Computer Science 324 — 31 March, 2007
St. George Campus — University of Toronto

Homework Assignment #6
Due: Friday, 13 April, 2007, by 11:59 PM
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 100 marks available in this assignment. This assignment represents 10% of the course grade.

Handing in this Assignment
What to hand in on paper: No paper submission is required for this assignment.

What to hand in electronically: 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`

Warning: marks will be deducted for an incorrect submission.

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.

Clarification Page and Newsgroup Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 6 Clarification page, linked from your section's CSC324 home page. You are also responsible for monitoring the CSC324 newsgroup.

Do's and Don'ts

1. You should not use any special Prolog/SWI-Prolog features like `assert, retract, arg, ...` Put differently, the only predicates you may use apart from the ones you implement yourself are

   `not/1, append/3, member/2, length/2, 'is',`
   arithmetic symbols (`'='`, `'<'`, `'>'`, `'>='`, `'=<'`, `'+'`, `'-'`, `'*'`, `'/'`),
   the symbols `'\='`, `'->'`, `'!'`, `';'`, and of course `','`.

   Of course you may also use output predicates like `writeln/1` for testing and debugging, but these have to be removed prior to submission. If there is another built-in predicate you wish to use, please consult the newsgroup before using it.

2. Avoid *singleton* variables! A singleton variable is a variable that only occurs once in the arguments or the body of a predicate and is thus useless. For example in the following two definitions:

   ```
   doit( X, Y, Z) :- Y is X*2.
   doitagain( X, Y ) :- Z = doesitmatter, Y is X*2.
   ```

   `Z` is a singleton variable and should be prefixed by an underscore (`_Z`) or removed entirely if possible. Note that it usually won't be possible to simply remove a singleton argument as the arity of the predicate is often fixed:

   ```
   times( [], [], Z) :- !.
   times( [H1|L1], [H2|L2], Z ) :- H2 is H1*Z, times( L1, L2, Z).
   ```

   Here in the first definition `Z` is singleton but cannot be removed. Therefore we should replace it with '`_Z`' or just '_'.

   Although not harmful, it is useful not to have any singleton variables to keep the code easy. SWI-Prolog will point out any singleton variables you have. This is very useful information because it often helps you finding typos, a common source of bugs in Prolog. For instance if we want to increase a number we could write:

   ```
   inc( FirstNr, Result ) :- Result is Firstnr+1.
   ```

   Here both `FirstNr` and `Firstnr` are singletons and SWI-Prolog will tell you so, which in turn will make you realize that you have a typo (`Firstnr` should be spelled with a capital 'N').

   Marks will be deducted for each singleton you have in your code!

The Flight Scheduling System II

Assignment 5 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 5. Recall the example flight network from Assignment 5, depicted here in Figure 1.

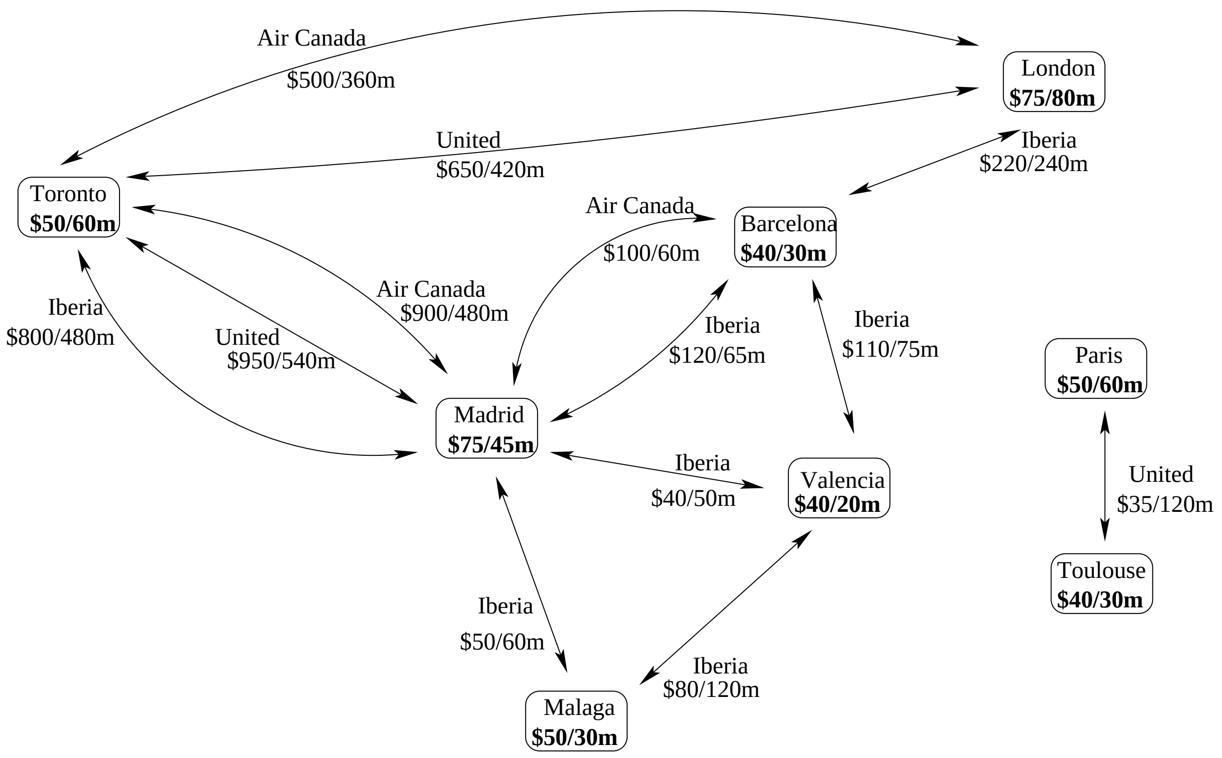

Figure 1: This is the flight network from Assignment 5. Each node denotes an airport-city with its corresponding tax and minimum security delay. Each link denotes a flight and is labelled with its corresponding airline name, price, and duration.

