CP coursework

Instructions

This is an assessed coursework in five parts. Each part of the coursework gives details of a program or function we want you to construct. To obtain credit for your work, you will need to submit electronically the most advanced version of your programs.

The deadline for completion and electronic submission of the CP coursework (via the `submit` *command on* DICE*) is* **4pm***, Friday 20th November (Week 9). Unless you inform the Course Organiser of CP about medical or other difficulties, work submitted after the deadline will attract zero credit.*

Assessment

This coursework is the only *assessed coursework* for CP. It is worth 10% of your final grade for CP (the other 90% will come from the exam).

For CPMT students, this coursework is also worth 10% of the final mark; however, the exam contributes only 70% (and the final 20% comes from the project done with the Music Technology department in week 11).

The coursework is marked out-of-100. There are five parts to the coursework, and they differ in value.

- The tasks of Parts A-D relate to the "Travelling Salesman Problem" (TSP), explored through `descartes`. Part A is worth 30 marks, Parts B and C 15 marks each, and Part D 10 marks.
- The tasks of Part E relate to the problem of finding cheap flights (both direct and connecting) from a database of flight times and prices. This is worth 30 marks.

In implementing your solutions, please do not add any extra global variables to the program files, apart from the one that already appear in the template (those being `numCities` and the `city` array in `tour.c`, and `numflights` and the `allflights` array in `travel.c`).

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 or one of the course lecturers.
- It is fine to generally discuss the project with your fellow-students, or get help with debugging. It is also fine to ask Lecturers/demonstrators for general advice. However, you may *never* share code, nor take code from others, in completing the task. *Your submission must be entirely your own work.*

Preparation

This practical has associated template and other files.

To copy the files, go to the course webpage and click on `cwk15.tar`. Your browser should bring up a window listing the different files within the tar directory. Save each of these *into*

the particular directory that you have created for the coursework. When doing parts A-D you will not be able to compile your code unless you have `descartes.h` and `descartes.o` (and later, you will need `fdescartes.o`) in your working directory.

If you are not yet 100% confident working with the `descartes` library, please return to the Labsheet for Lab 4 and make sure you can carry out all tasks in that Lab sheet.

Compilation

Parts A, B, C, D

To compile `tour.c`, you will need to link with the `descartes.o` executable code, by typing

```
gcc -Wall tour.c descartes.o -lSDL -lm
```

in the terminal window. Notice you will be linking *both* to the `SDL library` on which `descartes` depends, and also to the `math` library. If compilation is successful, the executable file `a.out` will be created; and you will be able to run it by typing `./a.out` in the shell window. This compilation produces an interactive version of the program in which you may input data by clicking on the graphics window. This mode is good for testing and debugging.

Once you feel your implementation is correct, you should rename `a.out` to `tour` (say) to have it for further use (in which case, you will later type `./tour` to execute).

Part E

In the case of `travel.c`, you can compile with the usual `gcc` command

```
gcc -Wall travel.c
```

in the terminal window. As usual, the executable (assuming the program compiles successfully) will be saved as `a.out`. You will then be able to run it by typing `./a.out` at the command line. Again, you might like to rename `a.out` to `travel` (or similar) once you have finished writing and debugging your code.

Using data files to test parts B, C, D

For Parts B-D, we have designed some test files of points (`data25`, `data50` and `data75`) to allow you to compare the difference between the heuristics B, C and D. But to use these, you need to take the input *not from the graphics window*, but from the provided data files. To achieve this redirection of input, we need to link to alternative "object code for `descartes` at compile time. This alternative object code has been provided as `fdescartes.o`. Everything else about your program should stay the same. Then to *test with file input*, you must compile as follows:

```
gcc -Wall tour.c fdescartes.o -lSDL -lm
```

You should expect compilation to be successful if it was successful with `descartes.o`. After this new compilation, the executable for *working with file input* will again will be created in `a.out`. It's a good idea to copy this over to `ftour`, say.

