Computer Science 324 22 November, 2004
St. George Campus University of Toronto

Homework Assignment #5

Due: Wednesday, 8 December, 2004, by 5pm

Last day of class, so no grace days permitted, and no late assignments accepted.

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked on the newsgroup, by email, or in person.*
Total Marks: There are 101 marks available in this assignment. This assignment represents 10% of the course grade.

Handing in this Assignment

What to hand in on paper: Please use an **unsealed** envelope, attaching the cover page provided to the front. Note that without a properly completed and **signed** cover page, your assignment will not be marked. Put inside:

1. A printout of all your solutions.
2. You must describe the testing strategy that you used for predicates `findPath/3`, `computePrice/2`, `findTrip/5`, and `bestTrip/4`. This should include a table listing the test cases that you designed for your predicate, what output your predicate returned, and an explanation of why the test case and output are significant in verifying the correctness of the predicate. Read the following for a discussion of good testing practises:

   `http://www.cs.toronto.edu/∼gpenn/csc324/software.testing.pdf`

You must hand in this part of the assignment in the CSC324 drop box in the Bahen Computer Lab, BA2220.

What to hand in electronically: In addition to your paper submission, you must submit your code electronically. The predicates (including helper predicates) for each question are to be submitted separately, with the filename as stated in the question. **Important:** Each submitted file must be self-sufficient. I.e., if you use predicates from Question 3 to solve Question 4, they must be included in the submission file for Question 4. To submit these files electronically, use the CDF secure website:
`https://www.cdf.utoronto.ca/students`
or use the CDF **submit** command. Type **man submit** for more information.

Warning: marks will be deducted for incorrect submission. Note that if the code submitted electronically differs from the code submitted on paper, we will only mark the electronically submitted version (if, in such a case, you put comments, etc. only on paper, we will mark the question as if no comments, etc. were provided).

Since we will test your code electronically, you must:

- *make certain that your code runs on CDF*,
- use the exact predicate names and argument(s) (including the order of arguments) specified,
- use the exact file names specified in the questions,
- not load any file in any of your submitted files,
- not display anything but the predicate output (no text messages to the user, fancy formatting, etc. — just what is in the assignment handout).

Marking Questions will be both automarked and inspected manually. Note that we may use a *different* flight network than the one given here to automark your code. Your solutions are expected to work for *any* legal flight network configuration.
Other Information You must read the requirements for code and marking information on the following web page: `http://www.cs.toronto.edu/∼anya/a5guidelines.html` This document constitutes part of the **official requirements** for this assignment.
Clarification Page and Newsgroup Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 5 Clarification page, linked from your section's CSC324 home page. You are also responsible for monitoring the CSC324 newsgroup.

Assignment 5

Due Date: This assignment is due on **Wednesday December 8, 2004 at 5pm.**

No grace days permitted and no late assignments will be accepted.

Please include this cover sheet, when handing in your paper assignment.

Last Name:

First Name:

Email:

CDF Login:

Student Number:

Date and Time you are handing this in:

I understand that collaboration is not allowed. All answers are my own, written in isolation, without help from others. This submission is in accordance with the University of Toronto Code of Behavior on Academic Matters `http://www.artsandscience.utoronto.ca/ofr/calendar/rules.htm#behaviour`

Signature ______________________

Prolog Notational Conventions: Predicate and Mode Spec

We shall use the following notation when *referring* to Prolog predicates: Predicates in Prolog are distinguished by their name and their arity. The notation `name/arity` is therefore used when it is necessary to refer to a predicate unambiguously; e.g. `append/3` specifies the predicate named "`append`" that takes 3 arguments.

Sometimes, we may be interested in specifying how specific predicates are meant to be used. To that end, we shall present a predicate's usage with a *mode spec* which has the form: `name(arg1, ..., argn)` where each `argi` denotes how that argument should be instantiated when a goal to `name/n` is called. `argi` has one of the following forms:

+ArgName This argument should be instantiated to a non-variable term.

-ArgName This argument should be uninstantiated.

?ArgName This argument may or may not be instantiated.

For example `delete(+List,?Elem,?NewList)` states that, when using `delete/3`, the first argument should be instantiated whereas the second and third arguments may or may not be instantiated.

Note that these Prolog notational conventions provide a convenient way to *specify* Prolog predicates and their usage. They do not represent in any way the form of your actual code. E.g., when defining predicate `findPath/3` in Question 3a, do not use `''?Origin''`, `''?Destination''` and `''?Path''` as the arguments in your code.

