TDDD86 – Laboration #3

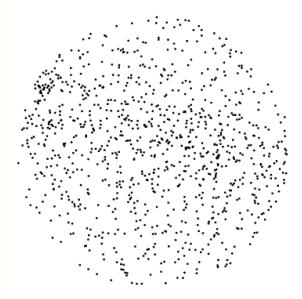
8 september 2022

I den här uppgiften får du öva på att implementera och använda en datastruktur baserade på utökningsbara länkade listor. Filerna du behöver för att komma igång finns som `labb3.tar.gz` på kurshemsidan.

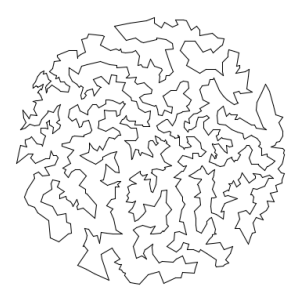
Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 3 redovisning''` och en `git push`. Se till att filerna **Tour.h** och **Tour.cpp** är med. Informera sedan din assistant genom att maila honom/henne.

Handelsresandeproblemet (TSP)

Givet N punkter i planet är målet för en handelsresande att besöka alla punkter (och komma hem igen) samtidigt som den totala längden av resan är så kort som möjligt. Implementera två giriga heuristiker för att hitta bra (men inte optimala) lösningar till handelsresandeproblemet (*Traveling Salesman Problem, TSP*).



(a) 1000 punkter.



(b) Optimal tur.

TSP är ett viktigt problem, inte bara för att det finns så många handelsresanden som vill minimera sitt resande, utan snarare för den stora mängden tillämpningsområden som fordonsruttning, kretskortsborrning, VLSI-design, robotstyrning, röntgenkristallografi, schemaläggning och beräkningsbiologi.

TSP är ett omryktat svårt kombinatoriskt optimeringsproblem. I princip kan man enumerera alla möjliga turer och välja den kortaste — i praktiken är antalet möjliga turer så stort (ungefär $N!$) att detta inte är görbart. För stora N känner ingen till någon effektiv metod för att hitta den kortaste möjliga turen för en given mängd punkter. Däremot har många metoder studerats som verkar fungera väl i praktiken, även om de inte är garanterade att producera den bästa möjliga turen. Sådana metoder kallas *heuristiker*. I den här uppgiften ska du implementera insättningsheuristikerna **nearest neighbour** och **smallest increase** som bygger upp en tur inkrementellt. Börja med en en-punktstur (från första punkten tillbaka till sig själv) och iterera följande process till det inte finns några punkter kvar.

- **Nearest neighbour**: Läs in nästa punkt och lägg till den till turen *efter* den punkt vilken den är närmast. (Använd den första sådana punkten om det finns flera punkter som är närmast.)
- **Smallest increase**: Läs in nästa punkt och lägg till den till turen *efter* den punkt där den resulterar i minsta möjliga ökning av turlängden. (Använd den första sådana punkten om det finns flera punkter med den egenskapen.)

Du har tillgång till en datatyp för punkter i planet, `Point`, med följande gränssnitt.

```
class Point (2D point data type)
--------------------------------------------------------------------------------
       Point(double x, double y)                 // create the point (x, y)
string toString()                                // return string representation
  void draw(QGraphicsScene *scene)               // draw point on scene
  void drawTo(Point that, QGraphicsScene *scene) // draw line segment between
                                                 // the two points
double distanceTo(Point that)                    // return Euclidean distance
                                                 // between the two points
```

Din uppgift är att skapa en datatyp, `Tour`, som representerar en sekvens av punkter besökta i en TSP-tur. Representera turen som en *cirkulär länkad lista* av noder, en för varje punkt. Till din hjälp finns en färdig nod-datatyp, `Node`, där varje nod har en `Point` och en pekare till nästa `Node` i turen. Dessutom har du tillgång till

huvudprogrammet `tsp.cpp` som använder din `Tour`-klass för att hitta bra lösningar till givna instanser av TSP-problemet. Bland de givna filerna finns också många testfiler att prova med och förväntat utdata för vissa av dessa.

implementationsdetaljer:
I den här uppgiften specificerar vi exakt vilka datamedlemmar du får ha i `Tour`-klassen. Du måste ha exakt följande datamedlem, du får inte ha några andra.

