CHALMERS | GÖTEBORGS UNIVERSITET

Lösning till handelsresandeproblemet med hjälp av planfyllande kurvor

Kandidatarbete inom civilingenjörsutbildningen vid Chalmers

Simon Hall
Andreas Henriksson
Hakim Khalafi

Institutionen för matematiska vetenskaper
Chalmers tekniska högskola
Göteborgs universitet
Göteborg 2012

Lösning till handelsresandeproblemet med hjälp av planfyllande kurvor

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk fysik vid Chalmers

Simon Hall

Kandidatarbete i matematik inom civilingenjörsprogrammet Teknisk matematik vid Chalmers

Andreas Henriksson Hakim Khalafi

Handledare: Torbjörn Lundh
Examinator: Carl-Henrik Fant

Institutionen för matematiska vetenskaper
Chalmers tekniska högskola
Göteborgs universitet
Göteborg 2012

Sammanfattning

I denna rapport har vi utforskat olika metoder som löser handelsresandeproblemet med hjälp av rumsfyllande kurvor. Vi har utgått från en klassisk algoritm av Bartholdi och Platzman och förbättrat den med avseende på lösningslängd med två egna algoritmer. Vi har under arbetets gång utvecklat programvara som använder dessa algoritmer. Vi har undersökt körningstid, lösningslängd och stabilitet för de olika algoritmerna och analyserat resultaten. Rapporten innehåller även en inledande teoretisk beskrivning av planfyllande kurvor och NP-komplexitet.

Abstract

In this paper we have explored different ways of using space-filling curves to solve the travelling salesman problem. Starting with a classical algorithm by Bartholdi and Platzman we developed two new algorithms with better solutions with respect to total solution length. During the process we developed software incorporating these algorithms. We compared tour length, calculation time and stability of these methods with the classical approach, and analysed the results. A theoretical introduction to space-filling curves and NP-complexity is provided as well.

Innehåll

Förord

Denna rapport har gjorts som en del av ett kandidatarbete vid Matematiska vetenskaper på Chalmers våren 2012. Arbetet har primärt utförts av Andreas Henriksson och Hakim Khalafi vid Teknisk Matematik samt Simon Hall vid Teknisk Fysik.

Genom arbetets gång har en gemensam loggbok förts, samt tre individuella loggböcker. Dessa loggböcker innehåller detaljerad information om hur arbetet fortskridit samt tidsangivelser för individuella och gemensamma arbetstillfällen.

Då detta arbete har utförts av tre personer har alla gruppmedlemmar haft en del i de flesta arbetsmomenten. En grov arbets- och ansvarsfördelning följer nedan.

Produkt - Utveckling av vår lösare

- Huvudsakligt utvecklingsansvar - Andreas
- Assisterande programmerare - Simon

Analys - Utvärdering av vår lösare

- Stabilitetsanalys - Hakim
- Prestandaanalys - Simon
- Beräkningstidsanalys - Simon
- Kvalitativ analys - Hakim
- Jämförande med andra algoritmer - Andreas

Rapport - Sammanställning av slutrapporten

- Förord - Simon
- Bakgrund - Hakim
- Syfte - Hakim
- Teori (Beräkningskomplexitet) - Hakim
- Teori (NP-Problematiken) - Simon
- Teori (Planfyllande kurvor) - Simon
- Teori (Teoretisk optimal lösningslängd) - Simon
- Genomförande (Standard) - Andreas
- Genomförande (Punktbyte) - Andreas
- Genomförande (Smartsplit) - Andreas
- Resultat (Prestanda) - Simon
- Resultat (Stabilitet) - Hakim
- Resultat (Beräkningstid) - Simon
- Resultat (Beräkningstid) - Simon
- Slutsatser och Diskussion (Analys av Prestanda) - Hakim
- Slutsatser och Diskussion (Stabilitet) - Hakim
- Slutsatser och Diskussion (Körningstider) - Hakim

1 Inledning

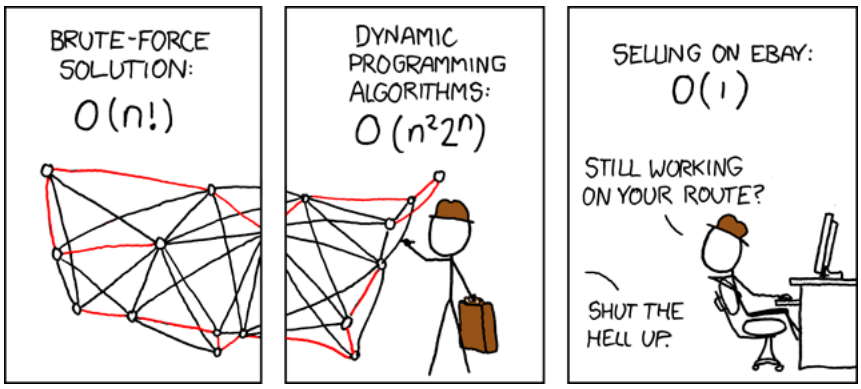
Den här studien beskriver ett sätt att lösa handelsresandeproblemet (Travelling Salesman Problem - TSP) med hjälp av en särskild typ av kurvor. Handelsresandeproblemet går ut på att hitta kortaste vägen mellan N antal punkter i planet om man skall besöka varje punkt endast en gång. Traditionellt betraktar man punkterna som städer och lösningen beskriver då kortaste vägen mellan dessa.

Det finns många andra praktiska tillämpningar än just städer. Till exempel att hitta det snabbaste och effektivaste sättet för en borr att borra ett antal hål i ett kretskort. Problemet är enkelt att beskriva men svårt att lösa. Handelsresandeproblemet tillhör nämligen klassen av problem som är så kallade NP-kompletta [1]. Grovt sett innebär detta för handelsresandeproblemet att en optimal lösning blir mycket svårare att hitta för små ökningar i storleken på problemet. Just för handelsresandeproblemet ökar antalet möjliga lösningskombinationer som n-fakultet.

1.1 Bakgrund

TSP definierades på 1800-talet av den irländska matematikern W.R. Hamilton och brittiske matematikern Thomas Kirkman. Den generella formen av TSP verkar dock inte ha studerats förrän 1930 av matematiker i Wien och på Harvard. På 50-talet ökade problemet i popularitet i olika vetenskapliga sammanhang i Europa och USA där framstående insatser gjordes av matematiker på RAND Corporation i Santa Monica. Dessa matematiker uttryckte problemet med hjälp av linjärprogrammering och utvecklade cutting-plane metoden för dess lösning. Med dessa metoder löste de ett problem med 49 städer optimalt genom att bevisa att inga andra vägar kan vara kortare. På 70-talet lyckades man lösa problem med upp till 2392 städer med cutting-plane metoden och branch-and-bound metoder.

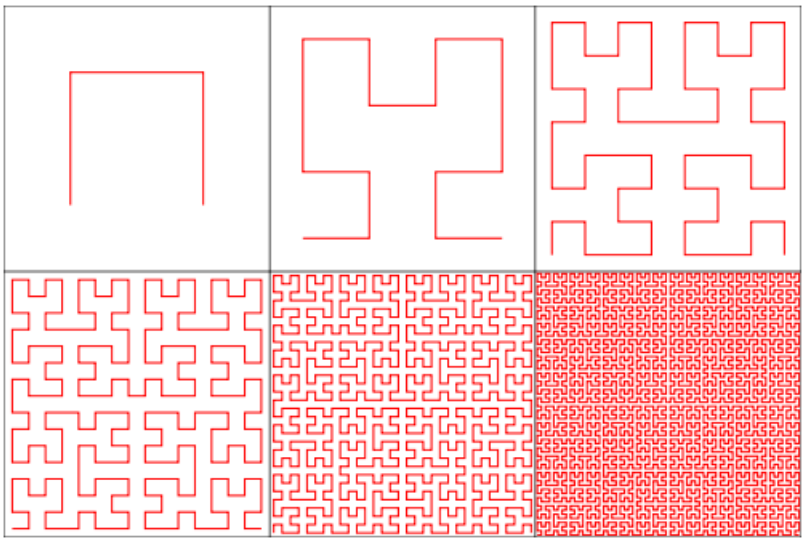
På 90-talet publicerade Gerhard Reinelt TSPLIB, en samling problem med varierande svårighetsgrad som har lösts optimalt. Många forskargrupper har sedan dess använt TSPLIB som ett sorts prestandatest gentemot sina egna lösningar. Cook m.fl. [2] visade 2005 en optimal rutt för ett problem med 33 810 städer, numera det största optimalt lösta TSPLIB-problemet. Skall man lösa TSP för miljontals städer måste man använda metoder som ger icke-optimala lösningar. Vår studie utforskar en av dessa metoder.



Figur 1: Olika algoritmer för att lösa TSP och deras tidskomplexitet. En seriestrip från xkcd.com [3]

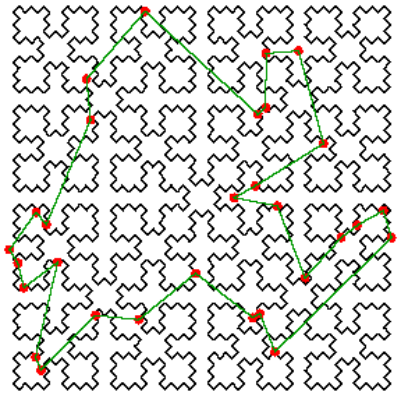
1.2 Vårt fokus

Kurvtypen som används i denna studie kallas för planfyllande kurvor (för exempel se fig. 2). Enkelt sett beskriver planfyllande kurvor ett sätt att besöka varje koordinat i ett område i en viss ordning.



Figur 2: Hilbertkurvan [4] i sex generationer. Kurvan använder fyra rektangulära delar i sitt itereringsmönster och en tydlig fraktalstruktur kan noteras. Detta är en av kurvtyperna vi använder för att lösa TSP.

Ordningen ges av ett förutbestämt konstruktionssätt och detta kan användas för att ordna ett antal punkter i planet på. Detta sätt att få fram en lösning för TSP benämns hädanefter standardsättet och visualiseras i fig. 3. Idag finns ingen känd algoritm som löser ett godtyckligt handelsresandeproblem optimalt på polynomiell tid (se sektion 3.2) vilket leder till att man söker goda approximationer.



Figur 3: Sierpinskikurvans genomlöpning av ett antal punkter i planet. Bilden är gjord av J.J Bartholdi [5]. Denna bild illustrerar hur en planfyllande kurva kan genomlöpa planet i en viss ordning och därmed även ett antal punkter.

Planfyllande kurvor är en oerhört snabb lösningsmetod jämfört med andra approximativa algoritmer för TSP och kan ge acceptabla lösningar i vissa tillämpningar. Vi blev inspirerade att optimera denna lösningsmetod för att komma närmare en optimal rutt än standardsättet beskrivet i Bartholdis ursprungliga artikel [6].

Lösningensmetoden med rumsfyllande kurvor har förmodligen sitt främsta användningsområde i att lösa TSP för system med flera miljoner punkter. Metoden hittar en approximativ lösning på ett fåtal sekunder.

2 Syfte

Studien har först och främst syftat till att ta fram kortare TSP-vägar än standardsättet med planfyllande kurvor. När detta uppnåddes låg fokus på att hålla algoritmerna snabba och effektiva, och komma ner till en relativt låg beräkningskomplexitet (teorin bakom beräkningskomplexitet finns beskrivet i sektion 3.1).

Följande frågeställningar är centrala i rapporten.

- Vilken påverkan har konstruktionssättet av våra rumsfyllande kurvor?

  Det kan visa sig att olika konstruktionssätt för kurvorna inte påverkar resultatet nämnvärt, eller tvärtom påverkar oerhört mycket.

- Hur stabila är lösningarna med avseende på störningar i punktplacering?

  Det är viktigt att veta hur känsliga ens algoritmer är för störningar för att kunna dra slutsatser om hur robusta algoritmerna är.

- Hur påverkar fördelningen av punkterna resultatet?

  I praktiken kan punkter t.ex. vara mycket mer klustrade - något som påverkar optimalitetsmåttet till det sämre.

3 Teori

Detta kapitel innehåller den bakomliggande teorin som är nödvändig för att förstå det övriga materialet i rapporten.