Then you can run the program on data file `data50` (for example) you type one of the following at the terminal prompt (depending on whether you have renamed your executable to `ftour` yet):

```
./a.out < data50
./ftour < data50
```

You can similarly run tests with the two other data files, `data25` and `data75`.
Note that:

- Even though you must compile with `fdescartes.o`, everything else about your program should stay the same (the header to be included is still `descartes.h`).
- Even though all *input* to your program is taken from the data file passed-in on execution (eg, with `< data50` above), still any *output* will be sent to the usual stream; that is, `DrawLineSeg` output goes to the graphics window, and output with `printf` goes to the Terminal screen.

  Even though output via `DrawLineSeg` goes to the graphics window, any clicks on this window will be ignored.
- When we redirect input for an execution, *all* input will be taken from the data file. Hence when testing against data files, `main` should be written so that the only input requested from the user is a set of points to be read with `GetPoint`.
- Please make sure you *first* carry out careful debugging using the standard method, before you re-compile with `fdescartes.o` and test on the data files provided. Basic errors will be easier to detect in the standard mode where you enter points by clicking on the graphics window.

You will probably need to recompile your file a few times, as you will want to test the various heuristics of Parts B, C and D. Even better would be to write code in `main` to ask the user which heuristic to apply.

Introduction to Parts A, B, C, D

In Parts A-D, we are going to explore the *Travelling Salesman Problem* (TSP). In this problem, 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 (which can be measured using the `Length` function of `descartes`). In the language of the problem, we assume 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 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 will explore *heuristics* for approximate solutions.

Part A [30 marks]

For Part A will be using the simple geometry library `"descartes.h"`. We will be writing our functions in `tour.c` . 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 work is similar to the final stage of Labsheet 4. 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. We will use the global *array* `city` to store these points.

```
#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`.

Your mission for Part A is to extend the program of `tour.c` by implementing the functions with the following headers:

```
int ReadCities(void);

void DrawTour(void);

double TourLength(void);
```

The work of these functions is similar to the code you wrote for the last task of Lab 4.

Your `ReadCities` function should accept a sequence of mouse clicks from the user (you will use the function `GetPoint` of the `descartes` library) 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` must also maintain the value of the variable `numCities` correctly (so that this is always equal to the number of points so far), 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 values* for the x-coordinate and y-coordinate. 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?

After `ReadCities` terminates with `TRUE`, the array `city` now represents a possible tour, in which the salesman starts at `city[0]`, then visits `city[1]`, `city[2]`, and so on until `city[numCities - 1]`.

The second function, `DrawTour`, should display the current tour stored in the `city` array, 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]`. (you will need to make use of the functions `LineSeg` and `DrawLineSeg` of the `descartes` library).

Finally, the `TourLength` function should compute the length of the tour, by summing the length of all its contributing line segments. (Remember that we are not counting the edge from `city[numCities - 1]` to `city[0]`.)

Important: None of `ReadCities`, `TourLength` or `DrawTour` should print to the screen, nor should they read any input from the Terminal (though `ReadCities` will read points from the graphics window).

Marks breakdown: 10 each for `ReadCities`, `TourLength` and `DrawTour`.

Testing: Test your new functions to ensure they are working correctly. You will want to add a `printf` command to your `main` function to print out the value returned by `TourLength` after the points have all been read by `ReadCities`. In devising a initial sanity check for `TourLength`, make use of the fact that the sides of the graphics window all have length 500.

When you have done enough basic tests to feel your implementations may be working, recompile the program using `fdescartes.o`.

```
gcc -Wall tour.c fdescartes.o -lSDL -lm
```

Now run the non-interactive version on the 50-city instance by typing `./a.out < data50`. The value returned by `TourLength()` at this point should be 11554.332413.

Part B [15 marks]

The order in which the user entered the cities specifies an initial tour, which will not in general be very efficient. In Part B we will write code to improve the tour 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);
```

The function `TrySwap` should behave as follows. First it should find the length of the current tour. 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!