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 $1200, a total duration of 750 minutes uses 2 different airlines, and whose actual path is the above path (1):

```
[1200,750,2,[[toronto,aircanada,madrid],[madrid,iberia,barcelona],[barcelona,iberia,valencia]]]
```

Question 1. [20 marks] Querying the Database.

Submit your answers to this question in a plain text file called **a6-q1.txt**.

For each of questions (a)-(e) listed below, write a Prolog query (not a rule!) that will answer the question using the `flight/5` and `airport/3` predicates, and any other logic you need. For this task, you may **not** use helper predicates. 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 Assignment 5.

(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 2. [30 marks] Finding Flight Routes.

Submit your code in a file called **a6-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 iff `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 iff `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 3. [50 marks] Finding Optimal Trips.

Submit your code in a file called **a6-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 = [845, 740, 2, [[toronto,aircanada,london],[london,iberia,barcelona]]]
```

Also, the query

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

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

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

X = barcelona
Route = [845, 740, 2, [[toronto, aircanada, london], [london, iberia, barcelona]]] ;

...
```

Note: You may use predicates that you defined in Question 2 in your solution to Question 3. If you do so, you must include these predicates in your Question 3 submission file. Note that we may test your Question 3 code with our own Question 2 predicates, to avoid penalizing you twice for errors you may have in your Question 2 predicates.

Clarifications

All questions

- you may reuse predicates defined in A5.
- You may assume the existence of a database `flight/5` and `airport/3` as described in A5. You do not know what the database is, just its format. Your predicates must work with any well-defined database.
- To make sure you can test your code well, we are providing you with the Prolog clauses implementing the A5 flight database. It's accessible from a password protected web site linked to the assignment Web page.
- You may not copy/load the database into any of the files you submit. You can do so temporarily, for testing purposes, but make sure to remove it when you submit. Marks will be deducted for incorrect submissions.

Question 1

duplicate answers In general, you should eliminate duplicates whenever you can. In some cases you cannot eliminate all duplicates, and that's OK. In each query, you should make a decision as to whether or not it is possible to eliminate duplicates. Sometimes there are different causes of duplicates, and you can deal with one but not the other. It is part of the question to figure it out. You don't have to provide any explanations, though.

parts c and d Since the question asks for ”city”, and not ”cities”, in case of a tie, any of the correct answers is acceptable. Either return one solution or return all solutions, one at a time, by pressing ”;” repeatedly.

Question 2

trip is meant to implement a brute force search technique for finding a route between an origin and a destination.

Question 3

Which algorithm? In solving Question 3a, you should not be calling trip to find a route, and then accepting or rejecting it based on whether it adheres to the `C_Limit`. This is inefficient. Instead, you should implement a search technique that stops constructing a particular route when it reaches `C_Limit`.

Most of the work in question 3 is done in 3a. Part 3b, though not short, is simply a wrapper that uses results from 3a and 2 to construct `bestTrip`, following the description at the beginning of question 3.

One Algorithm! You should not be repeating code. You should not be writing two separate rules, which are very similar, except for a line or two, where the difference arises because of the difference in the argument: whether it is price or duration. Instead, you should have one piece of ”main logic”, and the different cases should be handles in a helper method, for example.

Well-formed path and non-redundant path

Recall the following:

- the empty list is considered to be a well-formed path, and of course it is achieve when `origin=destination`.
- a non-redundant path has no loops, i.e., it never goes through the same airport city more than once from origin to destination.

These can work at odds with each other in question 2 and 3. As a consequence, we will accept non-redundant paths where the origin and destination are the same.

E.g., findPath(toronto,toronto,Path) may either succeed or fail. Both answers are acceptable. trip(toronto,toronto,Route) may either succeed or fail. Both answers are acceptable.

Coding Guidelines

1. Comments

   - Each predicate you write, except the ones specified in the assignment handout, should have a concise and clear documentation also indicating its mode of usage, i.e. mark the arguments using +/-/?.
   - You can find plenty of good documentation examples in the SWI help system. For instance, try `'help(append).'` to get the following documentation for the `append/3` predicate:

     ```
     append(?List1, ?List2, ?List3)
     Succeeds  when List3  unifies with  the concatenation  of List1  and
     List2.   The  predicate can be  used with any instantiation  pattern
     (even three variables).
     ```

   - Presumably, most of your predicates will have modes using '+' and '-' only, that is having a strict distinction between input and output arguments. In these cases, clearly specify the pre-conditions for the inputs and post-conditions for the outputs.
   - Do not use your predicates in a way conflicting with your mode documentation. Also, when marking something with '?', the text has to explain the allowed instantiation patterns.

2. Indentation

   - As always, your code should be easy to read. Putting every subgoal on a new line is a good idea.

3. Efficiency

   - You don't have to go out of your way for optimization purposes. However, the code must be reasonable. You should not be traversing the lists unnecessarily, for example.

4. Style

   - simpler, shorter code receives more marks,
   - try to use unification variables when possible; for example:
     `swap(List1, List2) :- List1=[A,B], List2=[B,A].` should instead be `swap([A,B], [B,A]).`,
   - you should pick intuitive names for your predicates and arguments,
   - a good naming convention for a helper predicate used by a predicate `pred`, is `pred_aux` ('aux' as 'auxiliary'). This makes it easier to understand what it is used for.