Practical 3: Doing the rounds

Instructions

This is an assessed practical in five parts, A–E. Each part of the practical guides you through the construction of part of a program.

To obtain credit for your work, you will need to submit electronically the most advanced version of your program.

The deadline for completion and electronic submission of Practical Exercise 3 is **2pm***, Monday 29th November (Week 11). In the absence of evidence of medical or other difficulties, work submitted after the deadline will attract zero credit.*

Aims

- To provide practice in the use of arrays in C.
- To provide an introduction to the use of heuristics to find approximate solutions to hard problems.

Assessment

The maximum credit that can be obtained for this practical is 35 marks (out of a total of 100 for the whole of the continuously assessed element of the course). The five parts, labelled A–E, differ in value: Part A is worth 12 marks, Parts B and C 7 marks each, Part D 5 marks, and E 4 marks. Try to complete at least Parts A–C, but don't spend *excessive* time on the practical, particularly Parts D and E. Note that a good quality solution to Parts A–C may possibly achieve a B grade, and Parts A and B alone may possibly achieve a pass mark.

Notes

- Read through this document *before* you reach the keyboard, and work out in advance what you need to do.
- If you are genuinely stuck, seek help immediately from your tutor, the course lecturer, or from the InfBase drop-in centre in Appleton Tower.
- It is fine to generally discuss the project with your fellow-students, or get help with debugging. However, you may neither share code, nor take code from others, in completing the task.

Preparation

This practical has associated template and other files. You will need to copy these files into your directory so that you can modify them.

To copy the files, make sure you are in the directory that you have created for this practical and issue the command

```
cp  /group/teaching/cp1/Prac3/*  ./
```

Electronic submission

In this practical there is just one program file to be submitted, namely `tour.c`. The specific form of the command for submitting this file is

```
submit cs1 cp1 P3 tour.c
```

In addition you are invited to submit a results file thus

```
submit cs1 cp1 P3 results
```

in which you return

- The output of your program on the test data in parts B, C, D and E, and
- if you attempt Part E, a description of the method you used and the output you obtained.

Compilation

The program you will add to is in the file `tour.c`. You may compile it by issuing the command `make tour` in the shell window. If compilation is successful, an executable file `tour` will be created; you may run it by typing `./tour` in the shell window. This produces an interactive version of the program in which you may input data by moving the mouse over the graphics window. This mode is good for testing and debugging.

You may also compile the *same* program by issuing the command `make ftour`. If compilation is successful, an executable file `ftour` will be created; `ftour` differs from `tour` in taking input from a file instead of the graphics window. You may run the program on data file `data50` by typing `./ftour < data50` in the shell window. (There are two other data files, `data25` and `data75`.) From time to time you will be asked to run your program on the data file `data50`, and enter the results obtained in the file `results`. This file forms part of your submission; see *Electronic submission.*

Introduction

We are given a set of n points in the plane, representing a collection of n cities which a salesman is required to visit. The classical *Travelling Salesman Problem* (TSP) is to find the shortest tour that visits each city once and returns to the starting city. Our version is simplified in two ways. Firstly, we assume that the distance between any two cities is just the Euclidean ("straight line") distance in the plane. If you prefer, there is a perfectly straight road from every city to every other. (This is the so-called *Euclidean* TSP.) Secondly, to simplify the programming slightly, we'll allow the salesman to start at any city and end at any other city; in other words, we don't require him to return to the starting point, though we do still insist that he visits all n cities. TSP is hard to solve exactly, so we'll explore *heuristics* for obtaining approximate solutions.

Part A [12 marks]

We return to the world of Practical 1, to the extent that we will be reusing the simple geometry library `"descartes.h"`. So a city will be represented by a variable of type `point_t`, and its position will be specified by the user clicking the (left) mouse button.

In this part we shall (a) allow the user to specify the positions of the cities by clicking the mouse in the graphics window, (b) display the cities and the route obtained by visiting them in the order in which they were input, and (c) output the length of that route. This is all very much like Part D of Practical 1. The only difference is that we shall need to *store* the locations of the cities so we can process them in subsequent parts of the practical. This is where the *array* comes in!

```
#define MAXCITIES 100

point_t city [MAXCITIES];
int numCities = 0;
```

The above C code declares an array `city` that can be used to hold up to 100 cities. In general, we won't want to use the whole array, so we shall have to remember how much of it we're actually using. The integer `numCities` is used to record how many cities there actually are. Of course, `numCities` should not exceed `MAXCITIES`.

Extend the program in the file `tour.c` by adding functions with headers:

```
int ReadCities(void)

void DrawTour(void)

double TourLength(void)
```

These are very similar to functions you were asked to write in Part D of Practical 1.

