Practical Class 5

Obtaining Multiple Solutions, Graphs and Search

Objectives:
- Obtaining Multiple Solutions
- Search in Graphs
- Applications in Games and Puzzles

1. Family Relations Reloaded

Consider exercise 1 from the first exercise sheet, regarding family relations.

a) Implement *children(+Person, -Children)*, which returns in the second argument a list with the children of *Person*.

b) Implement *children_of(+ListOfPeople, -ListOfPairs)*, which returns in the second argument a list with pairs in the format *P-C*, where *P* is an element of *ListOfPeople* and *C* is a list containing their children.

c) Implement *family(-F)*, which returns a list with all the people in the family.

d) Implement *couple(?C)*, which unifies *C* with a couple of people (in the format *X-Y*) who have at least one child in common. Example:

```
| ?- couple(phil-claire).
yes
| ?- couple(C).
C = dede-jay ?
```

e) Implement *couples(-List)*, which returns a list of all couples with children, avoiding duplicate results.

f) Implement *spouse_children(+Person, -SC)*, which returns in *SC* a pair *Spouse/Children* with a spouse of *Person*, and the children they have together.

g) Implement *immediate_family(+Person, -PC)*, which returns in *PC* a pair *A-B*, where *A* is a list with the parents of *Person* and *B* is a list with the spouses and respective children. Example:

```
| ?- immediate_family(haley, X).
X = [phil,claire]-[dylan/[george,poppy]] ?
```

h) Implement *parents_of_two(-Parents)*, which returns in *Parents* the list of people who have at least two children.

2. Teachers and Students Reloaded

Consider exercise 2 from the first exercise sheet, about teachers and students.

a) Implement *teachers(-T)* which returns a list with all the teachers.

b) How does the predicate implemented above behave in case a teacher teaches more than one course? How can you prevent duplicates?

c) Implement *students_of(+T, -S)* which returns a list with all the students of professor T.

d) Implement *teachers_of(+S, -T)* which returns a list with all teachers of student S.

e) Implement common_courses(+S1, +S2, -C), which returns a list of all courses attended by both student S1 and S2.

f) Implement *more_than_one_course(-L)*, which returns a list of all students attending more than one course. Note: avoid duplicate elements.

g) Implement *strangers(-L)*, which returns a list with all pairs of students who don't know each other, i.e., don't attend any course in common.

h) Implement *good_groups(-L),* which returns a list with all the students who attend more than one course in common.

3. Schedules

Consider the following code, representative of the schedules of a L.EIC class:

```
%class(Course, ClassType, DayOfWeek, Time, Duration)
class(pfl, t, '2 Tue', 15, 2).          class(ipc, tp, '4 Thu', 16, 1.5).
class(pfl, tp, '2 Tue', 10.5, 2).       class(fsi, t, '1 Mon', 10.5, 2).
class(lbaw, t, '3 Wed', 10.5, 2).       class(fsi, tp, '5 Fri', 8.5, 2).
class(lbaw, tp, '3 Wed', 8.5, 2).       class(rc, t, '5 Fri', 10.5, 2).
class(ipc, t, '4 Thu', 14.5, 1.5).      class(rc, tp, '1 Mon', 8.5, 2).
```

a) Implement *same_day(+Course1, +Course2)*, which succeeds if there are classes of both Course1 and Course2 taking place on the same day.

b) Implement *daily_courses(+Day, -Courses)*, which receives a day of the week and returns a list with all the courses taking place on that day.

c) Implement *short_classes(-L)*, which returns in *L* a list of all classes with a duration of less than 2 hours (list of terms in the format *UC-Day/Time*).

d) Implement *course_classes(+Course, -Classes)*, which receives a course and returns a list of all the classes from that course (list of terms in the format *Day/Time-Type*).

e) Implement *courses(-L)*, which returns a list of all existing courses. Avoid repeated results.

f) Implement *schedule/0*, which prints all the classes in the terminal, in the order by which they take place during the week.

g) Modify the previous predicates so that the day of the week is printed only as Mon, Tue, Wed, Thu or Fri. Tip: use a 'translation' predicate to convert between the internal representation format and the display format.

h) Implement *find_class/0*, which asks the user for a day and time, and indicates whether a class starts or is taking place at that time, printing the class, start time and duration. If no class is taking place, the predicate should display a message stating that.

