DD2440 Advanced Algorithms (fall 2009)
Project 1 – Euclidian TSP

Joel Pettersson
880519-0637
joelpet@kth.se

Linus Wallgren
880213-0099
linuswa@kth.se

October 30, 2009

1 Inledning

1.1 Problembeskrivning

Travelling Salesman Problem (TSP), också kallat handelsresandeproblemet, går ut på att hitta den kortaste hamiltoncykeln[1] i en komplett graf. Det är väl känt att TSP tillhör de NP-fullständiga problemen, och därmed ej kan förväntas lösas i polynomisk tid. Av den anledningen kommer denna rapport fokusera på olika sätt att ta fram approximativa lösningar.

Problemet framträder i en rad olika skepnader, där var och en innehåller skiftande restriktioner. Den variant av TSP som här behandlas beskrivs genom punkter i planet med euklidiska avstånd och därigenom är även triangelolikheten uppfylld.

Låt d_{ij} beteckna avståndet mellan städerna t_i och t_j. Då innebär triangelolikheten att $d_{ik} \leq d_{ij} + d_{jk}$ gäller. Det kan uttryckas som att sträckan mellan två städer t_i och t_k aldrig kan bli kortare genom att passera en annan stad t_j på vägen. I specialfallet då sträckan blir lika lång ligger städerna på en rät linje i planet. Målet är alltså att utifrån dessa förutsättningar minimera utrycket i (1), där n är det totala antalet noder i grafen.

$$\sum_{i=0}^{n}\sum_{j=0}^{n} x_{ij} \cdot d_{ij}, \qquad x_{ij} = \begin{cases} 1 & (i,j) \text{ tillhör turen} \\ 0 & \text{annars} \end{cases} \tag{1}$$

Slutligen måste turen beskriven av x_{ij} bilda en hamiltoncykel.

1.2 Metod

Här ges en kort sammanfattning av vilka metoder vi har valt att använda. Algoritmerna som nämns i följande stycke kommer att presenteras mer detaljerat senare i rapporten.

Inledningsvis implementerade vi den naiva startheuristiken som låter noderna vara sorterade från 1 till n. På den turen applicerade vi i nästa steg en

[1] En hamiltoncykel är en stig i en graf som passerar samtliga noder precis en gång, med undantag för noden där stigen startar och slutar.

lokal sökning, närmare bestämt 2-opt. När den lokala sökningen hade konstaterats fungera korrekt ersatte vi startheuristiken med en Nearest neighbor-konstruktion. Slutligen fick 2-opt kompletteras med 2.5-opt. Från början var det dessutom tänkt att vi skulle hinna implementera 3-opt, vilket dock ej blev fallet.

Ovanstående val av tillvägagångssätt och algoritmer baserades på de empiriska resultat som presenteras i *Notes for the course advanced algorithms* [2] på sidorna 133 och 135. Dessa tydde förvisso på ett övertag för 3-opt och framför allt för Lin-Kernighan med avseende på turernas kvalitet, men krävde i gengäld längre körningstid. Vidare bedömde vi Lin-Kernighan som jämförelsevis intrikat att implementera korrekt.

2 Konstruktionsheuristiker

2.1 Nearest neighbor

Nearest Neighbor fungerar precis som namnet antyder; från en slumpmässig startnod läggs hela tiden den närmsta noden till.

```
N := Nodes()
T := Tour()
for all nodes t_i ∈ N do
  best := ∞
  for all nodes t_j ∈ N do
    if t_j not used and d_ij < best then
      best := t_j
      add(T, best)
    end if
  end for
end for
```

2.2 Slumpmässig

Ett ytterst snabbt sätt att få fram en fungerande tur är att välja noderna i samma ordning som de ligger i indata. I och med att vi inte har några krav på indata kan ordningen på noderna vara helt slumpmässiga, vi får alltså en slumpmässig tur.

[2] http://www.csc.kth.se/utbildning/kth/kurser/DD2440/avalg07/algnotes.pdf

Då vi måste gå igenom alla noder kommer tidskomplexiteten bli $O(n)$.

3 Lokal sökning (optimering)

En lokal optimering tar, i fallet med TSP, en redan existerande tur och gör ändringar i begränsade delar av den för att på så sätt förbättra turens kvalitet, det vill säga minska dess totala längd.

3.1 2-opt

Om vi tar bort två kanter från en redan existerad tur finns det endast två sätt att sätta tillbaka dem på. Det ena är samma som tidigare, medan det andra är en eventuell optimering. Genom att titta på turens längd både innan och efter att vi flyttade de två kanterna kan vi få reda på om det faktiskt är en optimering eller ej.

