exempel

För att visa hur en genetisk algoritm fungerar och hur dess olika delar interagerar med varandra, ska vi i detta kapitel ge ett litet exempel på ett generationsskifte i en enkel genetisk algoritm. Som problem har vi valt 8-queens problem, som det går att läsa mer om i kapitel 4.2, då det är lätt att illustrera.

Det första som måste bestämmas är hur ett tillstånd, eller individ, ska representeras i den genetiska algoritmen. Varje kromosom kan exempelvis innehålla åtta gener, där varje gen representerar en kolonn på schackbrädet. Genom denna representation skulle alla tillstånd där två eller flera drottningar står på samma kolonn automatiskt elimineras, vilket skulle resultera i en mindre tillståndsrymd. Det ska påpekas att det skulle fungera på motsvarande sätt om istället varje gen representerade en rad på schackbrädet. En annan möjlighet för att begränsa tillståndsrymden ytterligare är att inte tillåta någon drottning att stå på samma rad eller kolonn som en annan. Denna implementation skulle dock kräva specialdesignade varianter av både cross-over och mutation. För att inte göra det hela alltför komplicerat har vi därför valt att låta varje gen i en kromosom representera en drottning på motsvarande kolonn, och att inte begränsa de individuella genernas värden på något sätt. Varje gen innehåller ett heltal vilket representerar den rad drottningen befinner sig på.



Figur 6.7: Representation av 8-queen problem som vi kommer att använda nedan i vårt exempel.

Till populationsstorlek har vi valt sju, mest för att hela populationen ska gå att illustrera men samtidigt inte vara för liten, men en verklig implementation skulle dra nytta av en betydligt större population.

Fitnessfunktionen vi kommer att använda oss av i detta exempel är förhållandevis enkel och ser ut på följande sätt

$$f(k) = \left(\binom{d}{d-2} - \frac{1}{2} \sum_{i=1}^{d} h(k, g_i) \right)^3$$

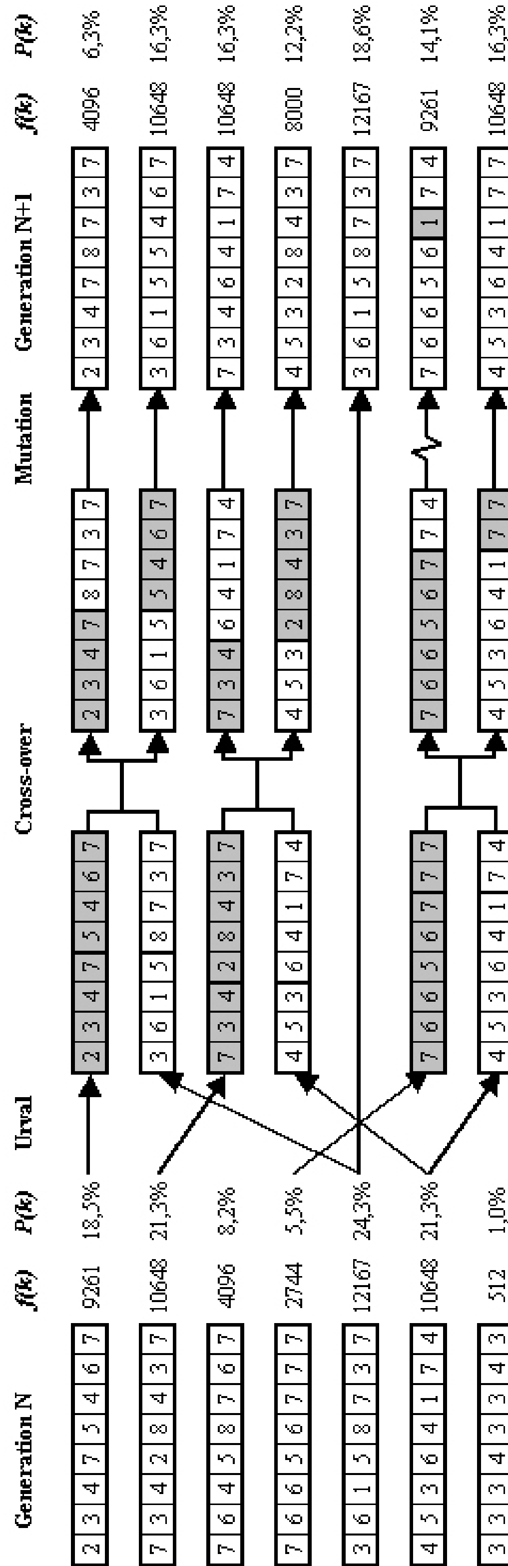
där k är aktuell kromosom, d är antalet drottningar, g_i är gen i tillhörande kromosom k, och $h(k,g_i)$ är det totala antalet krockar för en specifik gen, drottning, g_i. För att få lite större

spridning på fitnessvärdena har vi även valt att införa en exponentiell skalning. För den uppmärksamme läsaren framgår det kanske att en krock inte enbart utgörs av direktträffar utan även en bakomliggande drottning räknas som en krock. Vi anser detta vara en bra metod eftersom fitnessfunktionen då speglar problemet på ett bättre sätt.

För urvalet har vi valt att använda ett proportionellt urval direkt baserat på fitnessvärdena, beskrivet i kapitel 6.4.2, samt elitism, kapitel 6.4.5, där den bästa individen ur en population automatiskt går vidare till nästa generation.