Language Restrictions

In this assignment, you **may not** use `functor/3, arg/3, =../2, assert/1, retract/1, fail/0, bagof/3, findall/3, or setof/3`. Nevertheless, you **may** use `=/2, \+/1, ;/2, ->/2, !/1`, and parentheses for grouping. If you are in doubt about whether you can use a particular feature of SWI Prolog, please consult the A5 Clarification Page, and if it still isn't clear, consult the newsgroup.

The Flight Scheduling System II

Assignment 4 helped you hone your skills as logic programmers. For your final Prolog assignment, you will write important search tools for the flight management system. This will give you a feel for how you might write a real application in Prolog. First, you will design complex queries about flight networks. Finally, you will plan flight routes for long trips.

Before continuing, we repeat some definitions and notation that were introduced in Assignment 4. Recall the example flight network from Assignment 4, depicted here in Figure 1.

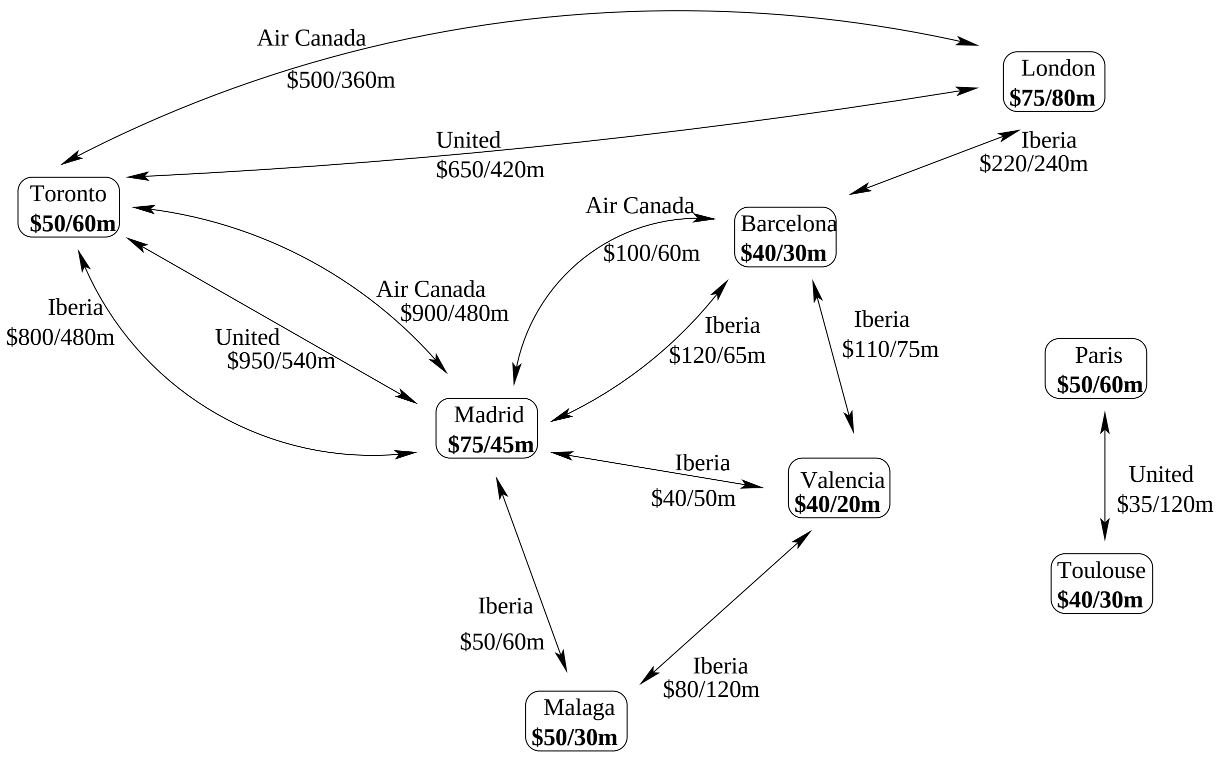

Figure 1: This is the flight network from Assignment 4. Each node denotes an airport-city with its corresponding tax and minimum security delay. Each link denotes a flight and is labelled with its corresponding airline name, price, and duration.

Flight Leg

A *flight leg* is described by a triple list of the form `[City1,Airline,City2]` where `City1` is the city where the leg commences, the *origin*; `Airline` is the airline used for the leg; and `City2` is the city where the leg ends, the *destination* of the leg.

