Optimisation & Operations Research

Haide College, Spring Semester

Practical 4 (1%)

See website for practical and due dates

Work through the below instructions in MATLAB and MATLAB Grader as indicated. Submit to MATLAB Grader by the due date.

Travelling salesperson problem

Mike Chen

The aim of this practical is to solve the travelling salesperson problem with a variety of techniques from the course notes.

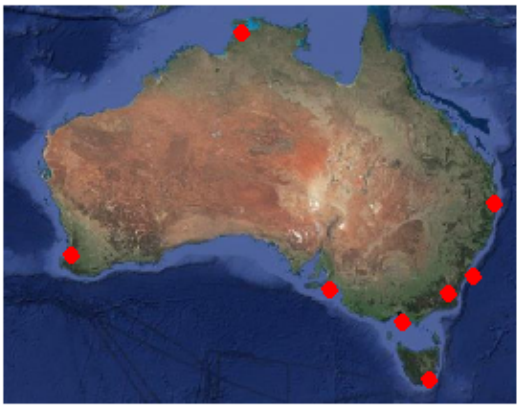
Recall that the travelling salesperson problem you are given a set of cities and the distance between each city. The goal is to find the shortest path that visits each city exactly once and then returns to the starting point.

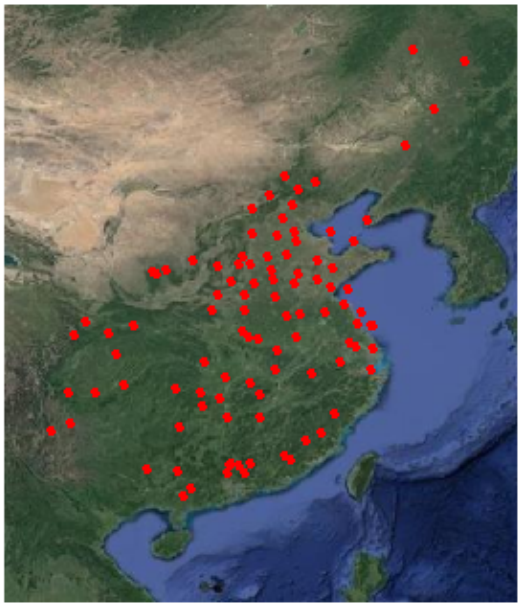
In this practical you will explore the following techniques to solve this problem:

- brute force search: perform an exhaustive search of all possible paths
- greedy heuristic: as described in the course notes (section 4.8)
- genetic algorithm: as described in the course notes (section 5.4)

We will consider data from Australia (8 cities) and China (100 cities) to use test these algorithms, see below for plots. Note the dots representing the cities do not quite appear in the correct location on the map ...various cities on the east coast of China are inland and Sydney is in the ocean!

You can perhaps guess what the shortest routes around the Australian cities is, but finding the shortest route around the Chinese cities is significantly more difficult.





Sources: Google Maps (image); simplemaps.com (city location data)

---

1. **Preliminaries.** Start by downloading the files for this practical from Cloudcampus, these are in the file `OOR_P4_files.zip`. Unzip this folder to your computer and open MATLAB.

   The first thing to do is load the data.

```
load('Prac4_au_cn_tsp.mat')
```

This adds the following variables to your workspace:

- `au`: a table of the names and locations of the 8 Australian cities.
- `au_distances`: a matrix of the distances (in kilometres) between the Australian cities.
- `cn`: a table of the names and locations of the 100 Chinese cities.
- `cn_distances`: a matrix of the distances (in kilometres) between the Chinese cities.

Double-click on each of these variables in the workspace browser to see their contents. The latitude and longitude of each city is given in `au` and `cn`. In the distance matrices, you can look up the distance between each city using their number in the table. For instance, Adelaide is city 1 in `au` and Hobart is city 5. The distance between them is:

```
au_distances(1,5)
```

As described in the course notes, we will write the route between the cities as a permutation of the integers. Note that here we **do not** require that the route starts with a 1 (although we will sometimes assume this).