3.1 Beräkningskomplexitet

När man undersöker hur snabba olika algoritmer är jämfört med varandra använder man något som kallas komplexitetsteori. I vårt arbete studerar vi tidskomplexitet som beskriver hur lång tid en algoritm tar på sig att slutföra som en funktion av problemstorleken. Detta beskrivs vanligast med ordonotation. Om en algoritm exempelvis kräver $3n^2 + 5$ beräkningssteg för att slutföra en beräkning på ett problem av storlek n, så är dess asymptotiska tidskomplexitet $\mathcal{O}(n^2)$, då problemet växer som n^2 för stora n.

Polynomet ovan är ett exempel på en algoritm som har polynomiell tidskomplexitet. En algoritm som har $\mathcal{O}(2^n)$ tidskomplexitet kallas för exponentiell. Man kan då se att för en liten ändring i problemstorleken ökar en exponentiell algoritm mycket kraftigare i beräkningstid än en polynomiell hade gjort.

Handelseresandeproblemet kan lösas optimalt genom att undersöka alla kombinationer av vägar som är möjliga. Problemet är bara att körningstiden för en sådan lösning är $\mathcal{O}(n!)$ vilket redan för tiotals städer ger väldigt lång körningstid. Standardsättet att lösa TSP på med planfyllande kurvor kräver dock endast $\mathcal{O}(n \log n)$ vilket är oerhört mycket snabbare. En härledning av våra beräkningskomplexiteter sker i analys och slutsatser (se kapitel 6).

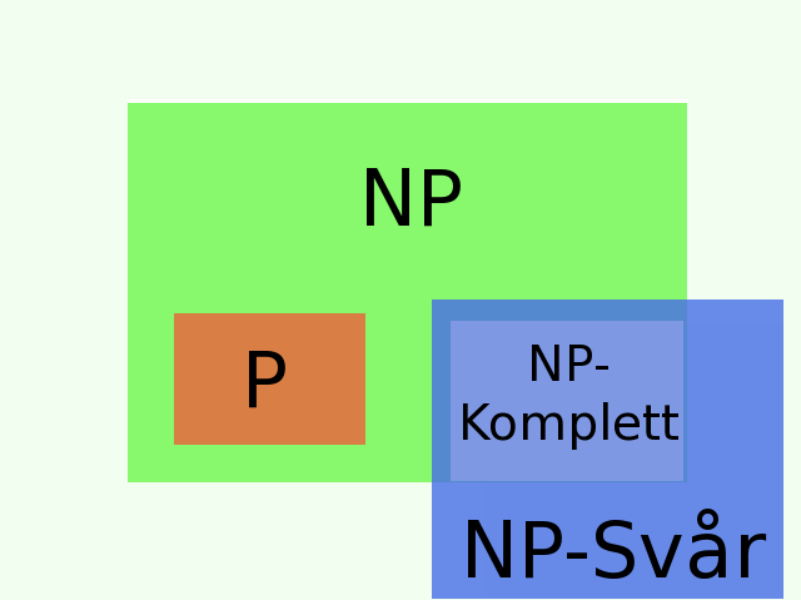
3.2 NP-problematiken och approximativa lösningar

Man brukar dela upp problem i olika klasser efter hur svåra de är att lösa och verifiera. Denna uppdelning är nödvändig för att förstå varför olika problem är olika svåra att lösa. Problemklassuppdelningen kan även ge information om hur svårighetsgraden ändras med problemstorleken.

Vissa klasser av problem är ofta enkla att lösa även för stora system. Andra problemklasser kan ha en kraftig ökning av beräkningskapacitetsbehov även för små ändringar i problemstorleken.

3.2.1 Klassen NP - Den största kategorin problem

Den största klassen problem brukar benämnas NP-problem. Klassen NP har sedan ett antal underkategorier som P, NP-kompletta och NP-svåra problem. En schematisk bild återfinns i figur 4.



Figur 4: En schematisk bild över de olika komplexitetsklasserna med antagandet $P \neq NP$. Problemklassen NP-svår ligger delvis i NP och delvis i andra komplexitetsklasser.

NP är i sig en väldigt diffust definierad problemklass och innefattar en mycket stor grupp problem. Ofta är det först när man specificerar problemets underkategori som man får någon relevant information om problemets egenskaper.

NP står för ”non-deterministic polynomial time” och problem i denna klass kan lösas med icke-deterministiska metoder inom polynomiell tid. Dessa problem har lösningar som karakteriseras av att de kan verifieras inom polynomiell tid även med helt deterministiska algoritmer.

Skillnaden mellan en deterministisk och en icke-deterministisk algoritm är att en icke-deterministisk algoritm kan ge olika resultat vid varje körning trots identiska förutsättningar. En helt deterministisk metod kommer dock alltid att ge exakt samma resultat efter varje körning givet att ingångsdatan är oförändrad.

3.2.2 Klassen P - Problem som ofta är lösbara i praktiken

Problem i P-klassen är alltid lösbara inom polynomiell tid med en deterministisk algoritm.

Att problem i P-klassen kan lösas inom polynomiell tid innebär vissa begränsningar på hur mycket svårare problem blir att lösa med ökad problemstorlek. Även om man inte satt någon övre gräns för hur stort polynomet som begränsar tidsåtgången får vara, slipper man ofta explosionsartade ökningar i beräkningstiden för små förändringar i problemstorleken.

Den viktigaste egenskapen hos klassen P är att den innehåller problem som oftast är praktiskt lösbara även för stora system.

3.2.3 Klassen NP-komplett - De svåraste problemen i NP

NP-kompletta problem är en speciell underkategori till NP-problem. Dessa problem karakteriseras av att det inte finns någon effektiv algoritm för att lösa dem snabbt. Typiskt för NP-kompletta problem är att svårigheten att lösa ett problem ökar enormt även för små ökningar av problemstorleken.

De kan även ses som de svåraste problemen i klassen NP. Handelseresandeproblemet har vissa frågeställningar som är NP-kompletta. Ett exempel på detta är: ”givet en rutt A och en uppsättning punkter, finns det en rutt som är bättre?”.

3.2.4 Klassen NP-svår - De svåraste problemen i NP eller svårare

En annan problemklass är NP-svåra problem. Dessa problem är åtminstone lika svåra som de svåraste NP-problemen men möjligheten finns att de är svårare än så. Detta betyder att det inte med säkerhet går att hitta en lösning till problemet inom polynomiell tid även med icke-deterministiska algoritmer. Handelseresandeproblemet i helhet brukar klassificeras som ett NP-svårt problem. Beräkningstidsskillnaden mellan att lösa TSP optimalt i ett system med 10 och 15 punkter är ca 72 gånger med en väldigt effektiv algoritm [7].

Då NP-svåra problem kan vara svårlösliga även för mycket små problem brukar man ofta ta till approximativa lösningar. Algoritmerna för att hitta dessa approximativa lösningar ligger oftast i klassen P och den approximativa lösningen kan ses som att man reducerar problemklassen från NP-svårt eller NP-komplett till P. Priset för problemklassreduceringen är att lösningen med största sannolikhet inte blir optimal.

3.2.5 Komplexitetsproblemet $P = NP$?

"$P = NP$?" är ett olöst komplexitetsproblem. Frågeställningen är om man med rätt algoritm skulle kunna reducera alla problem i klassen NP till P utan att göra avkall på lösningskvalitén. Om detta skulle visa sig vara sant skulle det innebära att det skulle vara teoretiskt möjligt att lösa alla problem i NP-klassen inom polynomiell tid. Detta skulle förhoppningsvis kunna leda till utvecklandet av smartare algoritmer för många problem.

Om antagandet skulle stämma skulle det kunna innebära att man kan lösa TSP optimalt på polynomiell tid. Detta skulle på sikt kunna innebära att vi inte behöver använda approximativa metoder ens för stora system. Det skulle även innebära att P-delen i figur 4 skulle växa och helt täcka NP-delen i figuren och enbart lämna ett litet fragment av den NP-svåra mängden.

Man ska dock ha i åtanke att bara för att ett problem kan lösas inom polynomiell tid behöver det inte betyda att just den algoritmen är praktiskt användbar. Klassen P sätter ingen begränsning på polynomet och tillhörande beräkningstid skulle mycket väl kunna växa alltför snabbt för att vara av praktisk nytta.

3.3 Planfyllande kurvor

Planfyllande kurvor är en speciell familj av kurvor som karakteriseras av att de trots sin endimensionella natur besöker varje punkt i ett område i $\mathbb{R}^2$. Anledningen till att detta är möjligt är att kurvorna har en speciell sorts fraktalstruktur [8].

Det faktum att kurvorna dels har en planfyllande natur och dels en kurvnatur medför att kurvan kommer att besöka varje punkt i ett område i en bestämd ordning. Det är denna egenskap som man utnyttjar när man använder planfyllande kurvor för att lösa handelseresandeproblemet. Det har visat sig sig att vissa speciella planfyllande kurvor genomlöper en uppsättning punkter i ett område på ett sådant sätt att det blir en approximativ lösning till handelsresandeproblemet.

3.3.1 Matematisk beskrivning av planfyllande kurvor

Planfyllande kurvor har en strikt matematisk definition. Förutsättningarna för definitionen är:

- Låt $\Omega \in \mathbb{R}^2$ med euklidisk norm och $I = [0, 1] \in \mathbb{R}$.
- Låt $f_*(A)$[1] vara avbildningen av A med f.
- Låt $J_2(A)$ vara Jordanmåttet i $\mathbb{R}^2$ av A.

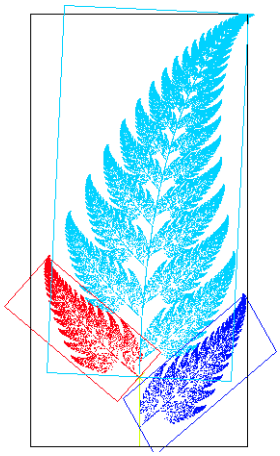
Med förutsättningarna ovan lyder den matematiska definitionen:

Om f är en kontinuerlig avbildning sådan att: $f : I \mapsto \Omega$ och $J_2(f_*(I)) > 0$ så är $f_*(I)$ en rumsfyllande kurva.

3.3.2 Design av planfyllande kurvor

Planfyllande kurvor kommer i oändligt många varianter, alla med olika egenskaper och utseende. Den viktigaste egenskapen hos en planfyllande kurva är att den besöker varje punkt i ett område $\Omega \in \Re^2$ minst en gång. Detta är ekvivalent med att kurvan har ett positivt Jordanmått.

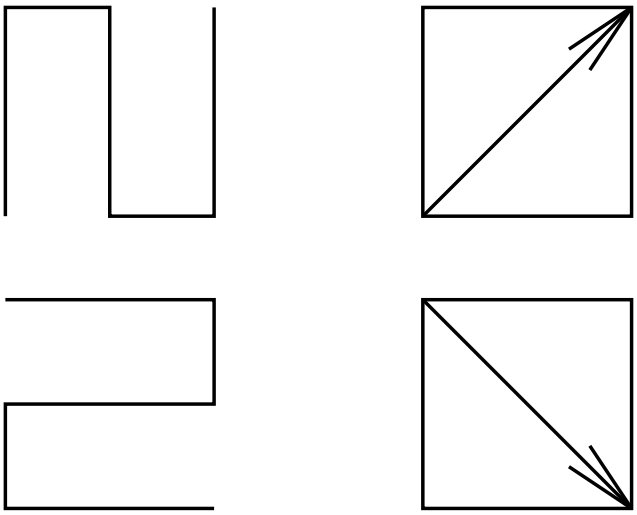
För att bygga kurvor som uppfyller kriteriet med att besöka varje punkt minst en gång används vanligtvis iterationsmönster. Det är iterationsmönstret som bygger upp kurvorna och ger dem dess fraktalstruktur. Det är vanligt, men ingen nödvändighet, att använda självliknande kurvor. Självliknande kurvor har samma struktur oavsett vilken ”inzoomning” man använder sig av. Detta illustreras väl i figur 5.