Reproduktionen sker med hjälp av one-point cross-over samt en liten chans för mutation. Anledningen till valet av cross-over är helt enkelt på grund av att det är enklast att illustrera.

I Figur 6.8 visas ett generationsskifte från en generation N nästa generation N+1. $f(k)$ är fitnessvärdet för en kromosom k returnerat av fitnessfunktionen och $P(k)$ är sannolikheten för en specifik kromosom k att bli vald till reproduktion.



Figur 6.8: Ett exempel på ett generationsskifte i en Genetisk Algoritm.

7 Experiment

Som vi tidigare nämnt, har vi utfört ett experiment där vi har jämfört två olika algoritmers prestanda vid lösning av Travelling Salesman Problem (kapitel 4.1). Som titeln på uppsatsen antyder har vi valt en algoritm som bygger på heuristik, simulated annealing, samt en algoritm som bygger på evolutionär beräkning. Målet med experimentet var att utröna hur dessa två algoritmer står sig gentemot varandra prestandamässigt vid applicering på samma problem samt att skapa en djupare förståelse för algoritmernas funktion och hur de egentligen implementeras.

Vi kommer att börja med att definiera problemet som ska lösas, för att sedan gå in på implementationen av algoritmerna och de val vi gjort i samband med denna. Kapitlet avslutas med en presentation av experimentets resultat och en diskussion kring detta.

7.1 Problemdefinition

Det problemet som vi har valt för vårt experiment är att48 som kommer från TSPLIB, vilket är en samling av TSP problem med eventuella lösningar. Att48 är en av TSPLIBs många problem för vilken den optimala lösningen redan är framtagen, och är definierad som 48 av USA:s staters huvudstäder. Problemet algoritmerna hade till uppgift att lösa var att på så kort tid som möjligt ta fram en lösning vilken är minst 95% av den optimala sträckan. En lösnings förhållande till den optimala lösningen har vi valt att definiera på följande sätt

$$f(s) = \frac{opt}{s}$$

där s är sträckan av lösningen och opt är den optimala sträckan. På grund av detta tillvägagångssätt blev det ett grundläggande krav att den optimala lösningen redan var framtagen. Ett alternativt tillvägagångssätt hade varit att uppskatta den optimala sträckan och utifrån denna uppskattning på samma sätt som ovan avgöra när vi nått en tillräckligt god lösning. Problemet med detta senare sätt är att det är möjligt att överskatta den optimala lösningen, dvs uppskatta den optimala lösningen till ett värde som är lägre än den faktiska optimala sträckan, vilket kan leda till att algoritmens mål aldrig nås. Förutom dessa två

nämnda metoder finns det egentligen bara ett val kvar – brute force, som använder en blind sökmetod – vilket varken har med heuristik eller genetisk beräkning att göra.

Tanken med vårt experiment var från början att applicera algoritmerna på flertalet olika TSP-problem av varierande storlek, men på grund av tidsbrist var vi tvungna att koncentrera oss på enbart ett TSP-problem och valet föll då på att48. Anledningen till att det blev just detta problem är helt enkelt att det på intet sätt är för litet eller för stort i förhållande till experimentets tidsram.

7.2 Implementation

Valet av programspråk för implementation av algoritmerna föll på C++ dels av den anledningen att det erbjuder den funktionalitet vilket krävs för uppgiften samt dess goda prestanda. Vi har också strävat efter att låta de båda algoritmerna dela på den kod som inte är specifik för själva algoritmen för att på så sätt eliminera eventuella skillnader i prestanda orsakade av icke algoritmspecifika delar. Till denna gemensamma kod hör funktionalitet för inläsning av TSP data samt beräkning av sträckor vilka erbjuds av klasserna Distances och Matrix. Efter inläsning av städer beräknas en avståndsmatris med avstånden mellan samtliga par av städer, och tack vare denna lösning behöver inte avstånden beräknas varje enskild gång utan kan enkelt nås genom matrisen. Beräkningen av avståndsmatrisen ligger utanför algoritmernas exekvering och påverkar således inte tidsmätningarna.

Nedan följer specifika implementationsdetaljer för respektive algoritm.

7.2.1 Simulated Annealing

Algoritmen för simulated annealing implementerades utifrån en grundstomme vi hittade i Zbigniew Michalewicz och D. B. Fogels bok [3]. Vi använde oss av två klasser för detta, State respektive SA. Klassen State representerar ett tillstånd, en cykel i TSP-grafen, och genererar dessutom sin efterföljare, ett närliggande tillstånd, genom att slumpartat byta ordningen på två städer i cykeln. Klassen SA ansvarar för temperaturminskningen och accepterandet av nästa tillstånd. Bättre tillstånd accepteras alltid medan sämre tillstånd accepteras med sannolikheten

$$p = e^{\frac{eval(v_c) - eval(v_n)}{T}}$$

där *eval(v)* är algoritmens evalueringsfunktion och motsvarar ett tillstånds totala sträcka, v_c är det nuvarande tillståndet, v_n det nya tillståndet och T är temperaturen. Temperaturen *T* initierades till $|(d_1+d_2+d_3+d_4) / \ln(0.5)|$ där d_1 är det längsta avståndet mellan två städer och d_2 det nästlängsta avståndet osv. Anledningen till att det är just de fyra längsta avstånden är att skillnaden mellan ett tillstånd och ett närliggande maximalt kan skilja på fyra då endast två städer bytt plats. Detta resulterar i att sannolikheten för att välja ett sämre tillstånd ursprungligen inte kan vara mindre än 50%. Temperaturen *T* avtar sedan procentuellt från sin initialtemperatur ner mot mintemperaturen beroende på avsvalningshastigheten *a* vilken beräknas genom ekvationen