For example, in the Australian cities a route of $(1, 8, 2, 7, 3, 6, 5, 4)$ would be Adelaide – Sydney – Brisbane – Perth – Canberra – Melbourne – Darwin – Hobart.

We will write our roues as a row vector in MATLAB, so the above route would be

```
au_route = [1 8 2 7 3 6 5 4];
```

A helper function `tsp_distance` to calculate the length of a route is provided. It takes as inputs the route vector and the matrix of city-to-city distances.

Calculate the distance of the above route as follows:

```
au_route_legnth = tsp_distance(au_route,au_distances)
```

The route can then be visualised using the provided plotting function `plot_route_au` (a function `plot_route_cn` is also provided). This can be called as follows:

```
plot_route_au(au_route,au,au_distances);
```

Task: Try constructing a few routes of your own, then calculate the route distance and make a plot! For the Chinese data your might find MATLAB's `randperm` function useful. Construct a route `cn_route` (vector with number 1 to 100) and call the following commands:

```
cn_route_length = tsp_distance(cn_route,cn_distances);
plot_route_cn(cn_route,cn,cn_distances);
```

2. **Brute force search.** One way to find the shortest route is to enumerate all possible paths, calculate their length and select the shortest one.

   You are provided with a function `tsp_brute_force` that does just that. We won't go into the details of this code, but will look at how it performs as the number of cities is increased.

   First, let's test it on the Australian data (and plot the solution) as follows:

   ```
   [route_bf,route_length_bf]=tsp_brute_force(au_distances);
   ```

   You should see that this gives what you might have guessed to be the shortest route around the Australian cities.

   Task: A exhaustive search for a route around the 100 Chinese cities is too computationally, but let's find out just how difficult this would be!

   Run the following code to time how long it takes to find the shortest route for different numbers of cities from the Chinese data:

   ```
   n_max = 11;
   bf_times = nan(n_max,1);
   for j=2:n_max
       tic
       tsp_brute_force(cn_distances(1:j,1:j));
       bf_times(j) = toc;
   end
   ```

   (Optional, try increasing `n_max` in the above code to 12. This may take a few minutes to run. Note, if you need stop a long MATLAB script from running enter `Control+c` in the command window.)

   Looking at the resulting times in `bf_times`, how long would you estimate it would take this exhaustive search to find the shortest path for 20 cities? How about for all 100 cities?

---

3. **Greedy heuristic.** A greedy heuristic for the travelling salesperson problem is as follows:

   - Choose a starting location, for example city 1.
   - Choose the nearest city to city 1 as the second town in the route.
   - Choose the nearest city (not already visited) to the second town as the third town in the route.
   - Continue until the route passes through all towns.

   See also section 4.8 of the notes for details.

   Task: You are provided with a template for this greedy heuristic `tsp_greedy.m`. Complete the missing code in line with the description above.

   Hint: MATLAB's `min` function can return the index of the minimum element in a vector if called as follows

   ```
   [min_value,index]=min(vector)
   ```

   Using this index value could be very here, since it is the number used to represent a

particular city in the route vector.

Now, run your code on the Australian data as follows:

```
[greedy_route_au,greedy_route_length_au] = tsp_greedy(au,start)
```

Try a few different starting cities. What is the difference in route length between this solution and the shortest route found in Question 2?

Finally, run your code on the Chinese data as follows:

```
[greedy_route_cn,greedy_route_length_cn] = tsp_greedy(cn,start)
```

Again, try a few different starting cities. How long is the route given by the greedy heuristic (ie. what is `greedy_route_length_cn`)? How would you expect that this compares to the real shortest route?