Figur 5: Figuren föreställer Barnsleys ormbunke som är en självliknande struktur. Vi kan se att figuren enbart består av ett enda segment som skalas och roteras. Detta enda segment bygger sedan upp hela figuren. Våra planfyllande kurvor uppvisar liknande beteenden.

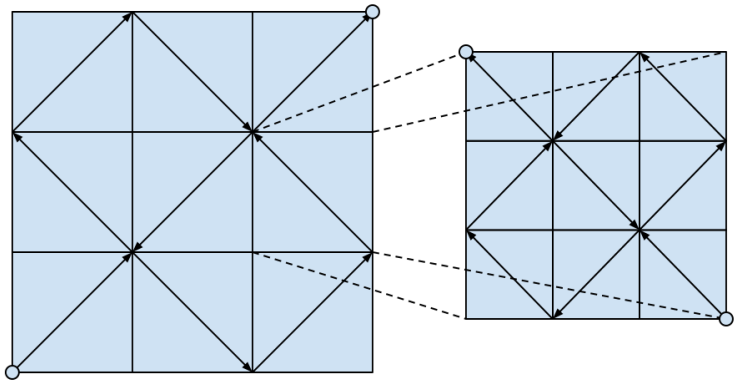
Vanligtvis brukar man använda ett grundläggande kurvsegment som byggsten för de planfyllande kurvorna. Detta kurvsegment kan sedan roteras/spegelvändas för att skapa olika varianter som sedan pusslas ihop. Dessa kurvsegment kan ses som de legobitar som bygger upp hela den planfyllande kurvan. Schematiskt beskrivs dessa kurvsegment med deras utgångs- och ingångspunkter samt deras utbredning. Detta brukar åskådliggöras med en ruta och en pil där pilens bakre del är inledningspunkten och dess spets är avslutningspunkten. Detta åskådliggörs i figur 6.

[1] Här är f den funktionen som definerar avbildningen och $f_*(A)$ den konkreta avbildningen av A med f.



Figur 6: Två faktiska kurvsegment (vänster) och deras schematiska beskrivning (höger). Den schematiska beskrivningen ger information om segmentets ingång- och utgångspunkt samt dess utbredning.

Dessa kurvsegment byggs sedan ihop efter något specifikt iterationsmönster. Processen upprepas sedan för varje delområde och på så vis får man en självliknande fraktalstruktur. Detta förklaras bäst i figur 7.



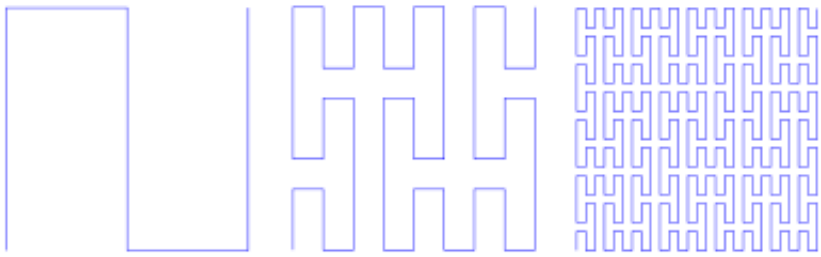
Figur 7: Uppbyggnad av en hel planfyllande kurva genom kombinering och skalning av kurvsegment. Just detta schema skulle kunna vara en del av Peanokurvan. I figuren svarar de små cirklarna mot segmentets ingångs- och utgångspunkt.

Genom att använda scheman som det i figur 7 kan man bygga upp kurvor som efter oändligt många iterationer når precis alla punkter i området minst en gång. På grund av datorernas diskreta och ändliga natur är det dock inte praktiskt möjligt att generera en oändligt fin planfyllande kurva. Med andra ord kan en datorgenererad planfyllande kurva aldrig bli planfyllande i dess sanna bemärkelse. Detta kommer dock inte att medföra någon begränsning i

vår praktiska tillämpning vilket vi visar i kapitel 4.1.

3.3.3 Peanokurvan

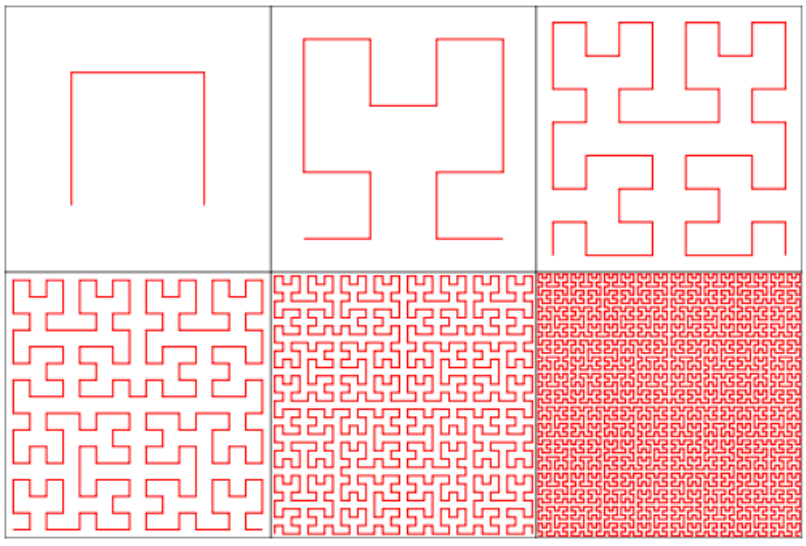
Peanokurvan var den första kvadratiska planfyllande kurvan och upptäcktes 1890 av Giuseppe Peano. Peanokurvan använder 9 stycken rektangulära segment i sitt iterationsmönseter och en schematisk bild över Peanokurvans uppbyggnad återfinns i figur 7. Peanokurvan har inte visat sig särskilt effektiv för att hitta bra lösningar till generella TSP, något vi även bekräftade i resultatdelen.



Figur 8: Peanokurvan i tre generationer. Peanokurvans komplexitet växer snabbt med antalet generationer då den bygger på en niofaldig symmetri.

3.3.4 Hilbertkurvan

Hilbertkurvan är en klassisk rumsfyllande kurva som använder fyra rektangulära segment i sitt iterationsmönster. Hilbertkurvan har visat sig vara effektivare än Peanokurvan för att lösa praktiska TSP-problem. Effektiviteten i kombination med den behändiga symmetrin har gjort Hilbertkurvan populär på senare tid. En grafisk representation av kurvan återfinns i figur 9.

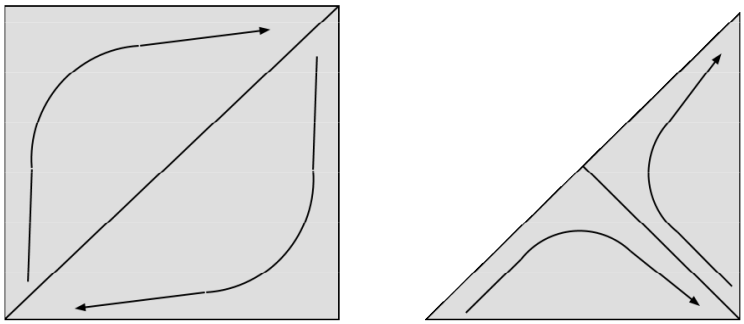


Figur 9: Hilbertkurvan i sex generationer. Hilbertkurvan är en relativt lättarbetad planfyllande kurva då den enbart bygger på räta linjer.

En annan fördel med Hilbertkurvan är att det är enkelt och smidigt att förändra iterationsmönstret utan att störa kurvans planfyllande egenskaper. Det är denna egenskap vi utnyttjat i vår Smartsplitalgoritm för att förbättra kurvans effektivitet i lösningen av TSP.

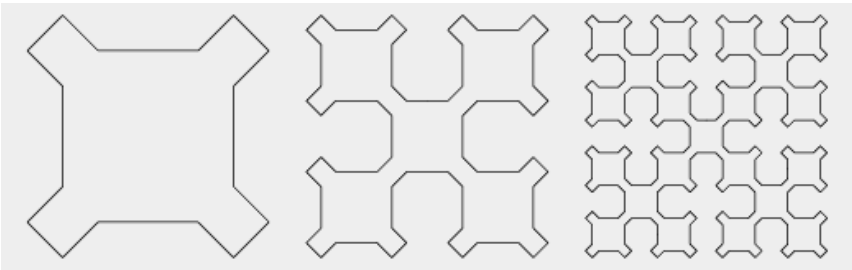
3.3.5 Sierpińskikurvan

Sierpińskikurvan har två triangulära kurvsegment vilket skiljer sig från både Hilbert- och Peanokurvan. En grafisk representation av iterationsmönstret ges i figur 10.



Figur 10: Iterationsmönstret för en Sierpińskikurva i två generationer. Vi kan se att Sierpińskikurvan använder en triangulär symmetri.

Den triangulära symmetrin gör att Sierpińskikurvan är besvärlig att arbeta med och modifiera i praktiska tillämpningar. Trots problemen med triangelsymmetrin används Sierpińskikurvan ofta i verkligheten då den har visat sig vara mycket snabb och effektiv. Sierpiniskikurvan åskådliggörs i figur 11.



Figur 11: Sierpińskikurvan i tre generationer. Triangelsymmentrin yttrar sig i de 45-gradiga vinklarna som uppkommer i kurvstrukturen.

3.3.6 Modifiering av iterationsmönster för planfyllande kurvor

För att skapa en planfyllande kurva krävs ett iterationsmönster. Vanligtvis är dessa iterationsmönster högst specifika och icke-dynamiska. Till exempel bygger Hilbertkurvan på att man delar in varje segment i fyra nya lika stora segment genom hela processen.

Detta iterationsmönstret går dock att modifiera så länge man behåller de planfyllande kurvornas grundläggande egenskaper; kurvan får aldrig brytas och alla punkter i området måste besökas minst en gång.

Man skulle kunna tänka sig att dela upp varje segment i Hilbertkurvan i fyra segment med olika storlek så länge ovanstående villkor är uppfyllda. Dessa segment behöver inte heller vara av konstant storlek utan storleken skulle till exempel kunna baseras på andra faktorer. Exempel på detta kan vara koncentrationen av städer i just ett delområde. Detta är grunden till vår Smartsplitalgoritm.

Liknande modifikation går självklart att göra för andra kurvtyper. Till exempel kan man tänka sig att skapa trianglar med olika storlek i Sierpińskikurvan.

3.4 Teoretisk approximation för optimal lösningslängd

TSP är ett erkänt svårt problem att lösa exakt och således är det ofta svårt att veta den optimala lösningslängden. Dock kan det ibland vara bra att kunna göra en teoretisk approximation, till exempel då man vill uppskatta kvaliteten hos en approximativ lösning.

Detta problem undersöktes av Beardwood, Halton och Hammersley och deras slutsatser presenterades i artikeln "The shortest path through many points" år 1959 [9]. De undersökte uniformt fördelade punktmängder och kom fram till att den kortaste sträckan mellan punkterna var:

$$l_{\text{uniform}} \approx 0.53\sqrt{2NA} \tag{1}$$

I denna ekvation står N för antalet punkter i problemet och A är arean för det minsta rätblocket som innesluter alla punkterna.

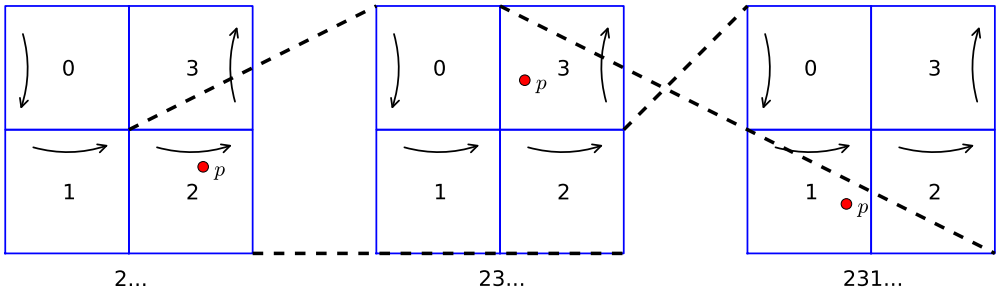
4 Genomförande

4.1 Omodifierade planfyllande kurvor

För att lösa TSP med planfyllande kurvor på det klassiska sättet, börjar vi med att transformera alla koordinater till kvadraten $[0,1] \times [0,1]$. Varje punkt tilldelas sedan ett kurvnummer, som är en approximering av hur långt in på den planfyllande kurvan punkten hamnar. Punkterna sorteras sedan enligt sina kurvnummer, och resultatet blir en lista med ordningen punkterna besöks av den planfyllande kurvan.