$$a = e^{\frac{\ln(\min T) - \ln(initT)}{i}}$$

där *minT* är minsta tillåtna temperaturen, *initT* är inititaltemperaturen, *i* är antalet iterationer algoritmen har på sig för att *T: initT→minT*. *minT* har vi i våra försök valt att sätta till 0.1 då det är tillräckligt litet för att eliminera dåliga val i algoritmens slutskede och samtidigt stort nog att inte resultera i en för snabb avtagning av *T*. Att inte T avtar för fort är viktigt eftersom algoritmen skall kunna täcka av en så stor del av sökrymden som möjligt och på så sätt undvika att fastna på ett lokalt maxima.

I de fall *T* = *minT*, dvs när algoritmen gjort *i* iterationer, har vi valt att räkna detta som att algoritmen fastnat på ett lokalt maxima och avbryter därefter algoritmen automatiskt. Detta leder till att varje testkörning av algoritmen med en specifik inställning kommer resultera i ett visst antal ”hängningar”, dvs när algoritmen fastnat i lokala optimum. Det hade självklart varit möjligt att låta algoritmen fortsätta köra tills den eventuellt hittar en lösning. Men om algoritmen verkligen fastnat i ett lokalt optimum när *T* har nått sitt minvärde kommer den i sådana fall att vara fast i en oändlig loop. Detta beror på att ett väldigt lågt *T* medför en stort sett obefintlig chans att välja ett sämre tillstånd vilket krävs för att komma ur lokala optimum. Allt detta resulterar i sin tur i oändliga tider. En oändlig tid kan inte redovisas som en del av ett genomsnitt, då det skulle resultera i ett oändligt stort genomsnitt, även om alla de övriga tiderna är låga och därmed ändliga. Därför har vi valt att inte inkludera hängningar i tidsmätningarna, dock kommer vi redovisa antalet hängningar vid sidan om tidsmätningarna.

7.2.2 Genetisk algoritm

Implementationen av den genetiska algoritmen innebar att vi ställdes inför en mängd olika valmöjligheter vilket oundvikligen speglas på algoritmens slutgiltiga effektivitet. Senare i

detta kapitel kommer vi att gå igenom dessa val och de bakomliggande anledningarna men vi önskar även poängtera att vårt lösningssätt på intet sätt behöver vara det optimala. Grundstommen i vår implementation är hämtad från idéerna vi beskrev i kapitel 6 med undantag för crossover vilken kräver ett problemspecifikt tillvägagångssätt för att undvika införandet av ogiltiga kromosomer. Implementationen består av tre klasser, Chromosome, Population och GA.

Klassen Chromosome representerar ett tillstånd, individ, med funktionalitet för att orsaka en genförändring, så kallad mutation. Ur representationssynvinkel lagras TSP-cykeln i en integer array där varje stad förekommer en och endast en gång och ordningen på städerna i arrayen anger ordningen i cykeln. Detta kallas path representation och är en av tre vanliga vektorrepresentationer för TSP[8][8]. Det är även den enklaste och mest rättframma av de tre. Varje representation har sina egna specifika reproduktionsoperatorer som går att applicera endast på den typen av representation de tillhör.

Det enklaste sättet att utföra mutation i en genetisk algoritm är att helt enkelt ändra en gens värde slumpmässigt någonstans i en kromosom, men då detta skulle införa ogiltiga vägar i fallet med TSP, krävs en lite annorlunda form av mutation. En mutation går därför till på så sätt att två gener, städer, helt enkelt byter plats med varandra i arrayen.

Populationsklassen innehåller ingen funktionalitet i sig utan dess huvudsakliga uppgift är helt enkelt att representera en grupp individer som en enhet. Ett objekt av populationsklassen representerar en generations individer.

Algoritmens huvudklass, GA, innehåller övrig funktionalitet såsom urvalsoperator, reproduktionsoperatorer såsom crossover och elitism och slutgiltigen fitnessfunktionen. Fitnessvärdet $f(x)$ för en individ x i populationen beräknas enligt funktionen

$$f(x) = \left(\frac{\sum_{i=1}^{k} dist(x_i)}{dist(x)} \right)^3$$

där $dist(x)$ är längden av TSP-cykeln hos en individ x och k är antal individer i populationen. Värdet skalas sedan om genom att upphöjas till tre, för att öka spridningen på populationens fitnessvärden och på så sätt gynna de mer lämpade individerna vid urvalet. Vi har använt oss

[8] De övriga två representationssätten är adjancency representation samt ordinal representation [8].

av vanligt proportionellt urval, där sannolikheten *p(x)* för att en viss individ *x* ska bli vald definieras enligt funktionen

$$p(x) = \frac{f(x)}{\sum_{i=1}^{k} f(x_i)}$$

där *f(x)* enligt ovan är en viss individ *x* fitnessvärde och *k* är återigen antal individer i populationen.

