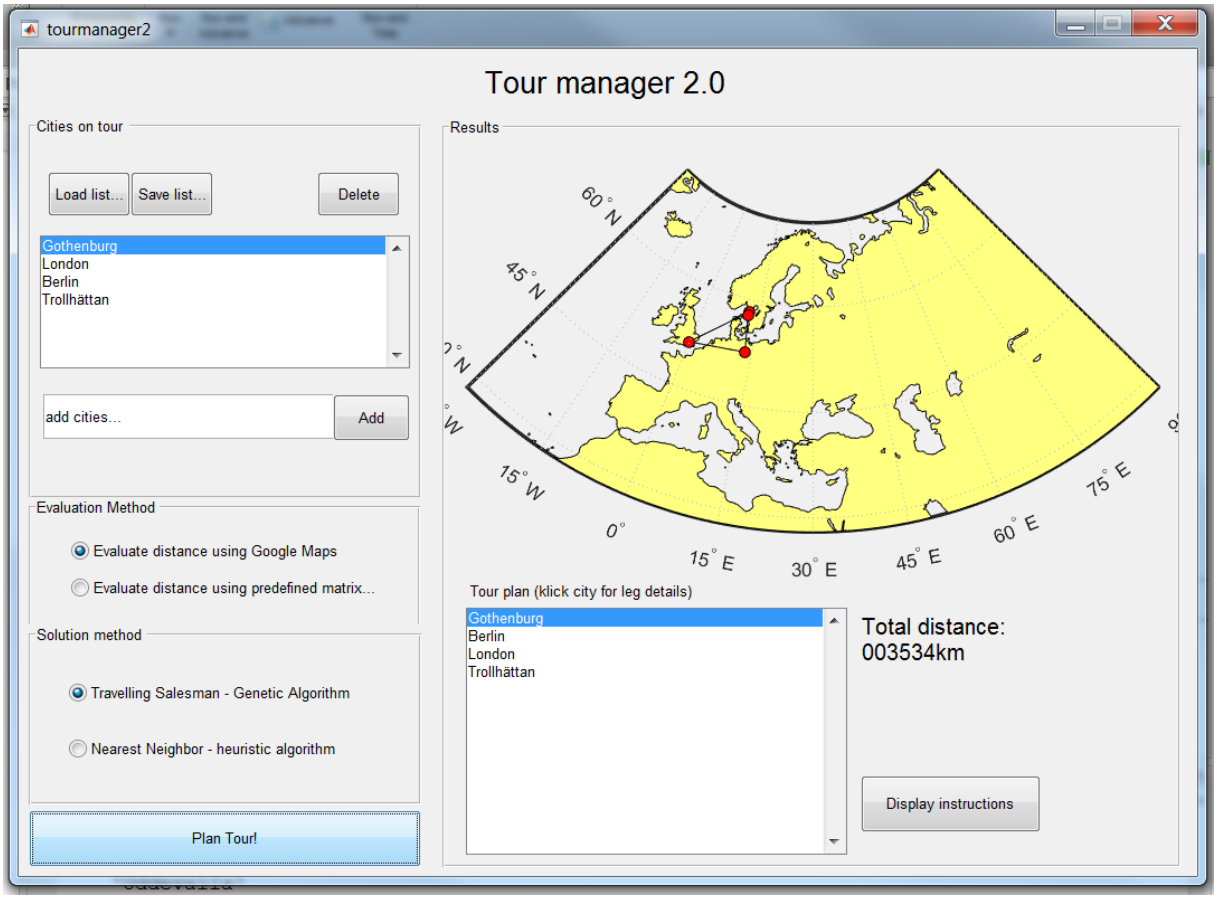
Inlämningsuppgift 3: Reseplanerare

Att planera en resa som besöker en bestämd grupp städer med så kort resedistans som möjligt är ett klassiskt problem inom matematiken. För den moderne resenären finns möjligheten att hitta den kortaste vägen mellan två städer i Google Maps – men för att effektivt planera större turnéer med besök i flera städer optimalt behöver *besöksordningen* för städerna också bestämmas.

I denna uppgift ska du alltså utveckla ett helt program i Matlab.



Figur 1: En turnéplanerare. I figuren exempel på resa mellan norra Europas viktigaste städer.