Kurvnumret för en punkt beräknas genom en iterativ process som motsvarar den rumsfyllande kurvans självliknande egenskap. Vi beskriver enbart processen för Hilbertkurvan. Övriga kurvtyper hanteras på motsvarande sätt.

Eftersom Hilbertkurvan bygger på rekursiv uppdelning i fyra delar, är det naturligt att betrakta kurvnumret i bas 4. Den mest signifikanta siffran sätts till ett värde mellan 0 och 3, beroende på vilken av fyra underrutor som punkten ligger i. Punkten transformeras sedan tillbaka från underrutan till $[0,1] \times [0,1]$, där transformationen som används beror på vilken underruta punkten låg i. Kurvnumrets näst mest signifikanta siffra bestäms på samma sätt, och processen upprepas till en önskad precision, som bestäms av antalet iterationsnivåer k. Figur 12 beskriver processen i tre nivåer.



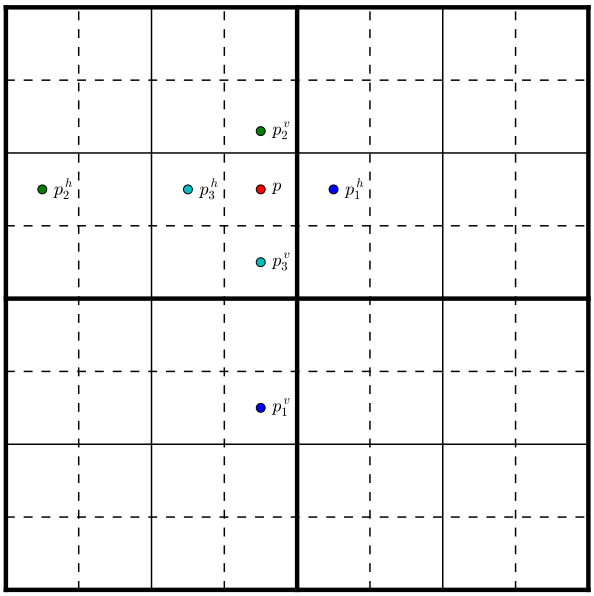
Figur 12: Iterativ tilldelning av kurvnummer för en punkt p. Notera att andra eventuella punkter i rutorna 0 och 1 i den första bilden kommer tilldelas lägre kurvnummer än p.

Då fler än en punkt tilldelas samma kurvnummer besöks de i odefinierad ordning, vilket leder till betydligt längre väg. Precisionen på kurvnumrena måste därför väljas så att detta sällan sker. För ett problem med N uniformt fördelade punkter innebär det att tidskomplexiteten för att tilldela ett kurvnummer till en punkt är $\mathcal{O}(\log N)$. I praktiken innebär dock detta inga problem, då precisionen kan väljas tillräckligt hög för att lösa alla tänkbara problem utan nämnvärd inverkan på körningstiden.

I stället domineras körningstiden av sorteringssteget, som har tidskomplexiteten $\mathcal{O}(N \log N)$.

4.2 Punktbyte

För att förbättra lösningarnas längd har vi implementerat ett efterbehandlingssteg med inspiration från en kortfattad beskrivning av Bartholdi och Platzman [6].

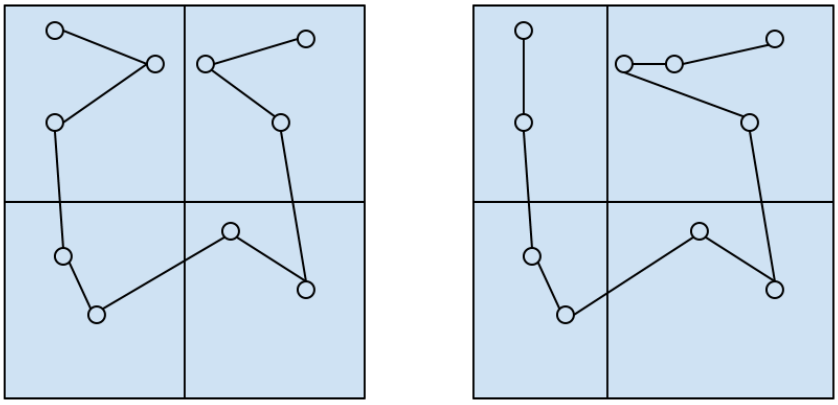


Figur 13: Punkten p speglas vertikalt (v) och horisontellt (h) för tre nivåer.

Indatan till efterbehandlingssteget är en sorterad lista av punkter med koordinater och kurvnummer. För varje punkt och iterationsnivå beräknas ett antal olika alternativa kurvnummer som speglingar in i de bredvidliggande underrutorna på nivån, detta illustreras i figur 13. Ett sådant nummer kan beräknas på konstant tid för en given precision genom att manipulera det tilldelade kurvnumret. Punktens alternativa platser på vägen söks upp i den sorterade listan av punkter med binärsökning, och för varje sådan plats beräknas skillnaden mellan den totala väglängden, och den resulterade väglängden om punkten skulle flyttas. Kurvnumret som ger den kortaste vägen sparas undan, och processen upprepas för alla punkter i problemet. Punktlistan sorteras sedan enligt de nya kurvnumrena.

4.3 Smartsplit

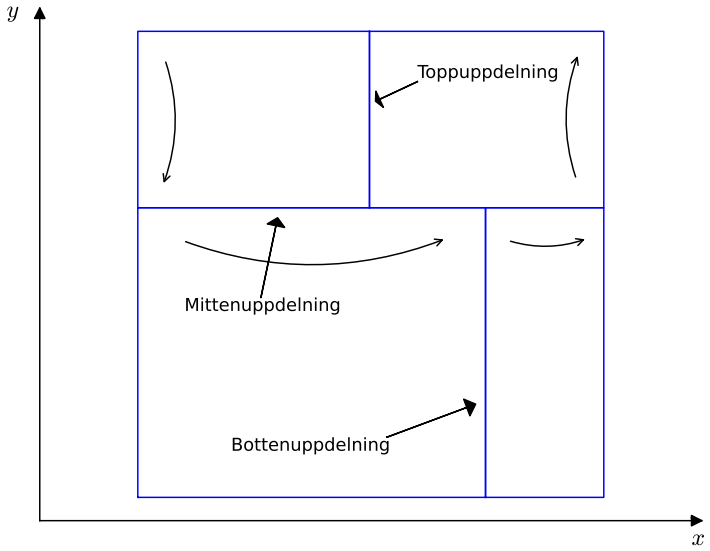
Efter att ha studerat ett antal TSP-lösningar med omodifierade planfyllande kurvor insåg vi att en stor problematik i lösningarna var ofördelaktiga subintervallindelningar i itereringsmönstret. Grundidén bakom algoritmen visas i figur 14. För klustrade problem märks detta tydligt, och den genererade lösningen hoppar ofta fram och tillbaka mellan kluster onödigt många gånger.



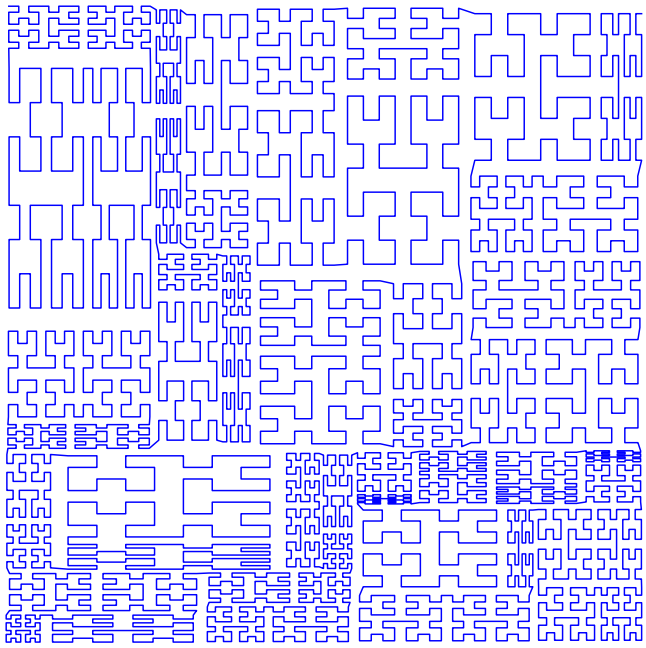
Figur 14: Grundidén bakom Smartsplit. I den vänstra figuren visas en (approximativ) genomlöpsordning genererad med en omodifierad Hilbertkurva. I den högra figuren har vi skevat iterationsmönstret en aning för att undvika problematiken med de två närbelägna punkterna som hamnat på var sin sida om den horisontella uppdelningslinjen.

Av de tre kurvtyperna vi studerat har Sierpinskikurvan i medel gett de bästa lösningarna på handelsresandeproblemet, se tabell 2 och 3. Trots detta har vi valt att studera modifikationen på Hilbertkurvan eftersom alla intervalluppdelningar sker längs koordinataxlarna, vilket förenklar beräkningarna betydligt.

Eftersom den resulterande kurvan ska behålla sin kontinuitet krävs att indelningen sker på ett sådant sätt att start- och slutpunkterna för de indelade rutorna hamnar bredvid varandra. För Hilbertkurvan innebär detta att tre olika förskjutningar kan göras vid varje iterationssteg, i mittenuppdelningen, bottenuppdelningen och toppuppdelningen. Detta illustreras i figur 15.



Figur 15: De tre olika förskjutningarna av intervalluppdelningen i Smartsplit. Pilarna markerar var kurvan går in och ut i de olika delkurvorna.



Figur 16: Illustration av en Hilbertkurva med kraftigt förskjutna intervalluppdelningar.

För att välja en lämplig uppdelningslinje går vi igenom en lista av punkterna i området. Punkterna är sorterade efter x- eller y-koordinaten beroende på vilken ledd uppdelning sker på. Då vi exempelvis gör mittenuppdelningen i figur 15 ska listan vara sorterad efter y-koordinaterna.

Mellan varje par av bredvidliggande punkter i listan, samt innan den första och efter den sista, räknas ett tal L ut. Talet är tänkt som ett relativt mått på hur lång en lösning förväntas vara givet en uppdelning på det stället. Beräkningen sker med den experimentellt framtagna formeln

$$L = c\sqrt{N} - KdN \tag{2}$$

Här är c avståndet från uppdelningslinjen till rutans centrum. N är antalet punkter. d är avståndet mellan punkterna (eller mellan punkt och kant) längs sorteringsaxeln. Uppdelningslinjens position väljs så nära centrum som är möjligt i det tillgängliga utrymmet.

K är en konstant som vi bestämt genom att optimera lösningslängden över ett antal problem av varierande storlek och karaktär. Det visade sig vara fördelaktigt att använda olika värden på K för de olika typerna av uppdelning. Tabell 1 visar de värden vi använt.

Tabell 1: Framräknade värden på konstanten K i ekvation (2)

Uppdelning	K
Mitten	0.008088
Botten	0.143059
Topp	0.108811

Intuitivt kan parametern K förstås som ett mått på hur mycket avståndet mellan de två punkterna där uppdelningen sker ska spela in i bedömningen. $K = 0$ innebär att uppdelningen alltid hamnar i mitten, alltså samma beteende som för den omodifierade Hilbertkurvan.

För att komma fram till ekvation (2) har vi undersökt diverse olika ansatser experimentellt. Vi har bland annat provat att väga in fler punkter och att hantera områdets höjd, bredd och antal punkter på olika sätt. Ekvation (2) valdes då den är relativt enkel att förstå, och ger bra resultat för de problem vi studerat.