Till reproduktionsoperatorer har vi valt elitism, cross-over samt mutation, vilket vi beskrivit ovan. Vi har använt oss av 10% elitism, vilket innebär att de 10% bästa individerna i populationen överförs direkt till nästa generation innan varken cross-over eller mutation applicerats. Vid en populationsstorlek på 100 skulle detta innebära att de 10 mest lämpade individerna överförs automatiskt och oförändrade vid varje generationsskifte. Att vi valde en procentsats beror på att vi behövde låsa elitismen för att på så sätt minska antalet variabler i algoritmen, och det verkade rimligt att göra elitismen proportionerligt mot populationsstorleken. Att det sedan blev just 10% beror på att vi behövde ett värde som inte var för litet och samtidigt inte för stort, och vi anser att 10% uppfyller detta krav. Vi har dock inte undersökt detta genomgående, och om vi hade gjort det är det mycket möjligt att vi hade funnit bättre värden.

För path representation finns det tre huvudsakliga metoder för att utföra cross-over. Dessa är PMX (Partially-mapped cross-over), OX (Order cross-over) samt CX (Cycle crossover). Alla dessa är specifika för TSP, och bygger på att inga ogiltiga cykler ska kunna skapas. För vårt experiment har vi valt OX, därför att den enligt tidigare experiment har visat sig vara 11% mer effektiv än PMX samt 15% mer effektiv än CX [8]. Eftersom OX cross-over inte är helt trivialt, har vi valt att beskriva det ingående med hjälp av bilder. Det går att läsa mer om detta i [8].

Med OX cross-over byggs avkomman genom att en delsekvens av ena förälderns cykel förs direkt över till avkomman, vartefter städer flyttas över från andra föräldern i ordning så att inga dubbletter uppstår i avkomman. Antag att vi har två föräldrar, *f1* och *f2*, där vi valt ut två brytpunkter för att markera vart delsekvensen börjar och slutar.

f1 | 3 4 8 | 6 1 5 | 7 2 |

f2 | 5 7 1 | 3 2 6 | 4 8 |

Städerna mellan de två brytpunkterna kopieras först till respektive avkomma, *a1* och *a2*.

a1	x x x	6 1 5	x x

a2	x x x	3 2 6	x x

Ett x i en avkomma betyder helt enkelt att det inte finns något värde i den genen än.

Nästa steg är att kopiera in värden med början vid den andra brytpunkten från *f2* in i *a1* och motsvarande mellan *f1* och *a2*. För att enklare illustrera detta väljer vi ut sekvenserna av städer *f1'* och *f2'* från de båda föräldrar med början vid andra brytpunkten.

f1'	7 2 3 4 8 6 1 5

f2'	4 8 5 7 1 3 2 6

Nu elimineras de städer från *f1'* och *f2'* som redan finns i *a2* respektive *a1*, för att undvika införandet av ogiltiga tillstånd. Detta ger oss *f1"* och *f2"*.

f1"	7 4 8 1 5

f2"	4 8 7 3 2

Slutligen kopieras alla värden från *f1"* in i *a2* med början från andra brytpunkten i *a2*, och motsvarande sker mellan *f2"* och *a1*, vilket ger oss den färdiga avkomman.

a1	7 3 2	6 1 5	4 8

a2	8 1 5	3 2 6	7 4

Namnet order cross-over kommer från att metoden utnyttjar det faktum att städernas ordning, och inte deras enskilda positioner, är vad som har betydelse [8].

Slutligen har vi infört något vi kallar stagneringskontroll, för att undvika hängningar i algoritmen. Det är nämligen fullt möjligt att evolutionen sätts ur spel och utvecklingen stannar upp av någon anledning, och därför har vi infört en kontroll för detta. Om populationens genomsnittliga lämplighet inte förbättras över ett visst antal generationer antar vi att utvecklingen stagnerat. Antalet generationer innan en omstart äger rum beror på populationsstorleken. En mindre population ger att ett större antal generationer måste äga rum innan en omstart. Detta förhållande är naturligt då en mindre population har sämre förutsättningar att förbättras gentemot en större men samtidigt kräver ett generationsskifte för en mindre population proportionerligt mindre tid. Antalet generationer innan en omstart beräknar vi med formeln

$$o(s) = 150000 / s$$

där s är populationsstorleken. Att vi valde 150000 baserar sig på det genomsnittliga antalet generationer som krävs för att finna en lösning innan det att vi valde att implementera stagneringskontrollen. När stagnering sker slumpgenereras en ny population innan exekveringen fortsätter med denna nya population, vilket vi kallar en omstart. Anledningen till detta är enkel – vi vill undvika alltför stora variationer i tidsmätningarna som kan påverka slutresultatet i hög grad. Detta kan jämföras med den hantering av hängningar vi införde i simulated annealing. Skillnaden är att vi här genererar en ny startposition och fortsätter exekveringen, medan vi i simulated annealing avbryter algoritmen vid hängning. Vi kommer givetvis att redovisa omstarterna vid sidan om övriga mätningar för den genetiska algoritmen.

7.3 Experimentutförande

Experimentet utfördes på en PC med en Intel Pentium 4 processor på 2,4 GHz. För att försäkra oss om att inte några bakgrundsprocesser påverkade våra tidsmätningar utfördes mätningarna direkt efter datorstart med hjälp av en Windows 98 CD i ”DOS-läge”. Källkoden kompilerades med hjälp av C++ kompilatorn DJGPP 332b.

Eftersom båda algoritmerna arbetar med hjälp av slumpen valde vi att utföra ett något högre antal mätningar än vad som vanligtvis är brukligt för att på så sätt öka tillförlitligheten. Antalet mätningar som utfördes vid varje mätpunkt var 50 stycken vilket är högre än de 25-30 vilket brukar förespråkas [9].