- en pekare till första noden i den cirkulära länkade listan

Du måste använda vår `Node`-struktur och din länkade lista måste ha följande publika medlemmar. Se `Tour.h` för mer detaljer.

```
class Tour
--------------------------------------------------------------------------------
       Tour()                       // create an empty tour
       ~Tour()                      // free all memory used by list nodes
  void show()                       // print the tour to standard output
  void draw(QGraphicsscene *scene) // draw the tour on scene
   int size()                       // number of points on tour
double distance()                   // return the total distance of the tour
  void insertNearest(Point p)       // insert p using nearest neighbor heuristic
  void insertSmallest(Point p)      // insert p using smallest increase heuristic
```

Observera att det här är tänkt att vara en övning i att manipulera länkade listor. Det betyder att du måste använda vår `Node`-struktur och att du inte får modifiera den. Du får inte heller skapa några arrayer, `vector`s, stackar, köer, mängder, avbildningar, andra STL-containrar eller andra datastrukturer; du måste använda länkade noder. I synnerhet behöver du göra följande:

- **Lägg till kommentarer** i `Tour.h` och `Tour.cpp`. (Kommentarer till metodhuvuden ska placeras i `.h`-filen. Implementationerna av dessa medlemmar i `.cpp`-filen behöver inte några kommentarer i metodhuvudet.)
- **Deklarera nödvändiga privata medlemmar** i `Tour.h`.
- **Lägg till** `const` där det är lämpligt till de publika medlemmar som inte modifierar listans tillstånd.
- **Implementera kropparna** för alla medlemsfunktioner och konstruktorer i `Tour.cpp`.
- **Hantera dynamiska minnet** ditt program ska se till att dynamiskt allokera och frigöra minnet som behövs för noderna.

Lämplig arbetsgång:

1. Studera den givna koden.
2. I felsökningssyfte kan du skapa en konstruktor `Tour(Point a, Point b, Point c, Point d)` som tar fyra punkter och skapar en cirkulär länkad lista med dessa fyra punkter i. Skapa först fyra noder och tilldela en punkt till varje. Länka sedan samman noderna i en cirkel.
3. Implementera metoden `show`. Den ska traversera varje nod i den cirkulära länkade listan med början i den första noden och skriva ut varje punkt. Den här metoden kräver bara några rader kod men det är ändå viktigt att tänka noga på dem eftersom att felsöka länkade listor kan vara svårt och frustrerande. Börja med att bara skriva ut den första punkten. I cirkulära länkade listor pekar den sista noden tillbaka på den första, så se upp för oändliga loopar.
   Testa metoden genom att ändra i huvudprogrammet så att det definierar fyra punkter, skapar en ny tur av dessa fyra punkter och anropar turens `show`-metod. Nedan följer ett förslag på hur det kan se ut.

```
// define 4 points forming a square
Point p(100.0, 100.0);
Point q(500.0, 100.0);
Point r(500.0, 500.0);
Point s(100.0, 500.0);

// Set up a Tour with those four points
// The constructor should link p->q->r->s->p
Tour squareTour(p, q, r, s);

// Output the Tour
squareTour.show();
```

Om din metod fungerar korrekt får du följande utdata.

```
(100.0, 100.0)
(500.0, 100.0)
(500.0, 500.0)
(100.0, 500.0)
```

Testa din metod `show()` på turer med 0, 1 eller 2 punkter och kontrollera att den fortfarande fungerar. Du kan skapa sådana instanser genom att modifiera 4-nodskonstruktorn till att bara länka samma 0, 1 eller 2 av de givna punkterna. (Om du inte tilldelar instansvariabeln som pekar ut den första noden har du en tom tur. Om du sätter instansvariabeln att peka på någon nod och länkar den tillbaka till sig själv har du en tur som innehåller en punkt.)