Mål: Inlämningsuppgiften avser att testa

1) Algoritmer, variabelhantering, loopar, logiska satser, script
2) Skapa och anropa funktioner
3) Hantera beräkningar i Matlab med värden i listor (vektorer)
4) Hantera struktur-objekt och cellmatriser

Figur 2: En första uppdelning av uppgiften i deluppgifter, representerat schematiskt. Pilarna indikerar informationsflöde: Steg 2 förutsätter alltså steg 1 och så vidare.

Inlämningsuppgift 3

Ditt uppdrag är att skriva ett program som planerar en resa mellan ett givet antal städer. För att kunna angripa denna uppgift behöver du först dela upp den; ett förslag är angivet i figur 2 här ovanför. Det innebär att uppgiften reduceras ner till de fyra delproblemen att välja städer och hitta data om dessa städer, att beräkna avståndet mellan dessa städer, att välja i vilken ordning städerna ska besökas och slutligen att presentera resultat för användaren.

Det är naturligtvis möjligt att göra detta på olika sätt där de olika sätten är olika svåra – precis som i alla utvecklingsprojekt måste därför ambitionsnivån sättas i ett tidigt skede. I figur 1 visas ett grafiskt användargränssnitt som hanterar såväl val av städer som presentation av resultat, det vill säga både punkt 1 och punkt 4 ovan. Ett sådant gränssnitt är för de flesta (bestämt för de som är intresserade av att göra en produkt av sin kod!) mer tilltalande än en minimalistisk textbaserad kommunikation med användaren, men genom att kasta sig in i utveckling av gränssnitt ökar å andra sidan risken en färdig produkt inte kan levereras i tid. I många fall är det därför rimligt att börja med att få programmet att fungera på en lägre ambitionsnivå för att sedan förbättra det del för del.

Del 1: Att välja städer och skaffa data om städerna

I filen `cities_on_west_coast.mat` finns en struktur med namnet `CityToCityRoadData`. Strukturen består av 13 vägsegment som tillsammans ska bygga upp vägnätet mellan de viktigaste städerna i Västra Götaland och har fält enligt förlagan nedan:

```
CityToCityRoadData(1)
ans =
        distance: 4719
      endAddress: 'Aalborg'
          endLat: 57.0488
          endLng: 9.9210
    startAddress: 'Aarhus'
        startLat: 56.1629
        startLng: 10.2041
```

Här finns alltså information om ett antal vägsträckor, och även positionen på de städer som de trafikerar, men inte så mycket mer. För att hålla ordning på vilka städer som besökts behöver den här informationen reduceras till en lista över vilka städer som finns. För att kunna göra det behövs funktionen `unique`. Det är även viktigt att kunna konkatenera cellmatriser och ta ut data ur strukturer.

När det fungerar med `cities_on_west_coast.mat` finns motsvarande data för hela Europa, automatläst från Wikipedia och Google Maps, i `cities_on_the_europe_roadmap.mat`.