Parametrarna vilka användes i experimentet var följande:

Simulated Annealing	Intervall	Steglängd
Avsvalningshastighet	[20000, 400000]	20000

Genetisk Algoritm	Intervall	Steglängd
Populationsstorlek	[20, 220]	20
Mutationschans	[0.01, 1]	0.1

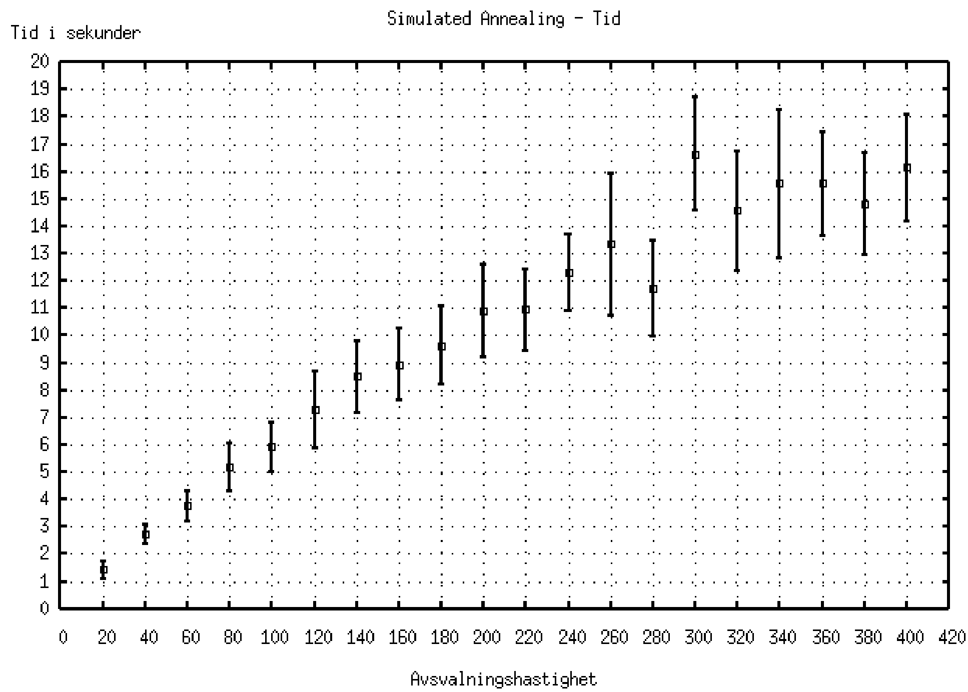
Tabell 7.1: Parameterinställningar för experimentet.

7.4 Resultat

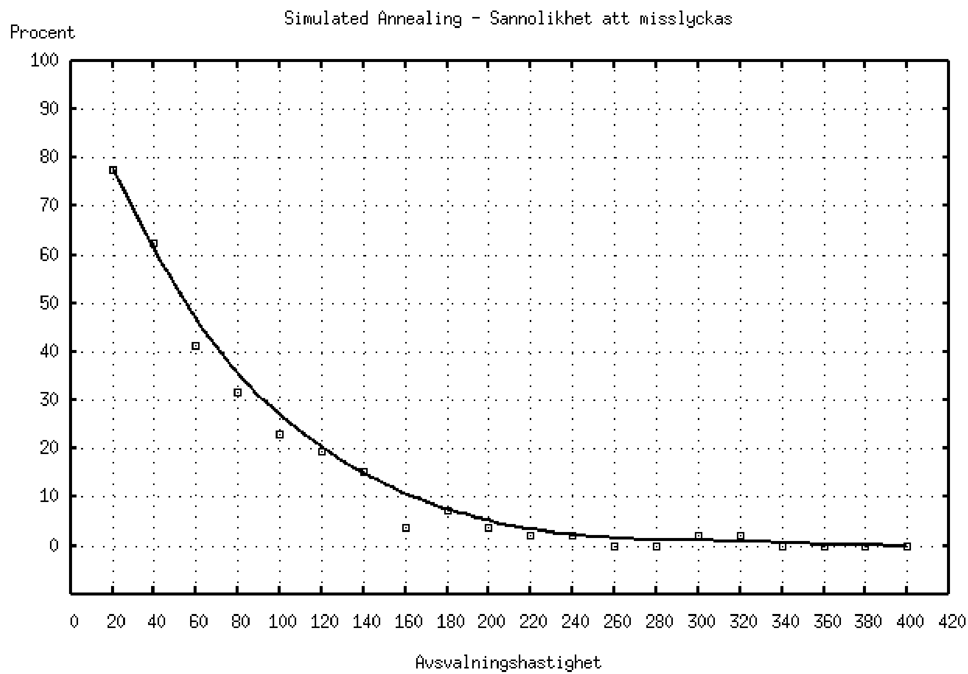
Figur 7.1 visar tiden som en funktion av avsvalningshastigheten för lösandet av att48 med hjälp av Simulated Annealing. Diagrammet visar förutom medelvärdet för mätningarna också det symetriska konfidensintervallet på 95%. Avsvalningshastigheten anges i tusental och ett lågt värde motsvarar ett hastigt avsvalningsförfarande.