Flight Path

A *flight path* is a possibly empty list of (consecutive) legs. For example, the following list represent a path that starts in Toronto and finishes in Valencia:

```
[[toronto,aircanada,madrid], [madrid,iberia,barcelona], [barcelona,iberia,valencia]] (1)
```

Well-formed Flight Path

A flight path is *well-formed* or *legal* if the destination city of each leg in the path matches the origin city of the subsequent leg in the path. The above path is well-formed but the following one is **not**:

```
[[toronto,aircanada,madrid], [valencia,iberia,barcelona], [barcelona,iberia,valencia]]
```

Note that the empty list `[]` is considered to be a well-formed path.

We can associate a *total price* and a *total duration* with every possible flight path. We may also be interested in the *number of different airlines* that are used by a flight path.

- The total *price* of a flight path, measured in dollars, is the sum of each leg's price, the tax at the initial airport, and the taxes at each airport where there is a change of airline. For example, the total price of path (1) is calculated by adding the price of each of the three legs and the airport taxes at Toronto and Madrid (notice there is no tax charge at Barcelona since there is no change of airline).
- The total *duration* of a path, measured in minutes, is the sum of the duration of each leg plus the security delay at each airport visited (including the initial origin airport and excluding the final destination airport). For example, for the path (1) above, the duration is calculated by adding the duration of the three corresponding legs plus the security delays at Toronto, Madrid, and Barcelona.
- The *number of (different) airlines* is the number of distinct airlines used by the path in question. For example, the above path (1) uses 2 different airlines, namely, Air Canada and Iberia. Observe that the following path also uses only 2 different airlines:
  ```
  [[toronto,iberia,madrid],[madrid,aircanada,barcelona],
   [barcelona,iberia,london],[london,aircanada,toronto]]
  ```

Note that in order to calculate the total price and duration of a path, one needs to query the flight database using both predicates `flight/5` and `airport/3`. On the other hand, the total number of different airlines can simply be obtained by inspecting the flight path in question without using the underlying database.

We now define a complex structure referred to as a *flight route*.

Flight Route

A (detailed) *flight route* is a list of the following form:

```
[Price, Duration, NoAirlines, Path]
```

where `Path` is a *non-empty* and *well-formed* flight path, and `Price`, `Duration` and `NoAirlines` stand for the total price, total duration, and number of different airlines of the flight path in question, respectively. For example, the following list term represents a flight route that has a total price of $1255, a total duration of 755 minutes uses 2 different airlines, and whose actual path is the above path (1):

```
[1255,755,2,[[toronto,aircanada,madrid],[madrid,iberia,barcelona],[barcelona,iberia,valencia]]]
```

Question 1. [1 mark] Fill out the cover sheet correctly.

Question 2. [20 marks] Querying the Database.

Submit your answers to this question on **paper**.

For each of questions (a)-(e) listed below, write a Prolog query (not a predicate!) that will answer the question using the `flight/5`, and `airport/3` predicates with the interface defined in Assignment 4, and any other logic you need. For this task, you may **not** use helper predicates or the exclusions mentioned at the beginning of the assignment handout. When a query asks for multiple answers (e.g., ”What flights ...”, ”What pairs ...”), these answers should be obtained via backtracking by typing “`;`” after each returned answer. Note that your queries must work with any flight system database that adheres to the specification described in A4.

(a) What flights have a duration greater than 3 hours?

(b) What pairs of distinct cities can be connected using exactly two flights with the same airline?

(c) What city can be reached from Toronto in one flight, and with the cheapest ticket?

(d) What is the city with the most expensive airport tax?

(e) What cities cannot be reached from Valencia with just one flight?

Question 3. [30 marks] Finding Flight Routes.

Submit your code in a file called **a5-trip.pl**

Now you are going to write Prolog code that will find paths and routes for our flight system.

a. [10 marks] Write a Prolog predicate

```
findPath(?Origin,?Destination,?Path)
```

that holds if `Path` is a *non-redundant* path from `Origin` to `Destination`. A *non-redundant* path is one that has no loops, that is, it never goes through the same airport city more than once from the origin to the destination.

b. [15 marks] Once you have a path between two cities, you will need to compute its properties, namely, its price, duration and number of different airlines. To that end, write the following three Prolog predicates:

- `computePrice(+Path,?Price)`: it holds iff the Flight Path `Path` has a total price of `Price` dollars.
- `computeDuration(+Path,?Duration)`: it holds iff the Flight Path `Path` has a total duration of `Duration` minutes.
- `computeNoAirlines(+Path,?NoAirlines)`: it holds iff the Flight Path `Path` uses `NoAirlines` different airlines.