Google Maps-variant

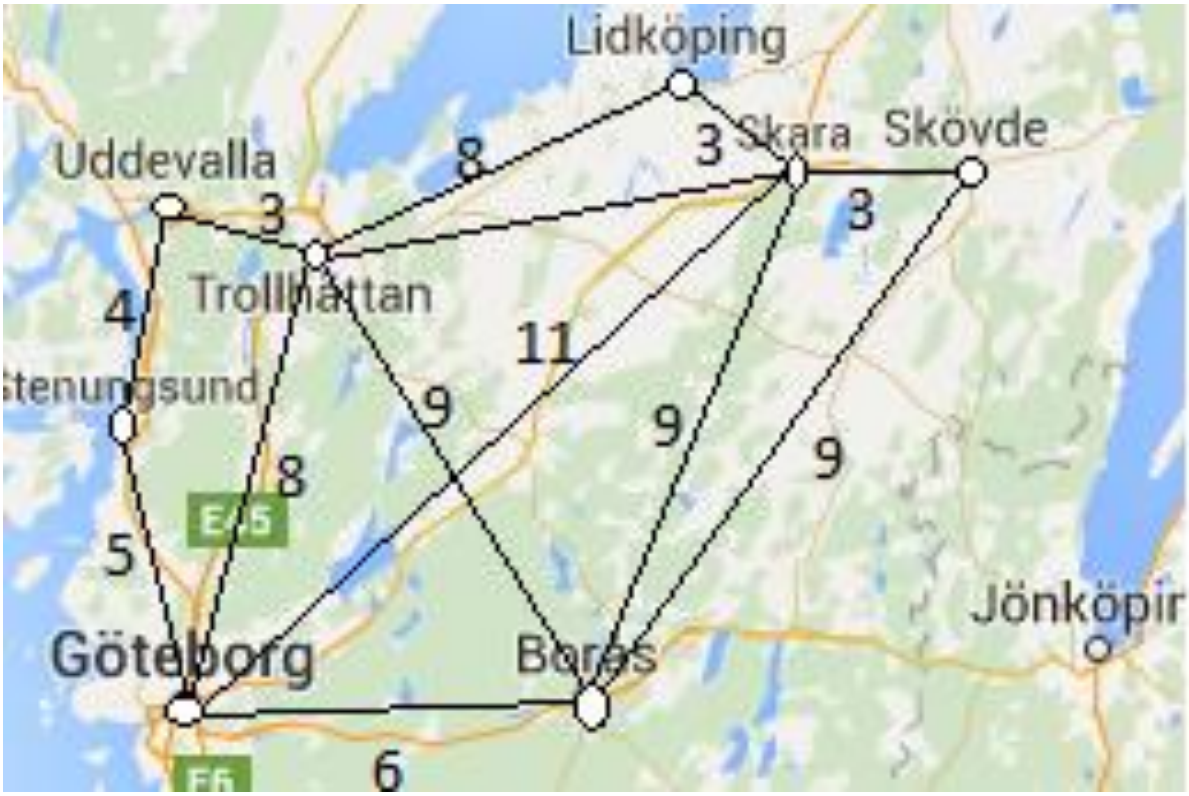
Information om avstånd går också att få för två namngivna städer vilka som helst genom att använda det API-gränssnitt[1] som Google tillhandahåller. Om namnet på städerna finns i två textsträngs-variabler med namnen `startCity` och `endCity` kan distansen läsas med hjälp av följande kod:

```
googleString=[...
      'https://maps.googleapis.com/maps/api/directions/json?origin='...
       startCity '&destination=' endCity '&ie=ANSI'];
data=webread(googleString);
```

För att kunna ta oss utanför närområdet behöver vi alltså implementera en mer avancerad version som kan ta reda på avstånd mer generellt. Datan som kommer från Google ovan är samlad i en struct liknande den i filerna, men med många fler fält.

Del 2: Att hitta avstånd mellan städerna

I strukturen `CityToCityRoadData` fås information om distansen mellan närliggande städer enligt figur 3. Genom att sätta upp en tabell över vägnätet[2] så att rad och kolonn motsvarar en given stad och varje rad i tabellen alltså visar avståndet till de andra städerna från en given stad – i exemplet nedan är rad ett Göteborg, varifrån det finns direkta väglänkar till Stenungsund (50km), Trollhättan (80km) , Borås (50km) och Skara (110km).



Figur 3: Förenklad variant av det västsvenska bilnätverket.

[1] API – Application Programming Interface. Google är för övrigt väldigt öppna med sin infrastruktur för den som vill prova. Se vidare https://developers.google.com/

[2] Detta kallas med matematiskt språk för en grafmatris.

Denna tabell blir av naturliga skäl symmetrisk – det är lika långt från Göteborg till Stenungsund som från Stenungsund till Göteborg. Dessutom blir värdet på diagonalen (dvs `A(n,n)` för alla `n`) noll eftersom sträckan från en stad till sig själv blir noll.

```
A=[ %Exempel på grafmatris: Förenklade bilnätverket i västsverige.
    0 5 0 8 6 0 11 0;    %Göteborg
    5 0 4 0 0 0 0 0;     %Stenungsund
    0 4 0 3 0 0 0 0;     %Uddevalla
    8 0 3 0 9 8 8 0;     %Trollhättan
    6 0 0 9 0 0 9 9;     %Borås
    0 0 0 8 0 0 3 0;     %Lidköping
   11 0 0 8 9 3 0 3;     %Skara
    0 0 0 0 9 0 3 0];    %Skövde
```

För att kunna räkna ut körsträckan kan det vara bra att också räkna ut hur långt det är kortaste vägen även mellan två städer som inte har en direktlänk i vägnätet. Vi kan nämligen (generellt) inte anta att det kommer att gå att täcka in alla städer genom att helt enkelt alltid bara åka till närmsta grannstad. För att lösa deluppgiften krävs alltså:

- Inläsning och viss omstrukturering av indata
- En algoritm som beräknar den snabbaste vägen mellan två städer i ett vägnät

Dijkstras algoritm beräknar den snabbaste vägen från en stad till alla de andra städerna i en tabell - grafmatris - som den ovan. Det är en abstrakt algoritm: Som väl är finns den lättillgänglig, till exempel från denna kursens gamla tentor (tentan från oktober 2014 uppgift 5 behandlar just Dijkstras algoritm). Eftersom tentan och tillhörande lösning är en offentlig handling är det fullkomligt acceptabelt att bara kopiera den lösning som finns där *förutsatt att du på ett tydligt sätt indikerar varifrån den kommit.* Andra implementeringar av Dijkstras algoritm finns också att tillgå, till exempel i funktionen `graphshortestpath`, eller från Matlab Central file exchange.