Figur 7.2 visar hur sannolikheten för ett misslyckande beror på avsvalningshastigheten. De markerade punkterna motsvarar medeltalet för våra mätningar och kurvan speglar en uppskattning för att tydligare påvisa en trend. Precis som i Figur 7.1 är avsvalningshastigheten angiven i tusental och ett lågt värde motsvarar ett hastigt avsvalningsförfarande.

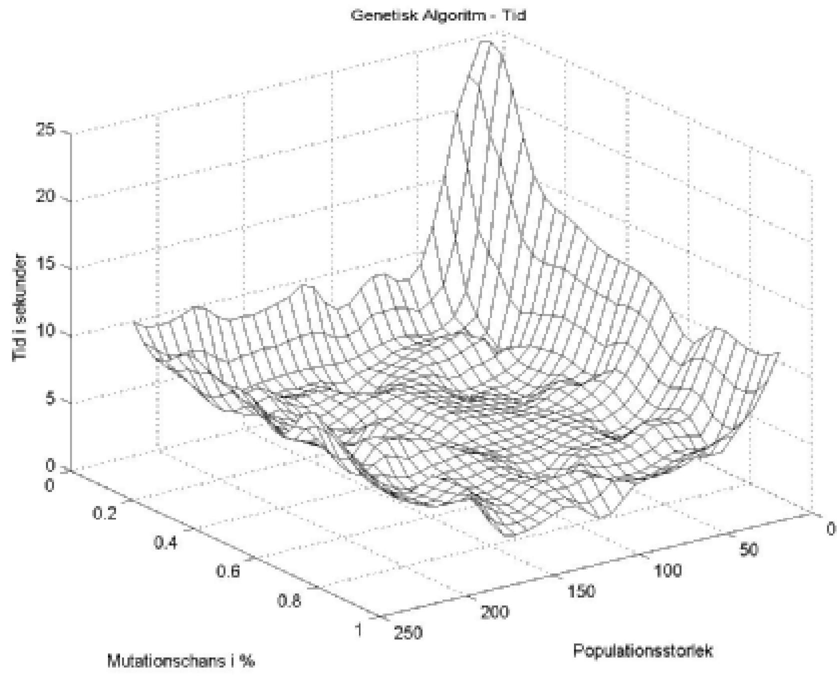
Figur 7.3 visar tidsmätningarna för vår genetiska algoritm som en funktion av populationsstorlek och chansen för mutation. På motsvarande sätt visar Figur 7.4 sannolikheten för att en omstart äger rum i den genetiska algoritmen. Då det kan vara svårt jämföra en ytgraf med en vanlig tvådimensionell graf har vi även tagit fram ett snitt ur figurerna vid 0.6% mutationschans. Dessa visas i Figur 7.5 respektive Figur 7.6 och på motsvarande sätt som i Figur 7.1 är ett symetriskt konfidensintervall på 95% angivet för tidsmätningarna.



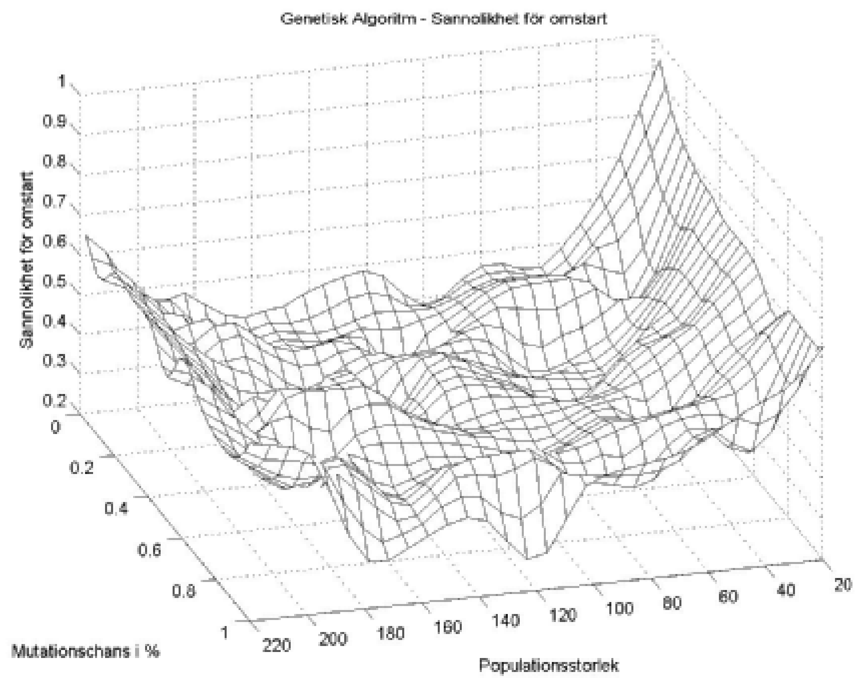
Figur 7.1: Tidsmätning av Simulated Annealing.