4.3.1 Implementation

Genom att underhålla två olika listor av problemets punkter, sorterade i x- respektive y-led, kan en mängd av N olika punkter delas upp på $\mathcal{O}(N)$ tid. Detta gör vi på följande sätt:

Alla punkter i listorna innehåller index till sig själva i den andra listan. Då en uppdelning ska ske mellan två punkter i till exempel den x-sorterade listan, används dessa index för att markera punkterna i den y-sorterade listan som hamnar efter uppdelningspunkten. Genom att använda en temporär lista flyttas sedan de omarkerade punkterna till början av den y-sorterade listan och de markerade till slutet, under bevarandet av deras inbördes ordning. Slutligen uppdateras x-listans index. Allt detta sker på linjär tid.

Resultatet blir att listorna kan delas upp vid uppdelningspunkten, och processen kan upprepas på de båda delarna. När endast en punkt återstår i en punktmängd, läggs den till en lista och rekursionen slutar. För att kunna kombinera Smartsplit med punktbyte tilldelas punkten ett kurvnummer, men genom att besöka punkterna i samma ordning som den planfyllande kurvan kan sorteringssteget i slutet undvikas.

4.3.2 Tidskomplexitet

Vi kan dela in algoritmen i två olika faser. I förberedelsefasen sorterar vi punktlistan på två olika sätt, och får tidskomplexiteten $\mathcal{O}(N \log N)$. Hänvisande index mellan de båda listorna kan skapas på linjär tid, vilken ger en total komplexitet på $\mathcal{O}(N \log N)$.

För att beräkna tidskomplexiteten i den andra fasen studerar vi varje uppdelningsnivå för sig. På den högsta nivån ska N punkter delas upp. Som beskrives ovan, kan man både hitta en uppdelningspunkt och utföra uppdelningen på linjär tid. På nästa nivå ska två punktmängder delas upp, och vi noterar att summan av punkterna i de båda delarna är högst N, vilket också gäller för alla nästkommande nivåer. Antalet nivåer är N i värsta fall, vilket ger en total tidskomplexitet på $\mathcal{O}\left(N^2\right)$.

I de problem vi studerat växer dock antalet nivåer snarare proportionellt mot $\log N$. För att garantera detta kan man låta algoritmen falla tillbaka till en omodifierad Hilbertkurva efter ett antal uppdelningar.

4.3.3 Smartsplit och punktbyte

Punktbytessteget kan tillämpas även på Smartsplit, utan förändring. Eftersom uppdelningarna är förskjutna, kommer inte det modifierade kurvnumret motsvara en exakt spegling. Detta verkar inte ha någon avgörande betydelse, och metoden uppvisar bra lösningsresultat.

5 Resultat

I denna sektion utvärderar vi prestanda, stabilitet och beräkningstider av de TSP-lösare vi har utvecklat under arbetets gång. Då dessa lösare bygger på olika metoder har vi jämfört dem mot varandra. En närmare beskrivning om hur lösarna utvecklats och vilka algoritmer de bygger på återfinns i kapitel 4.

5.1 Prestanda

Prestandautvärderingarna syftar till att kvantifiera lösningskvaliteten hos de olika lösarna. Lösningskvaliteten baseras på lösningarnas totala längd.

5.1.1 Utförande av prestandamätning

Prestandauppskattningarna är baserade på empiriska tester av lösarna. För att jämföra lösarnas effektivitet på olika sorters problem genererades två uppsättningar problem, en uppsättning med uniformt fördelade punkter och en uppsättning med klustrade problem. Totalt genererades 300 problem av de båda typerna. Problemstorleken var jämn t fördelad mellan 10 000 och 1 000 000 punkter i bägge fallen.

Som mått på de uniforma lösningarnas kvalitet användes det approximativa optimalitetsvärdet som beskrivs i kapitel 3.4.

För de klustrade problemen finns inget simpelt approximativt optimalitetsmått. Detta medförde att vi enbart kunde jämföra lösarnas relativa prestanda för våra 300 genererade problem. För att få lätthanterliga mått normerades alla resultat med det bästa, vilket visade sig vara Sierpińskikurvan med punktbyte.

För att få ett absolut prestandamått för problem med klusterstruktur användes ett antal klustrade problem med kända lösningar.

5.1.2 Tolkning av tabellerna

För att få konkreta mått på lösarnas prestanda har vi studerat deras lösningslängder. Alla resultat utom de i kapitel 5.1.4 indikerar hur mycket lösningslängden skiljer sig från det faktiska eller approximativa optimumet. Till exempel skulle ett värde på 40% betyda att lösningen är 40% längre än den bästa lösningen.

I kapitel 5.1.4 svarar procentsatserna för lösarnas relativa prestanda. Som norm användes Sierpińskikurvan med punktbyte.

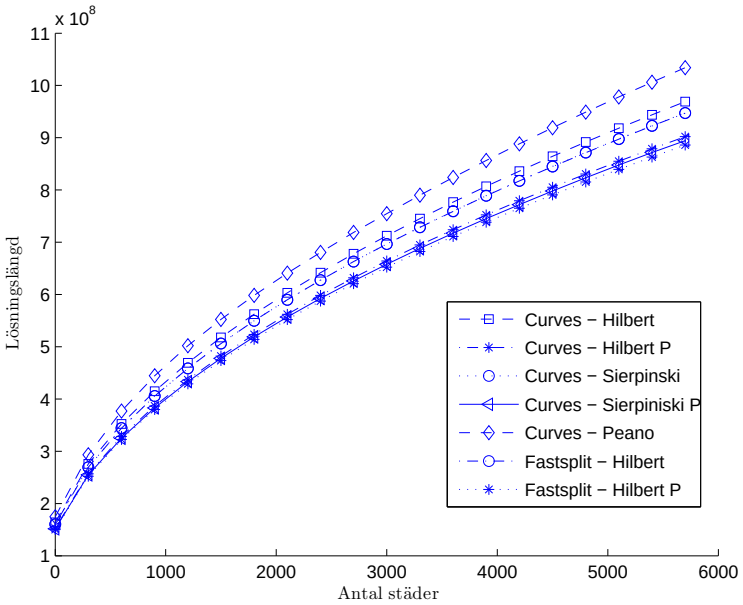
Fastsplit är vår implementation av Smartsplit. I våra tabeller har vi omnämt lösare som använder Fastsplit *kurvnamnF*, lösare som använder punktbyte *kurvnamnP* och lösare som använder både Fastsplit och punktbyte *kurvnamnFP*.

5.1.3 Absolut prestanda för uniformt fördelade problem

Tabellen visar medelvärdet av lösningskvaliteten för 300 genererade testproblem med uniform punktfördelning.

Tabell 2: Medelvärde av absolut lösningskvalitet för 300 genererade uniforma testproblem

	Hilbert	Peano	Sierpiński
Omodifierad	34.1%	42.7%	31.1%
Fastsplit	30.1%	-	-
Punktbyte	24.9%	-	23.8%
Punktbyte + Fastsplit	22.7%	-	-



Figur 17: En plot över hur den totala lösningslängden påverkas av problemstorleken med konstant area. Datan i plotten är baserad på 300 genererade testproblem. Vi kan se att lösningslängden växer ungefär som $\sqrt{N}$ vilket också är förväntat enligt ekvation (1). Vi kan även se att alla algoritmernas effektivitet verkar vara relativt opåverkade av problemstorleken.

5.1.4 Relativ prestanda för problem med klusterstruktur

Denna tabell visar medelvärdet av lösarnas relativa prestanda för 300 genererade testproblem med klusterstruktur. Som norm användes Sierpińskikurvan med punktbyte.

Tabell 3: Medelvärde av relativ lösningskvalitet för 300 genererade klustrade testproblem

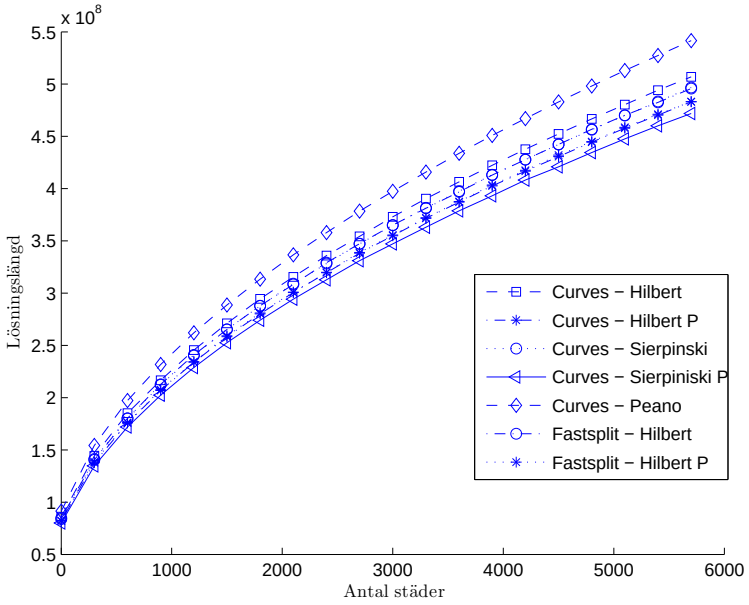
	Hilbert	Peano	Sierpiński
Omodifierad	107%	115%	105%
Fastsplit	105%	-	-
Punktbyte	102%	-	100%
Punktbyte + Fastsplit	102%	-	-

5.1.5 Absolut prestanda för problem med klusterstruktur

Denna tabell visar konkreta resultat från ett antal fördefinierade problem och lösningsmetoder. Se kapitel 5.1.1 för en mer detaljerad beskrivning.

Tabell 4: Lösningskvalitet för en uppsättning klustrade fördefinerade problem

Punkter	Hilbert	HilbertP	HilbertF	HilbertFP	Sierpiński	SierpińskiP	Peano
1 000	40.5%	35.5%	33.0%	28.9%	37.6%	32.9%	56.8%
1 000	42.3%	31.2%	41.4%	35.3%	41.4%	36.3%	47.2%
1 000	44.1%	37.0%	40.6%	30.1%	37.0%	27.0%	50.6%
1 000	42.4%	40.0%	43.8%	37.4%	37.4%	29.4%	52.2%
1 000	46.7%	39.9%	36.4%	32.5%	43.9%	34.8%	55.3%
1 000	47.6%	41.5%	41.9%	40.7%	42.5%	33.5%	66.6%
1 000	53.4%	45.1%	38.0%	28.5%	54.0%	46.8%	80.9%
1 000	42.5%	34.6%	42.9%	31.1%	47.4%	40.3%	53.9%
1 000	49.5%	41.6%	38.2%	38.8%	41.8%	36.9%	59.0%
1 000	43.7%	42.6%	39.1%	32.5%	38.7%	34.2%	53.8%
3 000	50.1%	42.5%	48.8%	40.7%	45.9%	40.3%	57.6%
3 000	51.7%	45.3%	45.2%	38.1%	44.9%	41.2%	55.9%
3 000	49.2%	44.6%	42.4%	43.6%	46.1%	46.4%	59.0%
3 000	50.2%	45.4%	43.6%	40.6%	42.8%	38.9%	60.7%
3 000	50.7%	43.5%	49.2%	41.3%	48.3%	38.7%	61.3%
10 000	49.9%	42.2%	46.9%	43.4%	44.8%	39.1%	62.6%
10 000	52.2%	46.1%	50.2%	47.1%	51.9%	43.1%	63.0%
10 000	53.4%	48.7%	47.3%	41.7%	49.8%	41.4%	62.3%
31 000	50.0%	46.8%	51.6%	48.1%	50.4%	43.0%	64.7%
31 000	55.2%	49.3%	51.3%	47.2%	50.4%	43.5%	64.2%
100 000	54.5%	47.6%	52.5%	47.8%	51.6%	44.3%	65.8%
100 000	54.9%	47.9%	52.0%	49.1%	52.0%	44.6%	64.2%
316 000	54.8%	47.8%	51.0%	47.5%	51.8%	44.8%	65.4%
Medel	49.3%	42.9%	44.7%	39.7%	45.8%	39.2%	60.1%
Varians	22.1	23.0	31.5	44.3	27.6	28.7	48.6