Observera att en implementering av Dijkstras algoritm inte är nödvändig om en egen Google-läsare skrivits, eftersom Google Maps på egen hand kan beräkna den kortaste vägen mellan två städer, och dessutom gör det utan att begränsa sig till de vägar du läst in.

Del 3: Hitta rutt mellan städer

Själva turordningsproblemet, att finna den turordning som ger kortast körsträcka samtidigt som alla städer täcks in, kallas inom matematiken för *Handelsresandeproblemet* och är väldigt svårt att lösa, speciellt snabbt.[3] Det är därför vanligt att istället för att hitta den allra bästa lösningen implementera en s k *heuristisk* metod, dvs en metod som antagligen inte ger den absolut bästa lösningen, men som är snabb och ger ganska bra resultat.

En vanlig och rimligt enkel algoritm att implementera är ”närmsta ännu inte besökta granne” – algoritmen. Denna algoritm implementeras av dig i funktionsfilen `nearestneighbor`.

Själva funktionen `nearestneighbor` ska ha som indata en grafmatris och ett index som indikerar startpunkten. I exempelmatrisen ovan innebär alltså 1 att resan tar sin början i Göteborg. I första steget ska algoritmen välja den närmsta grannen. Startar resan i Göteborg blir den närmsta grannen Stenungsund – alltså stad nummer 2 i exempelmatrisen. Vid alla efterföljande steg ska algoritmen välja den närmsta granne som ännu inte besökts. Det kan vara bra att använda en av två indexvektorer,

[3] Googla! Det finns massor med algoritmer som löser handelsresandeproblem i Matlab därute.

`tourList` eller `notOnTourList`, för att hålla ordning på vilka städer som ännu inte blivit inplanerade. (Om det inte sköts ordentligt så kommer algoritmen vilja åka direkt tillbaka till Göteborg igen.)

Genetisk Algoritm-variant

En genetisk algoritm är ett optimeringsverktyg som används för att lösa komplicerade optimeringsproblem genom att på ett systematiskt sätt söka igenom de möjliga lösningarna och fokusera på de bästa. Genom att implementera en sådan algoritm kan potentiellt en bättre lösning än den som hittas av `nearestneighbor` hittas. Återigen finns det hjälp att få på internet: Ladda ner och använd `tsp_ga`, som är en implementering av en Genetisk Algoritm specifikt framtagen för handelsresandeproblemet. Finns på Matlab Central file exchange.

Del 4: Presentera resultat

Från del 3 ska huvudprogrammet få en lista på alla städer i tur och ordning; denna information ska förmedlas till användaren. Det enklaste sättet att göra detta är genom utskrift till kommandofönstret. Ett roligare sätt är att använda en karta med den valda vägen, som till exempel kan ritas med hjälp av Matlabs mapping toolbox (Se uppgift 2 på tentamen från oktober 2014 för förlaga och inspiration). Om Google Maps har använts för att läsa data så fi

Ledning

För att kunna hålla reda på allt som ska göras bör uppgiften delas in i mindre delar. Funktionsfiler hjälper till med detta genom att det i huvudprogrammet inte behöver framgå hur själva beräkningarna görs utan bara vad som behövs för indata och hur utdata ska behandlas vidare. Det är en god idé att börja med funktionen `nearestneighbor`! Testa den sedan på det förenklade Västgötska vägnätet i figur 3 – kopiera och klistra in matrisen från detta dokument till en början - och övertyga dig om att funtionen fungerar innan du tar itu med huvudprogrammet! Följande delar kan huvudprogrammet med fördel delas upp i:

1) Läs in data från fil
    a. **Möjlig utveckling:** Ge användaren möjlighet att välja startstad.
    b. **Möjlig utveckling:** Ge användaren möjlighet att välja bort städer från en lista.
    c. **Möjlig utveckling:** Ge användaren möjlighet att själv välja städer genom att använda Google Maps API interface.
    d. **Möjlig utveckling:** Implementera ett grafiskt användargränssnitt som i figur 1, till exempel med hjälp av `guide`[4] eller med hjälp av successiva dialogrutor (`inputdlg`, `listdlg`, `msgbox` etc.)