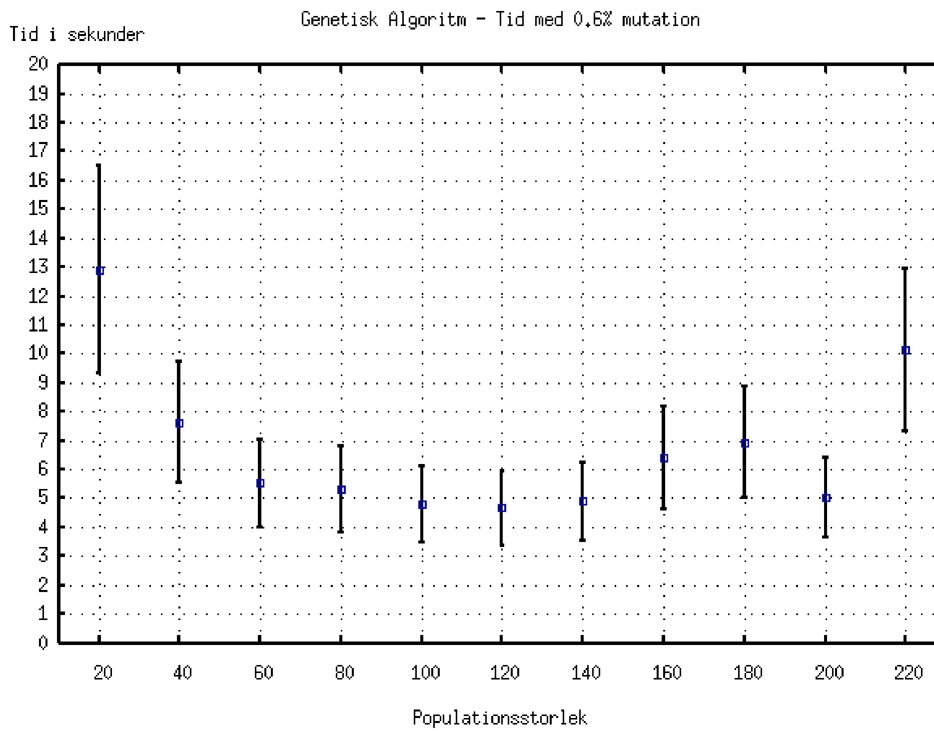
Figur 7.2: Sannolikhet för hängningar med Simulated Annealing.



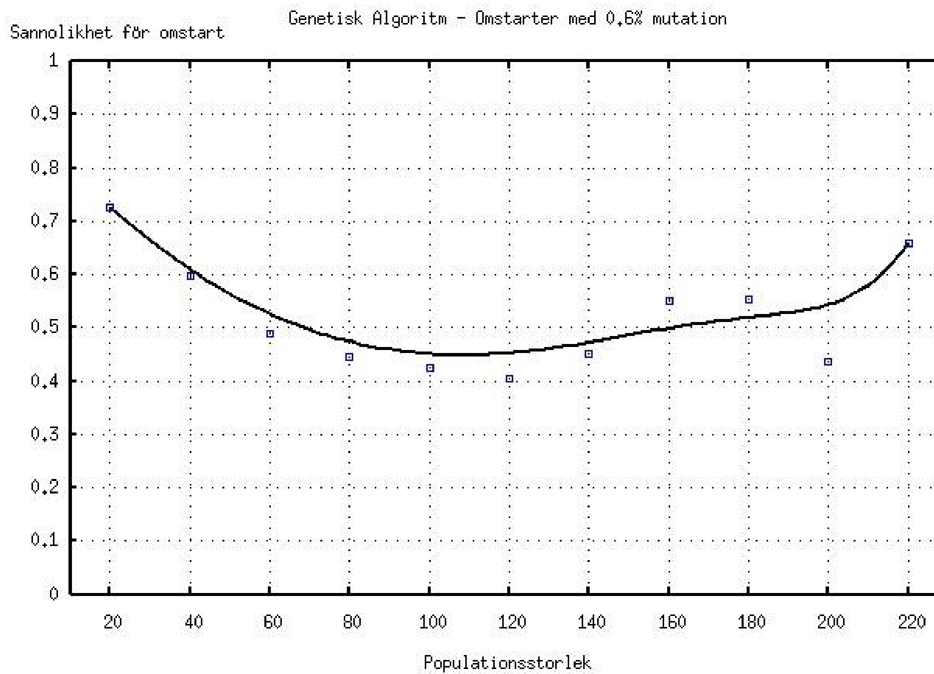
Figur 7.3: Tidsmätning av Genetisk Algoritm.



Figur 7.4: Sannolikhet för omstart med genetisk algoritm.



Figur 7.5: Ett snitt av tiden vid 0.6% mutationschans för genetisk algoritm.



Figur 7.6: Ett snitt av sannolikheten för omstart vid 0.6% mutationschans för genetisk algoritm.

7.5 Diskussion

En mycket viktig detalj i ovanstående mätningar som tåls att upprepas är att den genetiska algoritmens omstarter är inräknande i tiden för att lösa problemet. Detta gäller inte för Simulated Annealing där tiden endast avspeglar de lyckade försöken. För att vara någorlunda säker på att nå en lösning med hjälp av SA behöver avsvalningshastigheten närma sig 220000 vilket ger en medeltid på runt 11 sekunder. Jämför man denna siffra med vår GA:s bästa värden vid en population på 120 visar den ett medelvärde på strax under 5 sekunder. Den Genetiska Algoritmen är således överlag snabbare än Simulated Annealing, och det går att diskutera hur vår stagneringskontroll med tillhörande omstarter påverkar den Genetiska Algoritmens tider. Våra erfarenheter visar nämligen att GA:n tar längre tid på sig än de presenterade resultaten utan dessa omstarter, dock har vi inga siffror att visa som styrker detta på grund av tidsbrist. Under arbetets gång experimenterade vi även med avsvalnande mutationschanser och populationsstorlekar, men dessa plockades bort i slutskedet på grund av att det blev för många påverkande faktorer i algoritmen. Det ska nämnas att avtagande populationsstorlekar tenderade att förbättra algoritmens effektivitet. Det ska vidare nämnas att vi till en början hade en liknande implementering i Simulated Annealing med omstarter, men då dessa försämrade algoritmens tider avsevärt valde vi således att ta bort detta ur implementeringen och särbehandla så kallade ”hängningar”.