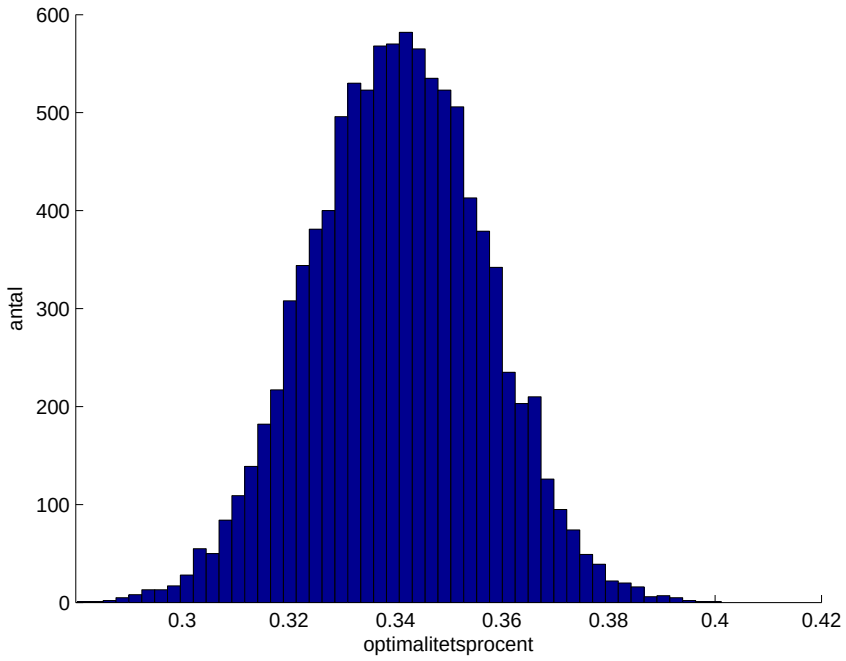
Figur 18: En plot över hur den totala lösningslängden påverkas av problemstorleken med konstant area. Datan i plotten är baserad på 300 genererade testproblem. Vi kan se att lösningslängden är kortare än i det uniforma fallet. Lösningslängden växer ungefär lika snabbt som de uniforma problemen med problemstorleken.

5.2 Stabilitet

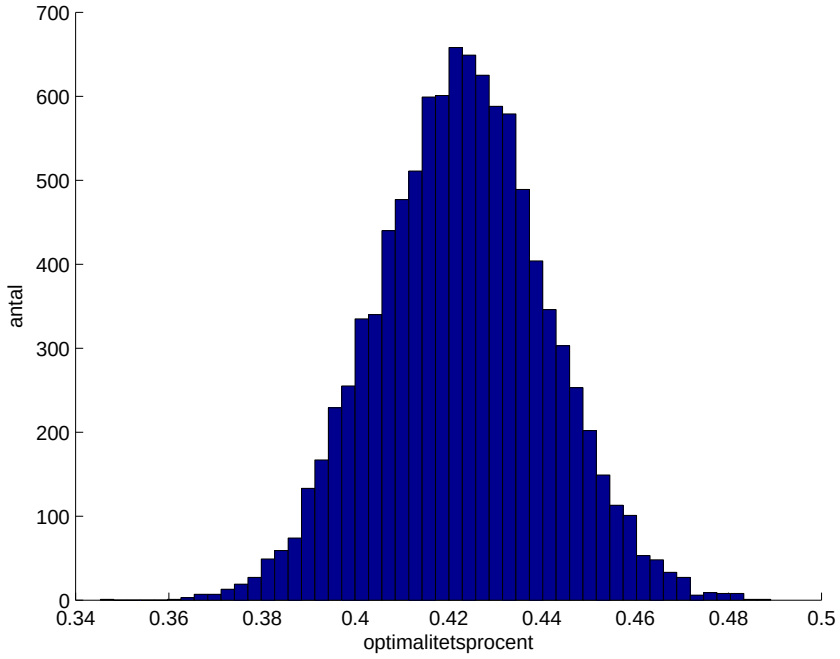
För att ta reda på hur stor varians det är i resultatet för våra olika kurvtyper för olika problem genererade vi 10 000 slumpmässiga problem med 1000 uniformt fördelade punkter vardera. Därefter har vi plottat histogram som visar fördelningen av lösningslängder för dessa 10 000 problem.

Vi testade hur bra dessa distributioner stämmer överrens med en normalfördelning med ett Lillieforstest. Det visade sig att alla tre rumsfyllandekurvor vi har använt har lösningar som tenderar att följa en normalfördelning. Detta kan vi säga med 95% sannolikhet.

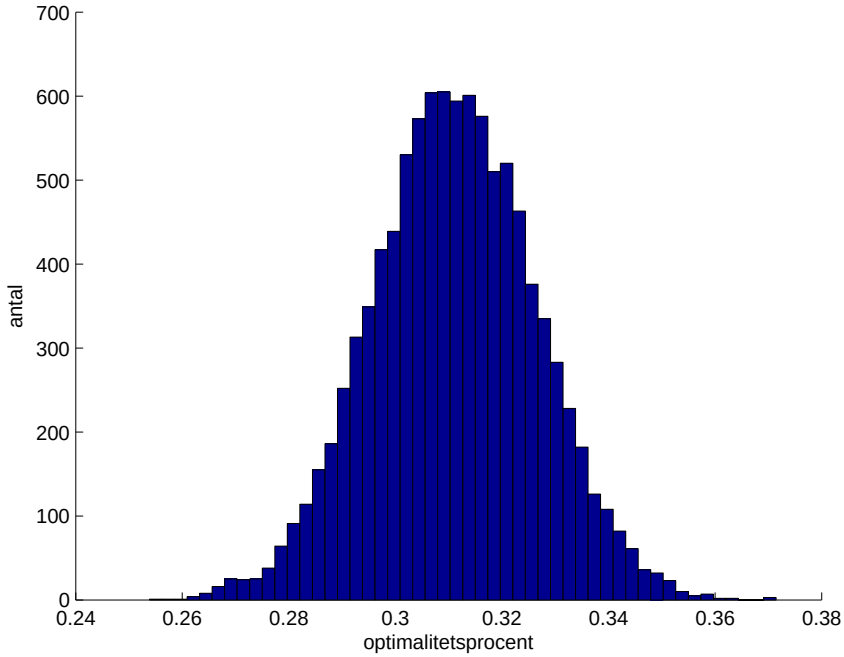
Histogram för dessa följer nedan.



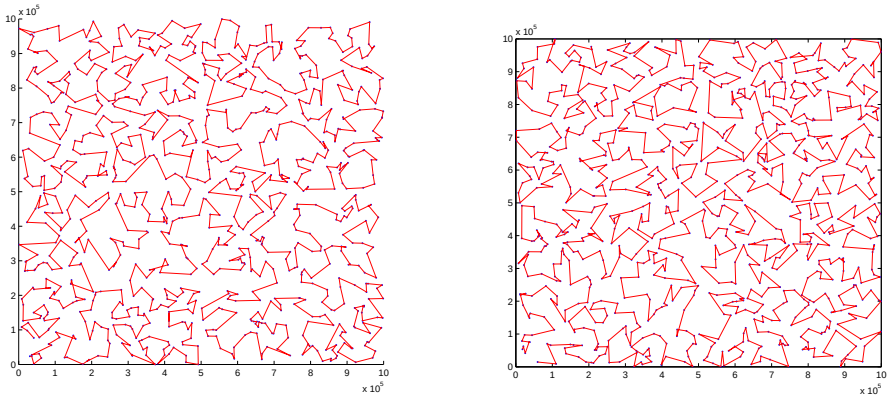
Figur 19: Histogram för hur lösningslängder varierar för Hilbertkurvan. För att få fram detta resultat har vi genererat 10 000 slumpmässiga problem med 1000 uniformt fördelade punkter vardera. En normalfördelning kan antydas och tester bekräftar detta.



Figur 20: Histogram för hur lösningslängder varierar för Peanokurvan. För att få fram detta resultat har vi genererat 10 000 slumpmässiga problem med 1000 uniformt fördelade punkter vardera. En normalfördelning kan antydas och tester bekräftar detta.

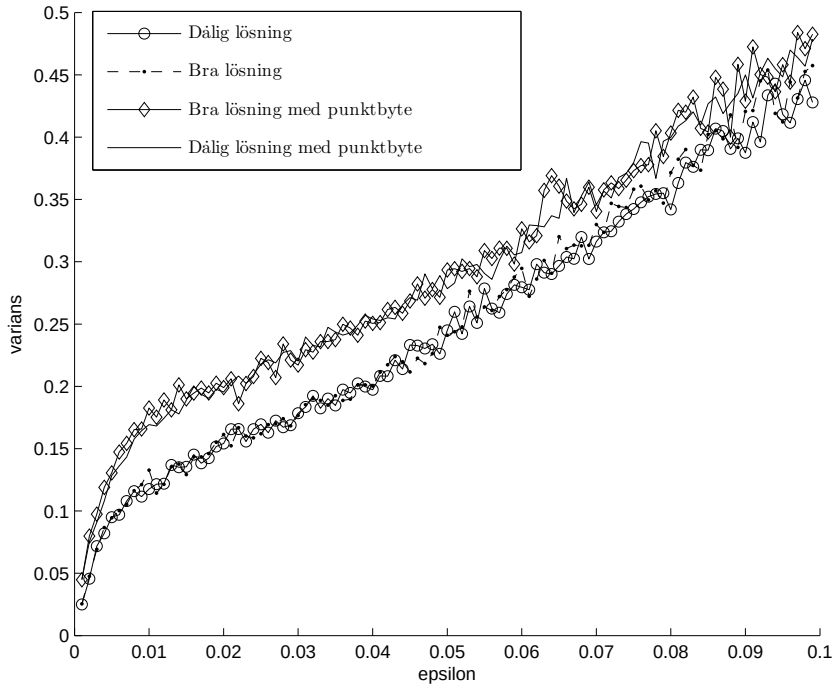


Figur 21: Histogram för hur lösningslängder varierar för Sierpińskikurvan. För att få fram detta resultat har vi genererat 10 000 slumpmässiga problem med 1000 uniformt fördelade punkter vardera. En normalfördelning kan antydas och tester bekräftar detta.



(a) Bästa lösningen med Sierpińskikurvan utav 10000 körningar. Man kan ana vissa strukturer som skulle kunna vara orsaken till det goda resultatet, bland annat viss regelbundenhet som fraktaler också uppvisar.

(b) Sämsta lösningen med Sierpińskikurvan utav 10000 körningar. Här ser man många överkorsningar och generellt något mer oordnad struktur än i föregående figur.



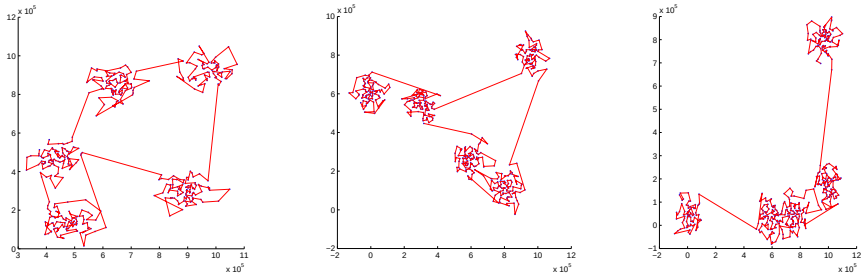
Figur 22: Stabilitet för bästa respektive sämsta Sierpińskiproblemet av 10 000. Här har vi jämfört algoritmerna med och utan punktbyte. Algoritmerna uppvisar liknande stabilitet oavsett hur bra eller dålig lösningen är. För små epsilon blir det en drastisk ökning i varians för att sedan övergå till vad som verkar vara ett linjärt samband.

Stabilitetsanalysen utfördes på ett givet problem med 500 punkter. Varje punkt i problemet har blivit stört från sitt ursprungsläge ett slumpmässigt avstånd med en normalfördelningssannolikhet. Ett problem har sedan störts 2000 gånger om för ett visst epsilon tillhörande normalfördelningen. Därefter har variansen extraherats för varje epsilon så att vi på så vis får ett mått på hur stabila våra lösningsalgoritmer är.

Vi mäter helt enkelt hur mycket våra lösningsmetoder varierar i längdmått beroende på hur

mycket vi stör problemet för att se hur robusta algoritmerna är. Analysen har gjorts för både uniforma och klustrade problem.