At this stage you might want to compile your program by issuing the command

```
gcc -Wall tour.c descartes.o -lSDL -lm
```

You can test your new function by typing `./a.out` at the command line. How well does the swap heuristic perform? (Use `TourLength` to print out the length of the tour.)

Testing: When you believe your program is working, recompile using `fdescartes.o`

```
gcc -Wall tour.c fdescartes.o -lSDL -lm
```

Now run the non-interactive version on the 50-city instance by typing `./a.out < data50`. At this stage (ie after running `SwapHeuristic` on the data read-in from `data50`), the value returned by `TourLength()` should be 8576.074028.

Part C [15 marks]

In Part C 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);
```

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.

Testing: 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 with `fdescartes.o`, and run the new heuristic against `data50`. Here are two tests you can use for `data50`:

- If we immediately run `TwoOptHeuristic` after reading/drawing `data50`, then after this `TourLength` should return the value 2472.821391 for the improved tour.
- If instead we run `SwapHeuristic` first (on `data50`), and follow this with a call to `TwoOptHeuristic`, then after this `TourLength` should return the value 2497.170922 for the improved tour.

Part D [10 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. *At each step the salesman visits the nearest unvisited city from his current location.*

Testing: How well does the greedy heuristic perform? When you are happy that your program is working, recompile it with `fdescartes.o`, and run the greedy heuristic against `data50`. If the `main` function calls `SwapHeuristic`, `TwoOptHeuristic` and `GreedyHeuristic` in that order, then after these three calls `TourLength` should return the value 2756.842017 for the improved tour. You would get a different value if (say) you ran `GreedyHeuristic` on the initial array for `data50`.

However, note that the value returned by `GreedyHeuristic` will usually not depend on the *order* that the points are presented, but just on the choice of which point is in the *first* cell of the `city` array (for `data50`, the point in the first cell gets changed when we run `TwoOptHeuristic`).

Part E [30 marks]

In Part E we consider the task of searching for flights in a travel database. You are given the template file `travel.c` which contains a collection of type declarations, function prototypes, and also some helper functions. Note that doing this part of the coursework requires understanding of *structured data types*, which will be covered on Monday and Tuesday of week 7.

The functions you must write for this question are cast in terms of a number of non-native datatypes, and the declarations of these relevant datatypes (`bool_t`, `date_t`, `timed_h`, `flights_t` and `pair_t`) are shown below:

```
#define MAXFLIGHTS 1000

typedef enum {FALSE, TRUE} bool_t;

typedef struct {int day, month, year;} date_t;
typedef struct {int hours, mins;} timed_t;

typedef struct {
  char apt[4];
  char dest[4];
  char airline[4];
  date_t date;
  timed_t depart;
  timed_t arrive;
  float cost;
  int seats;
} flight_t;

typedef struct {
  int first, second;
} pair_t;