Experimentet visar även att mutationen i den genetiska algoritmen inte har någon avgörande påverkan på algoritmens effektivitet förutom för mindre populationer och så länge mutationschansen ligger över ett visst gränsvärde. Det går nämligen att se att ytan i Figur 7.3 är i princip platt i mutations-led förutom runt kanterna, det vill säga när populationen blir relativt liten samt när mutationschansen är nära noll. Vi vet inte hur nära noll mutationschansen ska vara för effektiviteten ska påverkas negativt, dock vet vi att det sker omkring 0.1%. På liknande sätt kan effektiviteten förväntas försämras vid tillräckligt stora mutationschanser, då slumpen får för stor inverkan på algoritmen.

Vidare går det att utläsa populationens betydelse för effektiviteten i Figur 7.3. Vi ser att både en för liten och en för hög population är negativ medan det finns ett intervall däremellan där populationsvariationen har en mindre betydelse. Anledningen till detta är att en liten population har för dålig spridning och får det således svårare att konvergera och hitta en lösning. I en för stor population däremot har en bra individ mindre att säga till om, och dess gener har därför svårare att slå igenom vid generationsskifte.

Slutligen går det att diskutera omstarternas påverkan på mutationens betydelse för effektiviteten. Det kan diskuteras huruvida omstarterna eliminerar mutationens betydelse eller ej, men det kan sägas med säkerhet att omstarterna har någon form av påverkan på betydelsen av mutationen. Även valet av OX cross-over kan här spela en faktor för mutationens betydelse. Då sättet denna skapar avkomma på förändrar ordningen på generna, kan en enkel swap-mutation egentligen inte införa något nytt som inte OX cross-over kan införa. Trots detta verkar en liten chans för mutation trots allt ha positiv inverkan på algoritmens effektivitet.

Det som talar för Simulated Annealing är dess enkelhet, då själva implementeringen var väsentligt lättare gentemot vår genetiska algoritm vilket krävde långt fler arbetstimmar att utveckla. Vidare har den genetiska algoritmen problemet med ett stort antal variabler gentemot Simulated Annealing, vilket gör att det krävs mycket mer tid för finjustering av variablernas värden för att utnyttja dess fulla kapacitet.

På grund av tidsbrist beskars experimentsdelen till att endast innefatta ett TSP-problem, en av våra ursprungliga tankar var att testa algoritmerna på en rad olika TSP-problem av varierande storlek. Det hade också varit intressant att pröva andra tekniker för reproduktion i den genetiska algoritmen för att optimera den ytterligare. På liknande sätt hade det varit intressant att se hur en avtagande populationsstorlek avspeglas på effektiviteten. Även SA algoritmen kan effektiviseras genom att ändra definitionen av ett närliggande tillstånd. Istället för att byta plats på två städer vilket resulterar i att två till fyra vägar ändras kan istället två vägar avlägsnas varpå två nya vägar införs på så sätt att inga ogiltiga lösningar skapas.

8 Sammanfattande kommentarer

Sammanfattningsvis kan det sägas att optimering med heuristik samt evolutionär beräkning är mycket intressanta metoder för lösning av väldigt stora och komplexa problem. Det kan också sägas vara lite förvånande att algoritmer som faktiskt arbetar med hjälp av slumpfaktorer kan vara så pass effektiva som experimentet faktiskt har visat. Att lösa en 48-stads TSP exakt skulle med brute force (blind sökning) ta över 10^{40} år, vilket nog kan anses olösbart då universums ålder är i storleksordningen 10^{10} år. Ändå kan Simulated Annealing och Genetiska Algoritmer hitta en 95% lösning inom loppet av några få sekunder.

När det gäller själva experimentet och förhållandet mellan Simulated Annealing och Genetiska Algoritmer, visade sig de senare vara effektivare i vårt experiment, dock visade de sig även vara svårare och mer tidskrävande att implementera gentemot Simulated Annealing. Vidare är det inte alls säkert att våra implementationer på något sätt är optimala, och tiderna skulle säkerligen kunna pressas ytterligare. Det finns även många andra intressanta algoritmer som använder sig av heuristik, bland annat Tabu search som är deterministisk till skillnad från Simulated Annealing, och därmed inte alls baserar sig på slumpen. Hur står sig denna gentemot de två algoritmerna vi testat i vårt experiment? Det finns således mycket mer som skulle kunna testas och jämföras.

Referenser

[1] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[2] Stuart Russel och Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Higher Education, 2nd edition, 1995.

[3] Zbigniew Michalewicz och D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

[4] Melanie Mitchell. *An Introduction To Genetic Algorithms*. The MIT Press, 2nd edition, 1996.

[5] George F. Luger och William A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison Wesley, 3rd edition, 1998.

[6] Kalph P. Grimaldi. *Discrete and combinatorial mathematics*. Addison Wesley, 4th edition, 2000.

[7] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley, 2002.

[8] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 3rd edition, 1996.

[9] Kerstin Vännman. *Matematisk statistik*. Studentlitteratur, Andra upplagan, 2002.