Nedan visar vi resultaten av stabilitetsanalysen för klustrade problem.

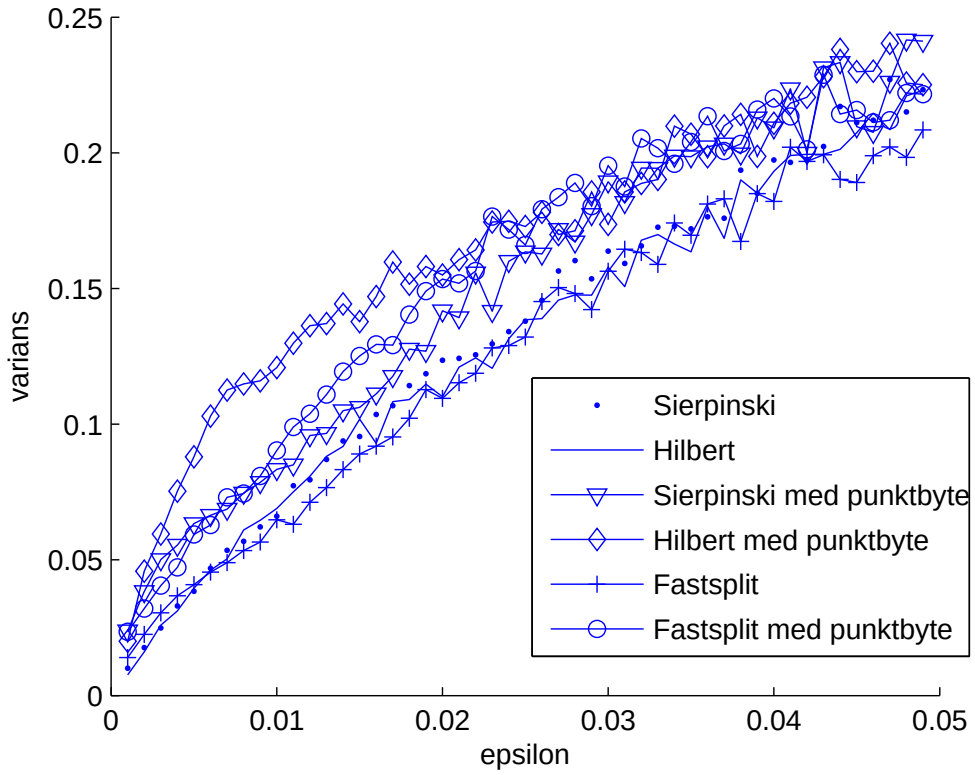


(a) Problem 2103. 5 rätt symmetriskt fördelade kluster.

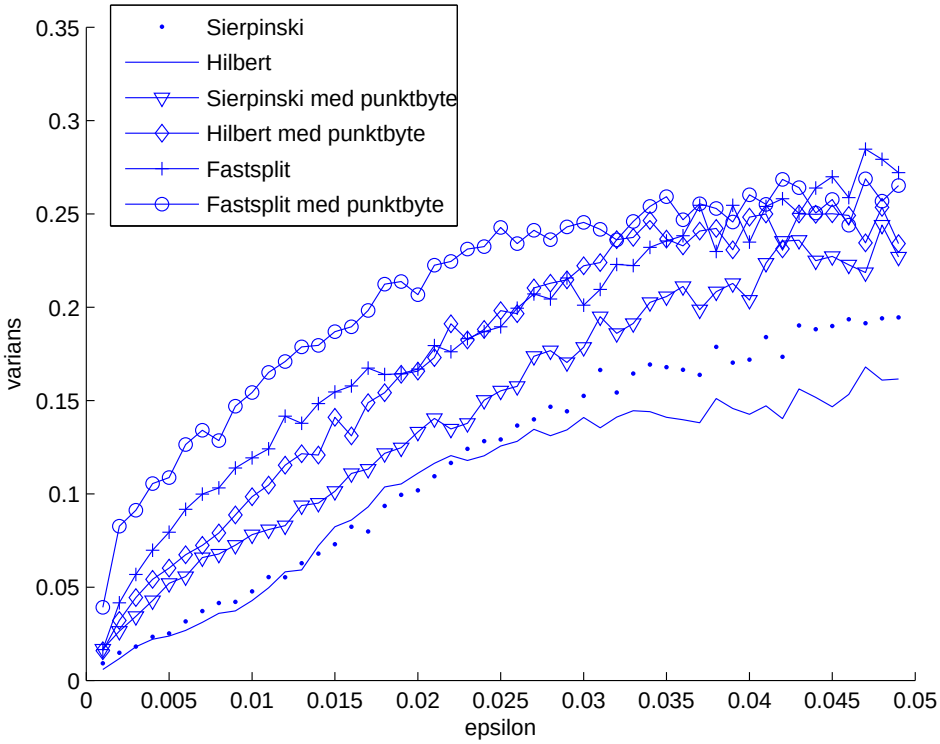
(b) Problem 64722. 5 något mindre regelbundna kluster

(c) Problem 9671. Här är 3 kluster så nära varandra att de bildar ett större kluster.

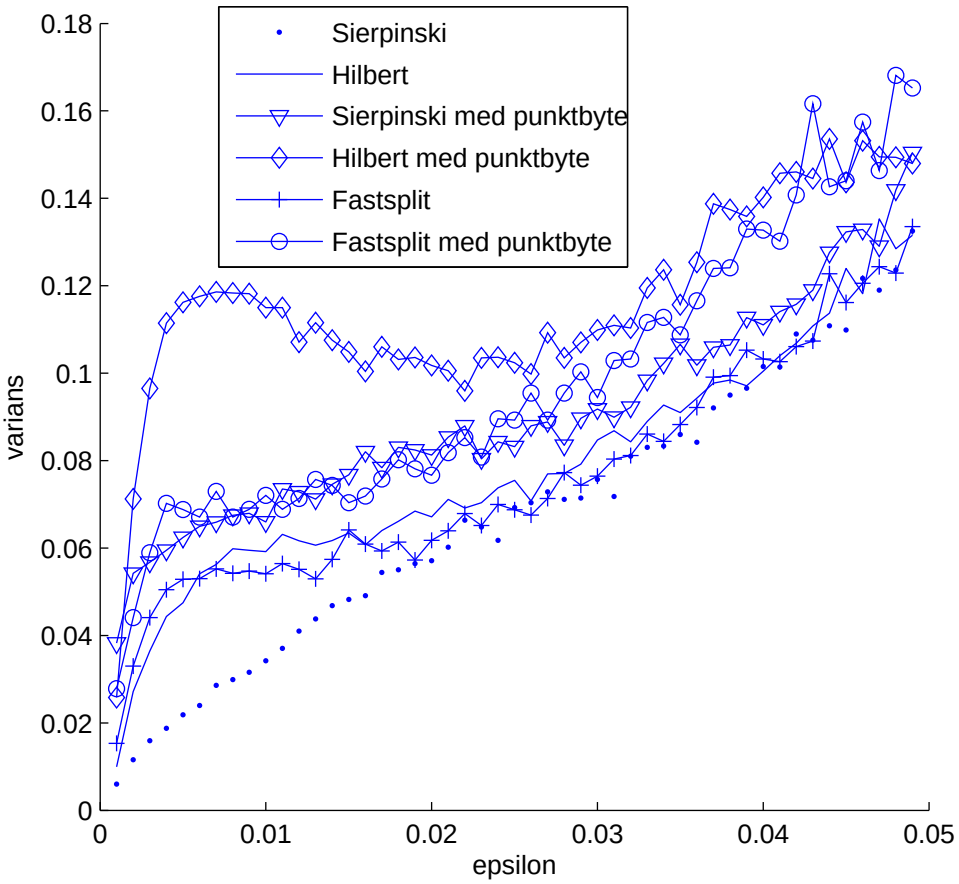
Figur 23: Dessa 3 problem valdes för att de uppvisade distinkta skillnader. Vår förhoppning med detta var att få mer givande resultat.



Figur 24: Stabilitetsanalys för problem 2103 med olika lösningsmetoder. Här blir Fastsplit mest stabil och Hilbert med punktbyte minst stabil.

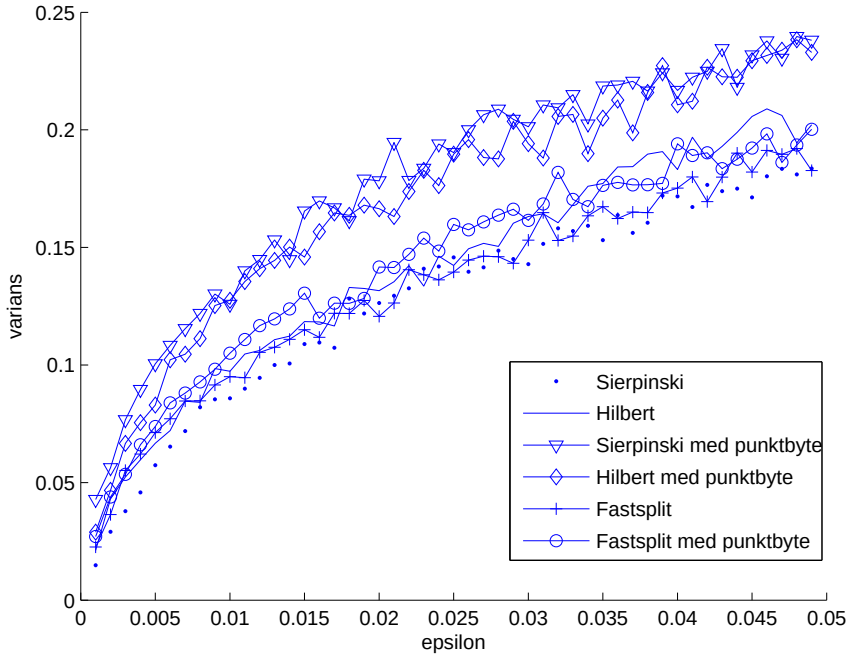


Figur 25: Stabilitetsanalys för problem 64722 med olika lösningsmetoder. Här blir Hilbert mest stabil och Fastsplit med punktbyte minst stabil.



Figur 26: Stabilitetsanalys för problem 9671 med olika lösningsmetoder. Här blir Sierpinski mest stabil och Hilbert med punktbyte minst stabil. En tydlig puckel syns för Hilbert med punktbyte och detta undersöks vidare i kapitel 6.

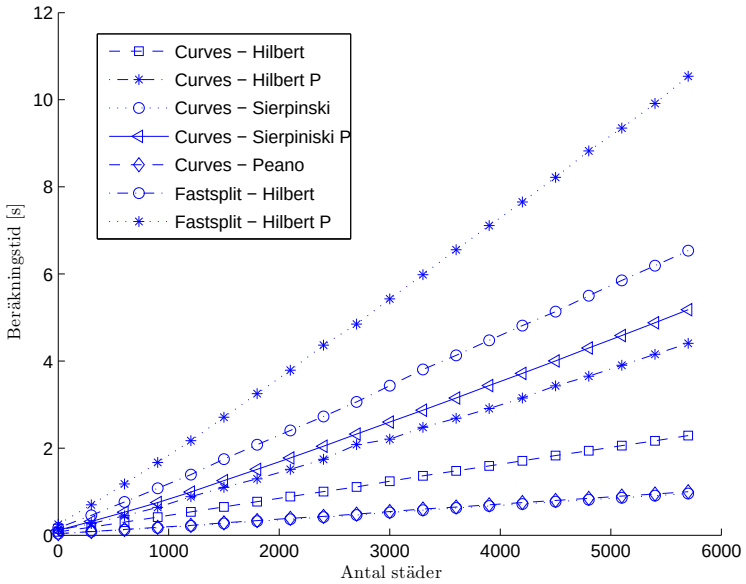
Resultatet av stabilitetsundersökningen för uniformt problem visas nedan.



