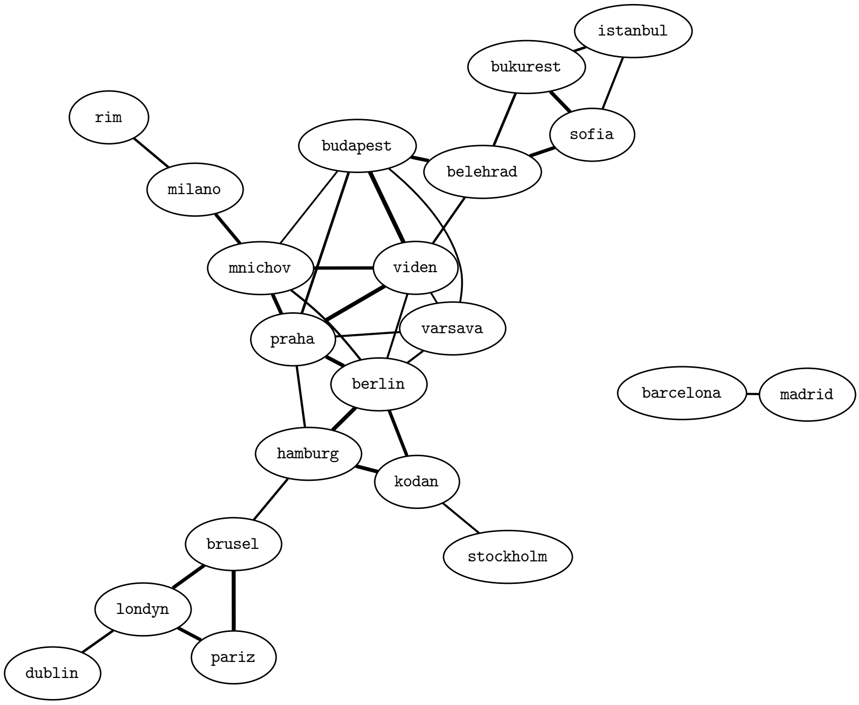
4. tutorial in Prolog

April 26, 2016

Task 1: Load the map of Europe into Prolog. The `europe` predicate connects european capitals that are closer than 600 km (flight distance).



```
europe(belehrad, bukurest, 447).
europe(belehrad, budapest, 316).
europe(berlin, kodan, 354).
europe(hamburg, kodan, 287).
europe(berlin, hamburg, 254).
europe(brusel, hamburg, 489).
europe(bukurest, istanbul, 445).
europe(brusel, londyn, 318).
europe(dublin, londyn, 462).
europe(barcelona, madrid, 504).
europe(berlin, mnichov, 501).
europe(budapest, mnichov, 563).
europe(milano, mnichov, 348).
europe(brusel, pariz, 261).
europe(londyn, pariz, 340).
europe(berlin, praha, 280).
europe(budapest, praha, 443).
europe(hamburg, praha, 492).
europe(mnichov, praha, 300).
europe(milano, rim, 476).
europe(belehrad, sofia, 329).
europe(bukurest, sofia, 296).
europe(istanbul, sofia, 502).
europe(kodan, stockholm, 521).
europe(belehrad, viden, 489).
europe(berlin, viden, 523).
europe(budapest, viden, 216).
europe(mnichov, viden, 354).
europe(praha, viden, 250).
europe(varsava, viden, 557).
europe(berlin, varsava, 516).
europe(budapest, varsava, 545).
europe(praha, varsava, 514).
```

Task 2: Implement a depth-first-search procedure with the properties:

1. It should connect any two cities, e.g. both `barcelona` to `madrid` and `madrid` to `barcelona`. Please note that the provided map only connects cities in alphabetic order (`barcelona` to `madrid`, but not `madrid` to `barcelona`).

2. The procedure is finite, it never ends in an infinite loop. You can try it by *disproving* a path between `barcelona` and `stockholm`. Implementing a closed-list is a good idea.

3. It returns the list of visited cities and the total journey length.

Check your result: How did you implement the visited cities? Do you get them in forward or reverse order? Did you use the accumulator in one of them? Ideally, you should implement them both to see the comparison:

```
?- dfs(dublin,stockholm,Journey,Reverse,Length,[]).
Journey = [kodan, hamburg, brusel, pariz, londyn],
Reverse = [londyn, pariz, brusel, hamburg, kodan],
Length = 2360 ;
...
```

Task 3: Study the `findall` meta-predicate, which finds all `berlin`'s neighbours:

```
?- findall(Dest, europe(berlin, Dest,_), Neighs).
Neighs = [kodan, hamburg, mnichov, praha, viden, varsava].
```

Task 4: Using `findall`, find the longest journey in the map. It should be 7156 km long.

Tip: Find lengths of all journeys using `findall` and your `dfs` predicate. Then sort the list using the built-in `sort(List,SortedList)`.

Task 5: Verify that the longest journey connects `istanbul` and `dublin`.

Tip: Members of the list in `findall` do not have to be simple constants! Try the following code:

```
?- findall(my_functor(Len,Dest),
           europe(berlin,Dest,Len),
           Neighbours).
```

Task 6: Implement all previous steps using the bredth-first-search procedure.

Note that in DFS, you don't implement the open-list (Prolog keeps the open-list on the stack automatically). For BFS, you will have to implement the open-list as a separate argument.

Task 7 (optional): Reimplement the closed list using red-black trees. Don't worry, there is a built-in library in SWI Prolog:

`http://www.swi-prolog.org/pldoc/doc/swi/library/rbtrees.pl`

If you replace the *insert* and *member* operations, their runtime drops from from $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$.

Task 8 (optional): Reimplement the open-list in DFS using difference-lists. It is a normal list except that at very end of it, instead of `[]` there is a logic variable, paired with that variable: `dl([a,b,c|E], E)`

A common example to explain why they are useful is:

```
append_dl(dl(I,M), dl(M,O), dl(I,O)).
```

Yes! They can do `append` in constant $\mathcal{O}(1)$ time!

One more tip: Verifying emptiness is done as follows:

```
empty_dl(dl(E,F)) :- E == F.
```