3.1.1 Programbeskrivning

```
N := Nodes()
while moreImprovements do
  moreImprovements := false
  for all nodes t_i ∈ N do
    t_j := next(t_1)
    for all nodes t_k ∈ N do
      t_l := next(t_k)
      if d_ij + d_jk > d_ij + d_jk then
        reverseBetween(j, k)
        moreImprovements := true
      end if
    end for
  end for
end while
```

3.2 2.5-opt

Tekniken som beskrevs i föregående avsnitt kan generaliseras till k-opt för ett fixt tal k. Speciellt kan 2-opt göras om till 3-opt där först (inte helt oväntat)

tre kanter i turen tas bort. Detta resulterar i tre isolerade segment, av vilka alla tillåtna kombinationer evalueras. Den ihopsättning som ger kortast total längd väljs, innan proceduren upprepas.

Anledningen till att ovanstående tas upp i detta avsnitt är att i 2.5-opt betraktas de fall av splittringar i 3-opt där ett av de tre segmenten endast består av en enda nod. Då finns nämligen endast en tillåten kombination utöver den ursprungliga, vilket erbjuder en billig kontroll av möjligheten till optimering. Kontrollen går ut på att jämföra längderna av de kanter som tas bort kontra de kanter som läggs till. Hur detta går till visas utförligare i programbeskrivningen nedan.

2.5-opt kan kombineras med 2-opt på några olika sätt. Ett sätt är att kombinera ihop sökningarna så att man för varje kandidat till stad t_j i pseudo-koden för 2-opt evaluerar motsvarande 2-opt samt även 2.5-opt:en som placerar den ensamma noden i turen mellan de två varandra intilliggande noderna.

Alternativt kan man låta 2-opt köra tills en lokalt optimal tur har skapats och först därefter applicera 2.5-opt. Detta visade sig i vårt fall gå snabbare, men innebar också en smärre försämring av turernas kvalitéer. Eftersom tiden hos KATTIS är begränsad, valde vi att gå på denna snabbare linje.

3.2.1 Programbeskrivning

$N :=$ Nodes()
while improvement **do**
  improvment := false
  for all nodes $t_i \in N$ **do**
    $t_j = \text{next}(t_1)$
    for all nodes $t_k \in N$ **do**
      $t_l = \text{next}(t_k)$
      $t_m = \text{next}(t_l)$
      if $d(t_i, t_j) + d(t_k, t_l) + d(t_2, t_m) > d(t_4, t_2) + d(t_3, t_2) + d(t_1, t_5)$ **then**
        place t_j after t_k
        improvement := true
        restart t_i loop
      end if
    end for
  end for
end while

3.3 Optimeringar

För att snabba upp våra lokalsökningar implementerade vi grannlistor som höll koll på de närmaste grannarna till varje nod. Det är större sannolikhet att en granne till en nod kan ge en förbättring i turen än en nod som ligger långt borta. Vi låter alltså bli att titta på de noder som ligger en bit ifrån den kant vi håller på att optimera med vår lokala sökning, vilket snabbar upp körtiden men samtidigt minskar kvaliteten på approximationen något.

När vi gör våra lokala optimeringar beräknar vi inte hela turens längd för att se om vi har en förbättring eller ej, vi beräknar endast vikterna på de kanter som flyttas.

Det visade sig vara ganska vanligt att vi vill få reda på vart i turen en viss nod är, därför implementerade vi en lista som innehåller nodernas position i listan med noder. Därmed slipper vi linjärsöka igenom hela varje gång vi vill ha reda på var en nod befinner sig. Kostnaden för att hålla denna lista uppdaterad betraktade vi som liten i sammanhanget.

Vi valde även att förberäkna alla avstånd och lagra dem i en matris. Vi beräknar avståndet mellan noder såpass ofta att det visade sig löna sig att strunta i symetrin hos avstånden, då kollen som behövde ske i distance metoden (en if-sats) visade sig vara betydligt mycket långsammare än att sätta ytterligare $n^2/2$ värden i matrisen.

4 Christofides algoritm

Christofides algoritm har en körtid på $\mathcal{O}(n^3)$ och fungerar på följande vis.

Vi börjar med att skapa ett minimalt spännande träd[3] i grafen. Därefter finnes den matchning mellan alla noder av udda grad som har minimal total vikt.

Sedan kombineras den minimala matchningen med det minimala trädet, vars resultat gås igenom ända tills en Euler-cykel funnits. För att få en Euler-cykel måste vi hoppa över alla noder som vi redan varit i. När vi sedan har kommit tillbaka till noden vi startade i har vi fått en hamiltonkrets, d.v.s. en uppskattning på TSP. När vi går igenom strukturen kan vi till exempel

[3]Ett spännande träd är ett träd som spänner upp alla noder i grafen, medan ett minimalt spännande träd är det spännande träd som har den minsta totala kantsumman.

alltid gå till den "högraste" noden, och ifall vi redan har besökt den så lägger vi inte in den i vår tur igen.

Det är det andra steget i algoritmen, att skapa den minimala matchningen, som kräver längst tid – nämligen $\mathcal{O}(n^3)$, vilket dominerar algoritmen i övrigt och gör den "långsam".