2) Konstruera en lista över antalet unika städer
3) Skapa en matris med lika många rader och kolonner som listan

4) Lägg till varje vägsegment i matrisen genom att hitta raden som motsvarar `startAddress` och kolonnen som motsvarar `endAddress` och lägg till avståndet `distance` – fälten hittar du i `CityToCityData`-strukturen, som ju innehåller de segment du ska lägga till.
    a. **Möjlig utveckling:** Använd istället Google Maps-datan från 1c.

[4] OBS! Den här funktionaliteten är ganska besvärlig att sätta sig in i och kan komma att ta ganska så mycket tid.

5) Anropa `dijkstra` en gång för varje stad och skapa genom att ställ de resulterande kolonnvektorerna bredvid varandra i en A-matris som är full sånär som på diagonalen
    a. **Möjlig utveckling:** Hoppa över detta steg om 1c implementerats och A redan är full.
6) Anropa `nearestneighbor` och få ut turnélista som indexvektor
    a. **Möjlig utveckling:** Implementera också `tsp_ga` och låt användaren välja metod: Snabb eller optimal.

7) Skriv ut turnélistan med `fprintf` eller `disp`
    a. **Möjlig utveckling:** Rita också upp en karta
    b. **Möjlig utveckling:** Presentera rutt och karta i det grafiska användargränssnittet från 1d.
    c. **Möjlig utveckling:** Ge användaren möjlighet att också få detaljerad information om varje led i resan (informationen finns tillgänglig i Google Maps-strukturen från 1c.)

Följande avsnitt i Jönsson är extra viktiga för just denna inlämningsuppgift (utöver det som är gemensamt med uppgift 1):

while- och for-loopar: Avsnitt 6.6-7
funktionsfiler, funktionsanrop: Avsnitt 7.2-7.3

Läs också om `struct` och `cell` i avsnitt 4.15-16 och i hjälp och dokumentation.

Formalia

Det är viktigt att programmen är generellt skrivna så att det är enkelt att använda dem igen., Om till exempel andra turneplaner dyker upp ska bara datafilen behöva bytas ut, med (som mest) en minimal ändring i programmet.

Varje m-fil (skript eller funktionsfil) ni skriver i kursen ska ha ett programhuvud med kort förklaring till vad programmet gör och vilka som har gjort det, med namn och födelsedag. För funktionsfiler ska in- och utvariabler anges extra tydligt.

Exempel:

```
% MASSVOLUME: Funktion som beräknar massa och volym av en geometrisk form.
%
% MASSVOLUME beräknar massa och volym av klossar. Följande typer stödjs:
%   'prisma' (tresidig) x1=bas,     x2=höjd,    x3=längd
%   'rätblock'          x1=bas,     x2=höjd,    x3=längd
%
% INPUT:    sort    -   textsträng som förklarar vilken sort det handlar om
%           x1,x2,x3-   variabler för volymsberäkning, se ovan
%           rho     -   densitet (defaultvärde=1);
%
% OUTPUT:   m       -   massa
%           V       -   Volym
%
% CALL:   [m,V]=massvolume(sort,x1,x2,x3,rho);

% TME136 (c) Anders Johansson 821206 2014-09-09.
```

Elektronisk inlämning

Inlämningsuppgiften ska rapporteras genom att programfilerna lämnas in elektroniskt via PingPong-systemet senast **den 12:e oktober kl. 23:59**. Ladda upp filerna, **en uppsättning per person**. Programkoden ska vara väl kommenterad och alla filer ska innehålla ett programhuvud med beskrivning av vad programmet i filen gör. Glöm inte att fylla i namn i programhuvudet (så att vi är säkra på vem som gjort uppgiften)!

Minsta godkända nivå är att ha gjort följande: Ett huvudprogram utan utvecklingar, dvs enligt steg 1-7 i listan. Kalla huvudprogrammet `tourmanager`. En funktionsfil `nearestneighbor` enligt beskrivning. En kopierad `dijkstra` med tydliga källhänvisningar.

Vid misstanke om plagiat skickas ärendet direkt till disciplinnämnden. (Gäller inte om man glömt källhänvisningen i `dijkstra` – då får man en retur).

Lycka till!