Once you have a feel for how the above commands work, run the following to generate a set a routes for all different starting cities for the Chinese data (we will use these in the next question):

```
greedy_route_lengths_cn=nan(1,100);
greedy_routes_cn=nan(100);
for j=1:100
    [greedy_routes_cn(j,:),greedy_route_lengths_cn(j)] = tsp_greedy(cn_distances,j);
end
```

---

4. **Genetic algorithm.** We will now implement the genetic algorithm for the travelling salesperson. This is quite a complicated algorithm, so you are provided with a nearly complete code `tsp_genetic.m` and just need to write two functions that implement the key random elements in the algorithm.

   Recall that the genetic algorithm for the travelling salesperson problem looks something like this:

   1. Generate an initial population of random routes, plus some solutions generated by the greedy heuristic from Question 3.
   2. Repeat the following until finished ...
      (a) Evaluate the length of each route in the current generation.
      (b) Use the current generation to generate a new generation (of the same size).
      - Elitism: keep the best (shortest) routes from the current generation.
      - Create new offspring as follows.
        – Selection: choose two routes (Parent 1 and Parent 2) from the current generation at random, with rank selection as described in the notes.
        – Crossover: For a random $1 \leq n_{\text{crossover}} \leq n_{\text{cities}} - 2$, keep the first $n_{\text{crossover}}$ cities from Parent 1 then scan Parent 2 to complete the route (see example on page 444 of the course notes).

  - Mutation: swap the position of two random cities, but only accept mutation it results in a shorter route (see example on page 446 of the course notes).

(c) Replace old population.

(d) Decide whether to finish. Here we just repeat for a fixed number of generations.

Spend a few minutes familiarising yourself with the provided code. Your task here is to complete the code by writing functions to perform the crossover and mutation described above.

Task: Use the provided template `crossover.m` to implement the crossover procedure described above.

> **Hint:** The only tricky part here is to scan the second parent. You many find that the MATLAB function `ismember` useful here.

Task: Use the provided template `mutation.m` to implement the mutation procedure described above.

> **Hint:** Since the first city in our route is always 1, make sure you do not mutate the first element here. Also, note that when calculating route distance you need to call `tsp_distance` on the original and mutated route.

Task: Run the genetic algorithm on the Australian cities data to confirm that it finds the shortest route. The command would be as follows:

```
n_population = 20;
n_generations = 30;
n_elite = 4;
[route_GA_au,route_length_GA_au] = tsp_genetic_algorithm(au_distances, ...
    n_population,n_generations,n_elite,[]);
```

In the above command the last input to the function is empty, this just means the initial population is a set of random routes.

Note that since there is an element of randomness in the algorithm you may need to run it multiple times find this route (or increase the parameter `n_generations`).

Task: Now run the genetic algorithm on the Chinese cities data as follows:

```
rng(2024)
n_population = 20;
n_generations = 1000;
n_elite = 10;
n_greedy_initial = 5;
initial_routes = greedy_routes_cn(randi(100,[n_greedy_initial 1]),:);
[route_GA_cn,route_length_GA_cn] = tsp_genetic_algorithm(cn_distances, ...
    n_population,n_generations,n_elite,initial_routes);
```

Here the initial population is 5 Greedy routes selected at random and 15 random routes. Note the `rng(2024)` which resets MATLAB random number generator so you can get reproducible results here.

You should adjust the four parameters above until you get a solution you are satisfied with. With an appropriate choice of the these parameters tt should be possible to get a reasonable optimal solution in quite a short computational time (1–2 minutes). Since there is an element of randomness here you may get a different (perhaps better) result by running the code multiple times.

To get a feel for how well the genetic algorithm has improved on the initial population you can look at a plot of the short route length in each generation:

```
plot(route_GA_cn)
```

or plot the shortest route from the initial population (from the greedy heuristic) next to the shortest route from the final population:

```
subplot(1,2,1)
plot_route_cn(route_greedy_cn,cn)
subplot(1,2,2)
plot_route_cn(route_GA_cn,cn)
```