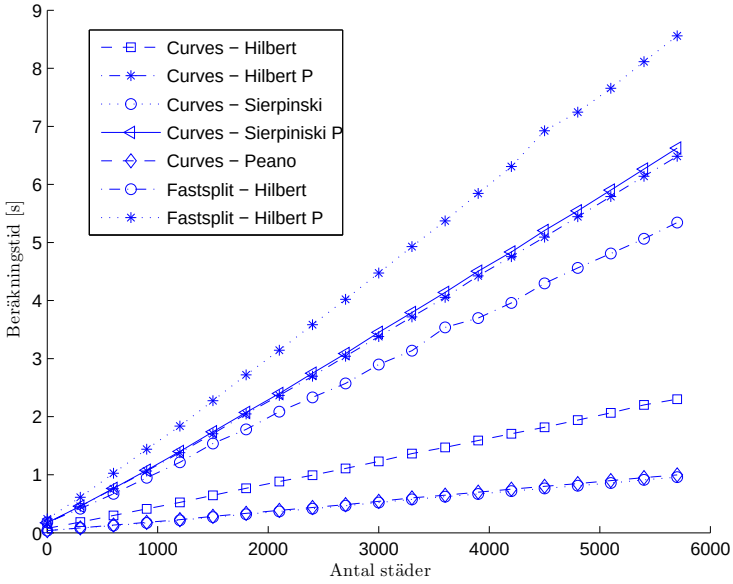
Figur 27: Stabilitetsanalys för uniformt problem 9671 med olika lösningsmetoder. Här blir Sierpinski mest stabil och Sierpinski med punktbyte minst stabil.

5.3 Beräkningstid

Beräkningstiden har utvärderats empiriskt genom ett stort antal exekveringar. Datan för uniforma problem finns sammanställd i figur 28 och datan för klustrade problem i figur 29.



Figur 28: En grafisk representation av beräkningstiden för uniforma problem för våra olika lösare. Vi ser att lösningstiderna förefaller växa ungefär linjärt med små antydningar till konvexitet.



Figur 29: En grafisk representation av beräkningstiden för klustrade problem för våra olika lösare. Även här växer beräkningstiderna nästintill linjärt med problemstorleken. Dock ser vi att vissa algoritmer arbetar snabbare på klustrade problem än för uniforma och vice versa.

6 Slutsatser och diskussion

I slutsatser och diskussion analyserar vi de resultat vi har och besvarar de frågor vi ställde i syftesdelen av rapporten.

6.1 Analys av prestanda

För våra tre rumsfyllande kurvor utan Fastsplit eller punktbyte visade sig Sierpińskikurvan vara bäst för både klustrade och uniforma problemuppsättningar. Vi tror att detta beror på att dess specifika konstruktionssätt ger mest önskvärda symmetriegenskaper för handelsresandeproblemet. Sämst blev Peanokurvan som ofta ger vägar upp till 15 procentenheter längre Sierpińskikurvornas lösningsvägar för klusterstrukturer. (se tabell 4 och fig. 17).

När vi sedan använder våra modifierade algoritmer så som Fastsplit och punktbytesmetoder ser vi att en kombination av punktbyte och Fastsplit för Hilbertkurvor ger oss bäst prestanda (se fig. 17) för uniforma problem.

För klustrade problem är Sierpiński med punktbyte bäst. Detta beror förmodligen på att Sierpińskikurvan genomlöper klustrade strukturer på ett fördelaktigt sätt. Fastsplit med punktbyte kommer dock mycket nära och är i genomsnitt två procentenheter längre i väg.

Dessa försök fick oss att vilja undersöka ett par saker närmare. Bland annat hur bästa och sämsta uppsättningen punkter kan se ut för våra kurvtyper, och hur stabila våra lösningsmetoder är. Resultaten av dessa försök analyseras i följande sektioner.

6.1.1 Uniforma och klustrade problem

Hur påverkar fördelningen av punkterna resultatet?

Vi kan se att våra lösare ger lösningar till klustrade problem med sämre optimalitet än för uniforma problem. Skillnaden är ungefär 15 procentenheter (se tabell 4 och fig. 17) för omodifierade lösningssätt, punktbyte och Fastsplit. Detta beror på att kurvtyperna genomlöper punktuppsättningarna på ett väldigt strukturerat sätt, vilket fungerar bra för uniforma problem men ställer till det när klustrena ligger olägligt.

Vår Fastsplit-algoritm designades för att handskas med just detta problem. Fastsplit förbättrar Hilbertkurvan genom att strukturera om den på ett sätt som tar klusterformationen i beaktelse.

Totalt sett verkar Sierpińskikurvan och Fastsplit vara bäst för uniforma och klustrade problem.

6.2 Stabilitet

När vi undersöker hur fördelningarna av optimalitet ser ut för Sierpiński, Hilbert och Peanokurvor för uniforma problem ser vi att de i högsta grad är normalfördelade. Detta följer av att optimalitetsmåtten är en summa av en stor mängd avstånd punkter emellan. Dessa avstånd har samma sannolikhetsfördelning och ändlig varians och kommer därför enligt centrala gränsvärdessatsen att ge oss en slutsumma som är normalfördelad.

Vi undersökte för- och nackdelar med stor varians och kom fram till att den främsta nackdelen är att resultatet man får blir för osäkert för våra tre kurvtyper. En fördel kan vara att man ibland får en riktigt bra lösning till ett visst problem, detta vägs dock upp av att man har lika stor sannolikhet att få en riktigt dålig lösning.

För bästa respektive sämsta Sierpińskilösningarna utav 10 000 problemuppsättningar ser vi att kurvformationen inte skiljer sig åt särskilt mycket. Möjligtvis ser vi att den bästa har mer fördelaktiga punktplaceringar som hamnar nära Sierpińskistrukturen vilket är varför vi tror att den just ger så bra resultat.

Stabilitetskurvorna för dessa två problem i figur 22 uppvisar mycket liknande stabilitetsegenskaper. Även detta något förvånande då vi trodde att en störning av punktmängden för bästa respektive sämsta uppsättningen punkter skulle ge avsevärda skillnader i stabilitet. De verkar dock vara mycket lika. Vi tror att detta beror på att Sierpiński har en naturlig varians i lösningslängder oavsett hur bra eller dålig lösningslängden är.

Anledningen till den snabbt ökande variansen för små epsilon tror vi beror på att den sämsta och bästa lösningen varierar som mest när man stör de väldigt lite. Variansmåtten blir mer opålitliga för stora epsilon vilket beror på att vi har stört det bakomliggande problemet såpass mycket att vi egentligen får helt nya problem.

Våra tre testuppsättningar (se fig. 24, 25 och 26) valdes så att klusterstrukturerna skulle vara olika varandra. Detta för att få mer givande stabilitetsanalyser. För problem 2103 ser vi att den stabilaste algoritmen verkar vara Fastsplit. Sämst blir Hilbert med punktbyte. Punktbyteskurvorna verkar generellt bli instabila och detta är inte förvånande då vi för varje störning även speglar denna störning ett antal gånger, vilket propageras i lösningslängdens varians.

För problem 64722 verkar Hilbertkurvan bli bäst medan Fastsplit ger mer instabila lösningar. Skillnaden mellan problem 2103 och 64722 är att den senare ger en mer ofördelaktig splittning för Fastsplit varpå dess lösningsvarians ökar.

Det tredje och mest intressanta problemet 9671 ser ut att ha en puckel för Hilbertkurvan med punktbyte, medan denna puckel uteblir för vanlig Hilbert. Vi tror att puckeln uppkommer då punktbytet sker i det ihoptryckta klustret. Detta ger väldigt stor varians fram till dess att epsilon blir stort nog att börja deformera det ursprungliga problemet totalt.

Generellt verkar det som att olika klusterstrukturer ger olika stabilitetsresultat. Vi kan därför inte förutsäga hur stabil en viss algoritm för ett visst klustrat problem är då detta beror på klusterstrukturen. Däremot kan vi se att punktbytesalgoritmer generellt är bland våra mer instabila algoritmer, och att rena Hilbert- och Sierpińskikurvor fungerar väl.

6.3 Körningstider

Vi kan se att alla våra metoder för att lösa TSP praktiskt taget anpassar sig efter en $\mathcal{O}(kn \log n)$ kurva där n är antal städer och k är antal nivåer vi använder i vår punktbytesalgoritm. Just för fig. 28 och 29 kommer vi upp i så stort antal städer direkt att den logaritmiska termen endast syns vagt. De snabbaste algoritmerna för uniforma problem är de omodifierade kurvtyperna Sierpiński, Hilbert och Peano. Lägger vi på punktbyte ökar k i vår tidskomplexitet och vi får något längre körningstider för dessa kurvor. Längst tid tar Fastsplit som är en något mer avancerad algoritm.

Vi får liknande resultat för klustrade problem. Dock ser vi att Fastsplit blir snabbare än både Sierpiński och Hilbert med punktbyte.

6.4 Vidare studier

En frågeställning som vi inte undersökt närmare är ifall det går att variera själva uppbyggnaden av kurvor för att lösa handelsresandeproblemet med rumsfyllande kurvor på ett effektivare sätt. En central frågeställning i en sådan undersökning skulle i så fall vara ifall det går att skapa en uppsjö av rumsfyllnadskurvor som löser TSP.

Vår ursprungliga ambition var just detta. Efter att vi läst en rapport [10] där de testat detta med genetiska algoritmer och fått föga imponerande resultat beslutade vi för att variera sättet kurvorna används på istället för hur deras grundläggande struktur byggs upp. Sättet att variera kurvornas användande i TSP-lösning är dock väldigt många och är något framtida kreativa problemlösare kan basera sina studier på.

Källor

[1] PAPADIMITRIOU, C. H. The Euclidean travelling salesman problem is NP-complete. Theoret. Comput. Sci. 4. 237-244.; 1977

[2] D. L Applegate, R. E. Bixby, V. Chvátal and W. J. Cook. The Traveling Salesman Problem: A Computational Study, Princeton University Press, ISBN 978-0-691-12993-8; 2006.

[3] Bild anpassad från xkcd.com `http://imgs.xkcd.com/comics/travelling_salesman_problem.png` [Uppdaterad 01 feb 2010, citerad 09 maj 2012]

[4] Bild anpassad från wikimedia.org `http://commons.wikimedia.org/wiki/File:Hilbert_curve.png` [Uppdaterad 23 okt 2005, citerad 17 feb 2012]

[5] Some combinatorial applications of spacefilling curves `http://www2.isye.gatech.edu/~jjb/mow/mow.html` [uppdaterad 22 nov 2011; citerad 28 feb 2012].

[6] Bartholdi, J.J, Platzman, L.K. An O(N log N) planar travelling salesman heuristic based on spacefilling curves. Georgia: Elsevier B.V; 1982.

[7] Held, M, Karp, R. M. A Dynamic Programming Approach to Sequencing Problems. Journal of the Society for Industrial and Applied Mathematics 10; 1962.

[8] Sagan H, Space-Filling Curves (Universitext). Berlin: Springer-Verlag; 1994.

[9] Jillian Beardwood, J. H. Halton and J. M. Hammersley. The shortest path through many points. Mathematical Proceedings of the Cambridge Philosophical Society, 55, 299-327; 1959.

[10] D.M. Tate, C. Tunasar, A.E. Smith. Genetically improved presequences for euclidean traveling salesman problems, University of Pittsburgh, Math. Comput. odelling Vol. 20, No. 2, pp. 135-143; 1994.

[11] Reinelt G, The Traveling Salesman - Computational Solutions for TSP Applications. Berlin: Springer-Verlag; 1994.

[12] Bild anpassad från wikimedia.org `http://upload.wikimedia.org/wikipedia/commons/6/64/Peanocurve.svg` [Uppdaterad 25 jun 2007, citerad 13 mar 2012]

[13] Bild anpassad från wolfram.com `http://mathworld.wolfram.com/images/eps-gif/SierpinskiCross_850.gif` [Citerad 13 mar 2012]

[14] Bild anpassad från wikipedia.org `http://en.wikipedia.org/wiki/File:Fractal_fern_explained.png` [Uppdaterad 16 maj 2006, citerad 28 feb 2012]

[15] Bartholdi, J.J, Platzman, L.K. Spacefilling curves and the planar traveling salesman problem. Georgia: Journal of the acm; 1989.

[16] PAPADIMITRIOU, C. H. The Euclidean travelling salesman problem is NP-complete. Theoret. Comput. Sci. 4. 237-244.; 1977