c. [5 marks] Finally, you need to put it all together to define a trip. Write a Prolog predicate

```
trip(?Origin,?Destination,?Route)
```

that holds if `Route` is a non-redundant route from `Origin` to `Destination`.
For example, the query

```
?- trip(toronto,barcelona,[Price,Dur,N,Path]).
```

would yield, as one of a number of answers:

```
      Price = 1175
      Dur = 705
      N = 2
      Path = [[toronto,united,madrid],[madrid,aircanada,barcelona]]
```

The query

```
         ?- trip(City,barcelona,Route).
```

would yield, as one of a number of answers:

```
      City = toronto
      Route= [1175, 705, 2, [toronto,united,madrid],[madrid,aircanada,barcelona]]
```

but would also yield as another answer:

```
      City = madrid
      Route = [195, 110, 1, [madrid,iberia,barcelona]]
```

Finally, the queries

```
         ?- trip(toronto,toronto,Route).
         ?- trip(toronto,paris,Route).
```

would just fail and return `No` as the origin and destination are the same and redundant paths are not considered.

Question 4. [50 marks] Finding Optimal Trips.

Submit your code in a file called **a5-optimizing.pl**

The `trip/3` predicate above is useful in finding a route between two cities, but it is of limited practical use because it doesn't discriminate between fast and slow routes or between cheap and expensive trips. In flight planning, we generally want to find the trip that is optimal with respect to some criterion. In our case, we may want to obtain the *fastest* trip or the *least expensive* one.

A naive, brute force approach to finding such a trip is to find all possible paths and then to choose the optimal one with respect to the criterion, by appealing to backtracking (to obtain all possible flight paths) and negation as failure (to verify that there is no better flight path than the one just found). Clearly, this approach is too computationally expensive for large flight networks. In this assignment we require you to develop a more efficient iterative search algorithm by imposing a limit on the criterion – a price or duration cut-off value – for your trip, and then discarding partial solutions as soon as they reach this limit.

Your iterative search algorithm is to be based on the following idea: given an initial route, repetitively apply your algorithm to find "better" routes until you find the optimal one. Rather than the brute force approach described above, your algorithm should stop exploring partial solutions as soon as they reach the criterion limit (i.e., the price or duration cut-off). More specifically, the `trip/3` predicate will yield an initial route. Depending on the designated `Criterion`, the price or duration of this initial route becomes the price or duration limit for your algorithm, which you use to find a second, better route. The price or duration of that route, in turn, becomes the corresponding limit for the next, and so on, until no better route is found, at which point you know that the last route you found is the best one.

a. [35 marks]
Write a Prolog predicate

```
findTrip(?Origin,?Destination,+Criterion,+C_Limit,-Route)
```

that holds iff `Route` is a Flight Route from `Origin` to `Destination` whose `Criterion` (i.e., duration or price) is less than `C_Limit`.

Precondition: `Criterion` is instantiated to either `price` or `duration`; `C_Limit` is instantiated to a number.

You should have one algorithm that works for both criteria. Do **not** write separate algorithms for `price` and `duration`.

b. [15 marks] Write a Prolog predicate

```
bestTrip(?Origin,?Destination,+Criterion,-Route)
```

that holds iff `Route` is *the best* Flight Route from `Origin` to `Destination` with respect to `Criterion`.

Precondition: `Criterion` is instantiated to either `price` or `duration`.

Again, you should have one algorithm that works for both criteria. Do **not** write separate algorithms for `price` and `duration`.

For example, the query

```
?- bestTrip(toronto,barcelona,price,Route).
```

would yield the following unique answer:

```
Route = [665, 530, 2, [[toronto,aircanada,london],[london,iberia,barcelona]]]
```

Also, the query

```
?- bestTrip(toronto,Dest,price,Route).
```

would yield, as two of the total *five* answers:

```
X = london
Route = [550, 420, 1, [[toronto, aircanada, london]]] ;

X = barcelona
Route = [665, 530, 2, [[toronto, aircanada, london], [london, iberia, barcelona]]] ;

...
```

Note: You may use predicates that you defined in Question 3 in your solution to Question 4. If you do so, you must include these predicates in your Question 4 submission file. Note that we may test your Question 4 code with our own Question 3 predicates, to avoid penalizing you twice for errors you may have in your Question 3 predicates.