4. Implementera `size()`. Den blir väldigt lik `show()`.
5. Implementera `distance()`. Den blir väldigt lik `show()`, så när som på att du måste anropa `distanceTo()` i punktdatatypen. Om du lägger till ett anrop till `distance()` i den kvadratiska turen du skapade ovan bör resultatet bli `1600.0`.
6. Implementera `draw()`. Den är också väldigt lik `show()`, så när som på att du måste anropa `drawTo()` i punktdatatypen.
7. Implementera `insertNearest()`. För att avgöra vilken nod du ska sätta in `p` efter behöver du beräkna det euklidiska avståndet mellan varje punkt i turen och `p` genom att traversera den cirkulära länkade listan. Lagra hela tiden noden som innehåller den hittills närmsta punkten och dess avstånd till `p`. Efter att du hittat den närmsta noden skapar du en nod som innehåller `p` och sätter in den *efter* den närmsta noden. Det betyder att du måste ändra på `next`-pekaren i både den nyligen skapade noden och den närmsta noden. Bland de givna filerna finns `tsp10-nearest.ans` som innehåller det förväntade resultatet, en tur med längd `1566.1363`, för 10-punktersproblemet `tsp10.txt`. Notera att den optimala turen har längd `1552.9612`, så den här heuristiken ger inte den bästa turen i allmänhet.
8. Nu borde det vara relativt enkelt att skriva `insertSmallest()`. Den enda skillnaden är att du ska sätta in punkten `p` där den resulterar i minsta möjliga ökning av turens totala längd. Som kontrollpunkt finns filen `tsp10-smallest.ans` som innehåller det förväntade resultatet, en tur med längd `1655.7462`, för 10-punktersproblemet `tsp10.txt`. I det här fallet är **smallest insertion**-heuristiken faktiskt sämre än **nearest insertion**-heuristiken (trots att detta inte är typiskt).

Möjliga utökningar

E3 — en bättre heuristik (3 poäng):
Lägg märke till att en tur som har delsträckor som korsar varandra kan transformeras till en kortare tur utan korsningar. För att göra detta kan det vara nödvändigt att lägga till fler metoder i `Tour`. Skapa en ny branch som heter E3 och implementera denna strategi som en ny publik medlem i `Tour`. Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 E3 redovisning''` och en `git push`. Se till att filerna **Tour.h** och **Tour.cpp** är med. Informera sedan din assistant genom att maila honom/henne.

E4 — egen heuristik (3 poäng):
Implementera en egen heuristik som ger, i 60 sek eller mindre, minst 1% kortare väg än insert-smallest på tsp1000. För att göra detta kan det vara nödvändigt att lägga till fler metoder i `Tour`. Utdataformatet måste dock vara samma som i det befintliga programmet. Skapa en ny branch som heter E4. Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 E4 redovisning''` och en `git push`. Se till att filerna **Tour.h** och **Tour.cpp** är med. Informera sedan din assistant genom att maila honom/henne.

E5 — tävling (4 poäng till vinnaren, 3 till 2:an och 2 till 3:an):
Programmet måste skickas in innan den deadline som anges på kurshemsidan för tävlingen (brukar vara runt slutet av november). Juryn kommer att plocka 5 banor bland dem som kommer med i uppgiften. För varje sekund över 60sek (på maskinerna i SU-salen) per bana får programmet som straff 2% extra i distans för den banan. Om en bana tar mer än 120 sek diskvalificeras programmet. Vinnaren blir det icke-diskvalificerade programmet som ger minsta summan. I övrigt gäller samma begränsningar som i E4. Skapa en ny branch som heter E5. Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 E5 redovisning''` och en `git push`. Informera sedan din assistant genom att maila honom/henne.