flight_t allflights[MAXFLIGHTS];
int numflights = 0;
```

The main datatype is `flight_t`, which represents flights in terms of the departure airport `apt`, the destination airport `dest`, the airline `airline` which runs the flight, the `date` of the flight, the departure time `depart` and arrival time `arrive` (these are in 24 hour format), plus the number of remaining seats for sale `seats` and the cost `cost` of 1 seat on that flight.

The `pair_t` datatype is defined as an ordered pair of integers, and is intended to represent pairs of connecting flights in the `allflights` database (and to be set to `-1, -1` when we want to signal that no relevant connecting pair exists).

We will make two assumptions about our collection of flight data:

- All flights in the database are assumed to have the same departure date and arrival date (so our database is assumed not to cover transatlantic flights, etc).
- We assume that the departure time `depart` is represented in the local time of the departure airport, and the arrival time `arrive` in the local time of the arrival airport.

Your tasks for this question will involve implementing some simple functions for reasoning about dates and times, and for searching for flights in the database. For this question, the only built-in libraries you should use are `stdio, stdlib` and `string`; in particular, you should not use the `time` library.

To help you test your implementations, we have provided you with a file `flightList.txt` containing a collection of flights, plus a pre-implemented function `initialiseDB` which will read in the file and initialise the global array `allflights` with the flights in the file (and also will set `numflights` to the total number of flights in the database). The `main` function of your template file `travel.c` contains code to test all of your functions (conditional on the initialisation done by the call to `initialiseDB`). You may want to comment out some of these tests in the early stages of your development, to focus on the pieces of code you have already written.

- Your first task is to implement the function `sameDate` which takes two parameters of type `date_t` as its arguments, and returns `TRUE` if those represent exactly the same date, and `FALSE` otherwise. The function prototype is:

  ```
  bool_t sameDate(date_t date1, date_t date2);
  ```

  The `main` function of `travel.c` contains 3 tests for `sameDate`.

- Your second task is to implement the function `cheapestDirectFlight`, which takes as its parameters two strings `from` and `to`, plus a parameter `date` of type `date_t`. The function should return the index of the cheapest flight in the array `allflights` which flies direct from airport `from` to airport `to` on the date `date` and which has spare seats for sale (it might be the case that there is more than one direct flight with the minimum cost available, in which case either index may be returned). Alternatively, the function should return the value -1 if there is no flight with spare seats from airport `from` to airport `to` on `date`. The function prototype is:

  ```
  int cheapestDirectFlight(char* from, char* to, date_t date);
  ```

  You will find it helpful to use the `strcmp` function from the `string` library, as well as the `sameDate` function from (a).

  The `main` function of `travel.c` contains 3 tests for `cheapestDirectFlight` (conditional on `initialiseDB("flightList.txt")` having been executed successfully).

- We now consider the question of whether two flights `fst, snd` of type `flight_t` are feasible as connecting flights. We say that these two flights are feasible connecting flights if the *destination* airport of `fst` is the same as the *departure* airport of `snd`, if both flights have at least 1 spare seat remaining, and if there are more than 60 mins between the arrival of the first flight and the departure of the second. You are asked to complete the function `connectingPair` conforming to the prototype below, returning `TRUE` if `fst, snd` is a connecting pair of flights, and `FALSE` otherwise:

  ```
  bool_t connectingPair(flight_t fst, flight_t snd);
  ```

  The `main` function of `travel.c` contains 4 tests for `connectingPair` (conditional on `initialiseDB("flightList.txt")` having been executed successfully).

- Finally you are asked to consider the question of finding a *pair of connecting flights* of cheapest possible total cost to travel from departure airport `from` to destination `to` on date `date`. Your program should return the indices of the best connecting pair of flights (in that order) as an object of type `pair_t`, or if there is no pair of connecting flights with spare seats from `from` to `to` on `date`, should return the indices -1 for both flights. You are asked to complete the function `bestTransferOption` according to the prototype below:

  ```
  pair_t bestTransferOption(char* from, char* to, date_t date);
  ```

  It is likely that your implementation will use `strcmp` from the `string` library, plus your own implementations of `sameDate` and `connectingPair`.

  The `main` function of `travel.c` contains 5 tests for `bestTransferOption` (conditional on `initialiseDB("flightList.txt")` having been executed successfully).

Marks breakdown: 5 for `sameDate`, 7 for `cheapestDirectFlight`, 8 for `connectingPair` and 10 for `bestTransferOption`.

Checklist

- Electronically submit the program files `tour.c` and `travel.c` by the deadline: **4pm**, Friday 20th November (Week 9).

  The specific form of the command for submitting your files is

  ```
  submit cp 1 tour.c travel.c
  ```

  You must submit **source code** (the .c files), **not** executable.