The first of these functions should accept a sequence of mouse clicks from the user (recall the function `GetPoint` from Practical 1!) and store the points (i.e., cities) in successive locations of the array `city` starting at `city[0]` (recall that arrays in C are indexed from 0). It is possible to assign one `struct` to another in C, so the assignment `city[i] = p` is perfectly valid, provided `i` is an `int` and `p` a `point_t`. Don't forget that `ReadCities` should set the variable `numCities` correctly, as well as the array `city` itself. The end of the input is marked by the user clicking the middle mouse button: this generates a bogus point with negative coordinates. The result returned by `ReadCities` should be false if any error occurred and true otherwise. There is probably just one possible error: what is it?

The array `city` now represents a possible tour, in which the salesman starts at `city[0]`, then visits `city[1]`, `city[2]`, an so on until `city[numCities - 1]`. The second function, `DrawTour`, should display that tour, by drawing the line segment from `city[0]` to `city[1]`, `city[1]`, to `city[2]`, and so on up to `city[numCities - 2]` to `city[numCities - 1]`. (Recall the functions `LineSeg` and `DrawLineSeg` from Practical 1.)

Finally, `TourLength` should compute the length of the tour. (Remember that we are not counting the edge from `city[numCities - 1]` to `city[0]`.)

Test your new functions to ensure they are working correctly. In devising a sanity check for `TourLength`, it may be useful to know that the sides of the graphics window all have length 500.

Part B [7 marks]

The order in which the user entered the cities specifies an initial tour, which will not in general be very efficient. We shall try to improve it by repeatedly swapping adjacent cities in the array (i.e., by reversing the order in which the salesman visits a pair of adjacent cities). The function `SwapHeuristic` repeatedly suggests pairs of cities to swap. Your job is to write the function `TrySwap` that acts on those suggestions.

The function `TrySwap` should have the header

```
int TrySwap(int i)
```

It should behave as follows. First it should find the length of the current tour. (Refer to Part A.) Then it should should swap the contents of `city[i]` and `city[i + 1]` and recompute the tour length. If the length is shorter, then `TrySwap` should return `true` indicating "success"; otherwise it should reinstate the original order between `city[i]` and `city[i + 1]` and return `false` indicating "failure". It is important that `TrySwap` returns `true` only if the new tour is *strictly* shorter than the old one. That way, the tour length always decreases and the function `SwapHeuristic` must terminate. Otherwise it may be possible for `SwapHeuristic` to go into an infinite loop!

Compile your program by issuing the command `make tour`. Test your new function. How well does the swap heuristic perform? (Use the `TourLength` function to print out the length of the tour.) When you believe your program is working, recompile it using the command `make ftour`. Now run the non-interactive version on the 50-city instance by typing `ftour < data50`. Record the length of the resulting tour in the file `results`. (Just write down the integer part; ignore the fractional part.)

Part C [7 marks]

Again we'll try to obtain a good tour by repeated improvements, but this time using more powerful improvement steps. The function `TwoOptHeuristic` repeatedly nominates a contiguous sequence of cities on the current tour, and proposes that these be visited in reverse order. Your job is to write the function `TryReverse` that acts on those suggestions. The function `TryReverse` should have the header

```
int TryReverse(int i, int j)
```

and should behave as follows. First it should find the length of the current tour. Then it should should reverse the contents of the array `city` between components `city[i]` and `city[j]` inclusive. (Thus `city[i]` will be the old `city[j]`, `city[i + 1]` will be the old `city[j - 1]`, and so on.) Next the tour length is recomputed. If the length is shorter, then `TryReverse` should return `true` indicating "success"; otherwise it should reinstate the original order between `city[i]` and `city[j]` and return `false` indicating "failure". As before, it is important that `TryReverse` returns `true` only if the new tour is *strictly* shorter than the old one.

Test your new function. How well does the "2-Opt Heuristic" (as it is known in the trade) perform? When you are happy that your program is working, recompile it using `make ftour`, and run the new heuristic on the same 50-city instance. Again, record the tour length in the file `results`.

Part D [5 marks]

Write a function

```
void GreedyHeuristic()
```

that implements the greedy heuristic applied to TSP. A reasonable interpretation of "greedy heuristic" in this context is as follows. At the outset only `city[0]` is "visited". At some intermediate step, suppose `city[0]` to `city[i - 1]` have been visited; find the unvisited city nearest to `city[i - 1]`, swap it into location `city[i]`, and regard it as visited. Repeat until all cities are visited. (In other words, at each step the salesman visits the nearest unvisited city from his current location.

Test your new function. How well does the greedy heuristic perform? Run the greedy heuristic on the 50-city data file and record the result in the file `results`.

Part E [4 marks]

Can you devise a method that beats all the above? No holds barred in this part! As in previous parts, record the performance on the 50-city instance in the file `results`. This time you should also provide a high-level description of the method you have used (just a couple of paragraphs).

Checklist

- Electronically submit the program `tour.c` and the results file `results` by the deadline: **2pm**, Monday 29th November (Week 11).