Introduction to Computing Science

Simulated Annealing for the Travelling Salesman Problem

Introduction
The *Travelling Salesman Problem* (TSP) is the following: Given a collection of n cities, find the shortest route that visits every city *exactly* once.

This problem seems simple, but to date nobody has ever developed a truly efficient algorithm to solve this problem *exactly*. The problem is one of a category called combinatorial optimisation problems, which occur frequently in logistics. In the case of n cities, the number of routes to be checked to be sure to find the optimal solution is $O(n!)$, which is prohibitively expensive for quite small n.

The aim of this project is to write a program that can give an *approximate* solution efficiently. The algorithm used is unusual because

a. it works very differently from any algorithm you implemented in Imperative Programming,

b. it does not guarantee an optimal solution.

The algorithm works as follows: Select a random solution, i.e. a chain of all cities in arbitrary order. Loop until fed up with running the program, or no improvement is found in many iterations:

- Randomly change the route
- *Always* accept the change if it is an improvement
- *Sometimes* accept a worse solution.
- If accepted, replace the old solution with the new

During the iterations the likelihood that a worse solution replaces the previous is reduced. The reason for occasionally accepting worse solutions is that the algorithm does not easily get stuck on sub-optimal solutions.

Simulated Annealing
The above process is called simulated annealing, because it is based on the physical process of improving the strength of steel by gradually cooling it, rather than shock-cooling it, which results in brittle steel. The reason for this brittleness is that atoms do not get the time to settle in the best (= lowest-energy) positions in the crystal lattice. By cooling slowly, the atoms are bumped around by their neighbours more, which means they are removed from sub-optimal locations. These locations are local minima in the energy, and the atoms stuck there must receive an energy increase to escape from these bad positions (equivalent to accepting worse solutions occasionally). As the steel cools slowly, and the atoms settle in lower and lower energy positions, the likelihood of them being bumped

away reduces, because collisions with other atoms are less likely, and there is simply less (thermal) energy in the system.

We simulate this process in this case by assigning each solution with an "energy". In the case of TSP this would be the total path length. We choose an initial configuration C_0 of our n cities, and compute the initial energy E_0. We also choose an initial "temperature" T_0. We make a new *candidate* configuration C_{temp}, and compute its energy E_{temp}. We now select the next configuration C_1:

If $E_{\text{temp}} < E_0$: C_{temp} *always* becomes the new configuration C_1.

If $E_{\text{temp}} \geq E_0$: C_{temp} becomes the new configuration C_1 with probability

$$P = e^{\frac{E_0 - E_{\text{temp}}}{kT}}, \tag{1}$$

with k a constant; otherwise $C_1 = C_0$.

The equation for the probability P means that any new solution that is *much* worse than the old, i.e. ($E_0 \ll E_{\text{temp}}$) are unlikely to be chosen. As T is reduced, the probability that small increases in energy are accepted is reduced. In physics (real annealing) k is know as Boltzmann's constant, but because we are not dealing with real temperature we can just set $k = 1$, and change the "start temperature" T_0 to any value that gives the best results.

The above process is repeated a number of times (say m, times) at a given temperature. After that we lower the temperature slightly, for example using:

```
T = 0.99*T;
```

We then perform M annealing steps at the new temperature, and repeat this process until, either

- the desired temperature has been reached,
- some maximum number of iterations has been reached (equivalent to the first)
- The energy (total length of route) has not decreased for a certain number of iteration.

In principle, this is simple enough to implement, but the execution time, and time needed to fix various parameters correctly by experimentation may take a while.

The data structures
We first need the x and y coordinates of the n cities for example on a grid which we can populate randomly, by picking integers in a certain range. Next, we must compute the distances between the cities (assume straight lines). We can store these in the following variables:

```
#define NMAX   50

int  city[NMAX][2];          // x and y coordinates for each city


int distance[NMAX][NMAX]; // afstanden tussen steden
```

Constant NMAX can of course be set to other values. We can store the path in an array of NMAX integers:

```
int path[NMAX];
```

The program should ask for a value $1 <$n$\leq$NMAX of the actual number of cities to be simulated. Every value between 0 and n-1 must occur *exactly* once in the path. It is very easy to initialise the array in such a way that this condition holds (how?).

We can now devise several methods to change the path in such a way that the condition still holds:

- swap two arbitrary cities in the array,
- invert the order of an arbitrary section of the array
- perform a circular shift: move everything up (or down) the array by one place, and move the last (or first) to the first (last) position.

Implemented correctly, the array should still contain all cities exactly once.

Random numbers

Before we can start thinking about implementation, we need to consider how to generate (pseudo-)random numbers and bits. We use a standard library functions to do this. First declare

```
#include<stdlib.h>
```

You can then obtain random numbers with the function `rand()`. Getting a new random number and storing it in n is achieved by

```
n =  rand();
```

Function `rand` computes a pseudo-random number based on the previous number it generated. To initialise the sequence, we call

```
srand(seed);
```

with `seed` some integer value. For a given seed, the sequence of repeated calls to `rand` is completely fixed (but hard to predict for mere mortals). This allows you to generate the same "random" sequence in repeated experiments, if desired. By using e.g. the time when `srand` is called as seed, the sequence becomes really unpredictable.

Note that `rand` produces number between 0 and RAND_MAX, a predefined constant, guaranteed to be at least 32767. What you need is random numbers between 0 and some smaller number $n-1$. In the smallest case (for booleans) $n = 2$. One way to achieve this is using

```
r = rand() % n;
```

Though valid, it is not a good idea to do this for small values of n. Many random-number generators will yield a sequence

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 .....

for $n = 2$ (try it). For random booleans you can use:

```
r = ( rand() > (RAND_MAX / 2) );
```

For $n \geq 10$ the first form is usually OK.

Random floating point numbers uniformly distributed between 0 and 1 can be generated using

```
r = ((double) rand() ) /  ((double) RAND_MAX) ;
```

Assignment 1
Write a function to fill the array of positions of cities, for a given value of `n` $\leq$ `NMAX`. You can either use random positions or read them from the input. Compute the distances between them.

Assignment 2
Write functions to Initialize the path, and methods to perform the permutations listed above (use `rand()` to select arbitrary pairs/sections of the path).

Assignment 3
Write a function that computes the difference in energy $\Delta E = E_i - E_{\text{temp}}$ between configurations C_i and C_{temp}. Using this, and `rand()`, write a function that assigns C_{temp} to C_{i+1} with probability

$$P = e^{\frac{\Delta E}{T}}, \tag{2}$$

for negative values of ΔE, and otherwise assigns C_i to C_{i+1}. Question: what must the method do if $\Delta E \geq 0$?

Assignment 4
Construct a fully functional program for simulated annealing using the functions developed in the other assignments.

Test this with different configurations of input, values of `n` and initial temperature T, etc. Note: It is sufficient to declare just two paths: a current path and a candidate path. You then conditionally overwrite the current path with the candidate in the annealing step.