4. Flights

Consider the following code excerpt representing existing flights:

```
%flight(origin, destination, company, code, hour, duration)
flight(porto, lisbon, tap, tp1949, 1615, 60).
flight(lisbon, madrid, tap, tp1018, 1805, 75).
flight(lisbon, paris, tap, tp440, 1810, 150).
flight(lisbon, london, tap, tp1366, 1955, 165).
flight(london, lisbon, tap, tp1361, 1630, 160).
flight(porto, madrid, iberia, ib3095, 1640, 80).
flight(madrid, porto, iberia, ib3094, 1545, 80).
flight(madrid, lisbon, iberia, ib3106, 1945, 80).
flight(madrid, paris, iberia, ib3444, 1640, 125).
```

```
flight(madrid, london, iberia, ib3166, 1550, 145).
flight(london, madrid, iberia, ib3163, 1030, 140).
flight(porto, frankfurt, lufthansa, lh1177, 1230, 165).
```

a) Implement *get_all_nodes(-ListOfAirports)*, which returns a list of all the airports served by the flights in the database, with no duplicates.

b) Implement *most_diversified(-Company)*, which returns the company with the most diversified destinations, i.e., largest number of cities it flies from/to (if more than one exist, new results should be returned via backtracking).

c) Implement *find_flights(+Origin, +Destination, -Flights)*, which returns in *Flights* a list with one or more flights (their codes) connecting *Origin* to *Destination*. Use depth-first search, avoiding cycles.

d) Implement *find_flights_bfs(+Origin, +Destination, -Flights)*, with the same meaning as before, but now using breadth-first search.

e) Implement *find_all_flights (+Origin, +Destination, -ListOfFlights)*, which returns in *ListOfFlights* a list of all the possible ways to connect *Origin* to *Destination* (each represented as a list of flight codes).

f) Implement *find_flights_least_stops(+Origin, +Destination, -ListOfFlights)*, which returns in *ListOfFlights* a list of all possible ways to connect *Origin* to *Destination* (each represented as a list of flight codes) with a minimum number of stops, i.e. a list of the shortest paths between *Origin* and *Destination*.

g) Implement *find_flights_stops(+Origin, +Destination, +Stops, -ListFlights)*, which returns in *ListFlights* a list of possible ways to connect *Origin* to *Destination* (each represented as a list of flight codes) that stop in the cities indicated in *Stops*.

h) Implement *find_circular_trip (+MaxSize, +Origin, -Cycle)*, which returns in *Cycle* a list, of maximum length *MaxSize*, with flights that start and end in *Origin*, forming a cycle.

i) Implement *find_circular_trips(+MaxSize, +Origin, -Cycles)*, which returns in *Cycles* a list of cycles of maximum length *MaxSize*, starting and ending at *Origin*.

j) A graph or subgraph is strongly connected if from any of its vertices you can reach any of the other vertices (directly or indirectly). Implement *strongly_connected(+ListOfNodes),* which succeeds if the set of nodes received as an argument (which belongs to the flight graph) constitutes a strongly connected subgraph, failing otherwise.

k) Implement *strongly_connected_components(-Components)*, which determines the maximum strongly connected components (it is not possible to add a node/branch to the component without it ceasing to be strongly connected) of the flight graph.

l) A bridge is an edge that is not contained in any cycle of a graph. Implement *bridges(-ListOfBridges)*, which returns all the flights that constitute a bridge in the flight graph.

5. Unification

Implement *unifiable(+L1, +Term, -L2)*, where *L2* is a list containing all elements from list *L1* that can be unified with *Term*. However, the elements in *L2* are not unified with *Term* itself. Example:

```
| ?- unifiable([X, b, t(Y)], t(a), L).
L = [X, t(Y)]
```

Note that if *Term1* and *Term2* are unifiable, then *not(Term1 = Term2)* fails, and any instantiation of *Term1 = Term2* is nullified.

6. Missionaries and cannibals

Three missionaries and three cannibals want to cross from the left bank of the river to the right bank, and only one boat with a capacity of 2 people is available to make the crossing. Implement *missionaries_and_cannibals(-Moves)* to determine the movements to be made in order to make the crossing, knowing that the number of cannibals cannot be greater than the number of missionaries on either bank of the river at any given time.

7. Climbing stairs

Claude always climbs the stairs to his house in a different way, one or two steps at a time. Implement *steps(+Steps, -N, -L)*, which receives the number of steps on the stairs, and returns in *N* the number of ways to climb them, and in *L* the list with all the possibilities (each possibility being a list with sequences of 1s and 2s).

8. Sliding puzzle

Implement *sliding_puzzle(+Initial, -Moves)*, which receives the initial layout of a sliding puzzle (represented as a list of lists of size NxN, where the pieces are represented by a number from 1 to N^2-1, and the empty cell by 0), and returns a list with a possible sequence of moves that allows the puzzle to be solved. The possible movements are swiping up, down, left or right. Implement both a depth-first solution and a breadth-first one, and compare the solutions obtained by each approach.