5 Diskussion

5.1 Startheuristik

Det första vi insåg var att startalgoritmen gav stor skillnad på vårt resultat, trots att vi hade en fungerande 2-opt, som ska ge relativt bra resultat, fick vi inte alls bra approximationer. Det visade sig vara startheuristiken som var suboptimal. Istället för att använda sig av Nearest neighbor – som senare visade sig vara mycket bättre – använde vi den av indata givna turen. Efter att ha justerat detta såg vi drastiska förbättringar i resultatet.

5.2 Analys

Vi lyckades implementera alla algoritmer som vi hade tid att försöka oss på. Våra ursprungliga planer var att hinna implementera 3-opt men dessvärre stötte vi på tekniska problem vid implementeringen av 2-opt som fördröjde arbetet mer än vad vi hade räknat med.

För tillfället skalar vårt program väldigt dåligt mot ökad storlek på grannlistorna, vilket också är ett resultat av förseningen vi upplevde. Exempelvis kördes vår kod på kattis på $2,5$ sekunder med en granlista av storlek 11, ökade vi storleken 12 fick vi istället Time Limit Exceeded, vilket innebär att det skulle ta mer än 4 sekunder för koden att köra klart, vilket nästan är en fördubbling av tiden.

Om vi gör samma analys på en lokal dator tar det $0,4$ sekunder att köra klart ett testfall vi konstruerat med 1000 noder när vi har grannlistor av storlek 15. Ökar vi istället storleken på grannlistorna till 20 tar det $0,5$ sekunder att köra klart samma test, vilket ger en helt annan bild än den vi får från Kattis. Sannolikt är att Kattis har hittat ett testfall som är otroligt elakt mot vår kod, vilket gör att tiden inte skalar linjärt mot antalet grannar.

Vårt program är uppdelat i ett antal delar och i ett visst testfall med 1000 noder som indata får vi följande resultat, där n är antalet noder och k är antalet kanter i vår grannmatris.

Operation	Tidskomplexitet	Körningstid
Läsa in noderna	$\mathcal{O}(n)$	16,8 %
Beräkna alla kantvikter	$\mathcal{O}(n^2)$	11,3 %
Hitta närmaste grannarna till var nod	$\mathcal{O}(n * k)$	7.6 %
Nearest Neighbour	$\mathcal{O}(n^2)$	0.6 %
2-opt	$\mathcal{O}(n^2 * k)$	40.9 %
2.5-opt	$\mathcal{O}(n^2 * k)$	5,6 %
Skriva ut turen	$\mathcal{O}(n)$	0.6 %
Övrigt	-	16.6 %

Vi ser att den absolut största delen av tiden går åt till 2-opt. Både den höga tidskomplexiteten samt det faktum att vi behöver göra väldigt många operationer (ändra ordning på en hel del element i vår tur) bidrar till att den tar upp så stor del av tiden.

Att 2.5-opt går såpass fort trots tidskomplexiteten har antagligen att göra med att de flesta optimeringar redan är gjorda av 2-opten, som körs innan.

Att beräkna alla kantvikter är relativt dyrt, men det lönar sig ganska snart då kantvikterna är otroligt eftertraktade. Vi ser till exempel att Nearest neighbour tar väldigt kort tid trots att den har kvadratisk tidskomplexitet. Detta är i stort tack vare att vi förberäknat resultaten samt även de närmaste grannarna.

Framförallt tror vi att vi kan vinna ganska mycket tid genom att optimera koden rent programtekniskt, dvs utan att förändra kvaliteten på resultatet. Vi hade förhoppningen att detta skulle leda till att vi hinner med fler grannar och således får ett bättre resultat alternativt möjligheten att implementera flera optimeringsalgoritmer.

5.3 Framtida optimeringar

Vi skulle kunna implementera en så kallad don't look bit. Om vi inte hittar några förbättringar för en nod och ingen av deras grannar förändrats, är det osannolikt att det finns fler förbättringar att göra för den noden. Listan av don't look-bitar innehåller just den informationen – huruvida det hänt någon förändring med vår nod eller dess grannar eller ej. Men även om det minskar

körtiden minskar det också kvaliteten på resultatet, då vi riskerar gå miste om optimeringar.

Likaså kan man snabba upp koden med en annan datastruktur för själva turen. För tillfället lagrar vi turen i en vanlig vektor. Om man istället använder sig av en trädliknande struktur som håller koll på ordningen hos dess barn kommer operationen `reverse` att gå betydligt mycket fortare, vilket för tillfället tar upp det mesta av körtiden när vi kör 2-opt.

5.4 Arbetsfördelning

Vi delade upp de avancerade kodstyckena mellan oss på följande vis.

Joel skrev 2.5-opten samt metoden för att flytta in en nod mellan två andra medan Linus skrev startheuristikerna och metoden för att generera alla grannar.

2-opten arbetade vi med tillsammans då det hjälpte oss att få förståelse för hur optimeringen fungerade.