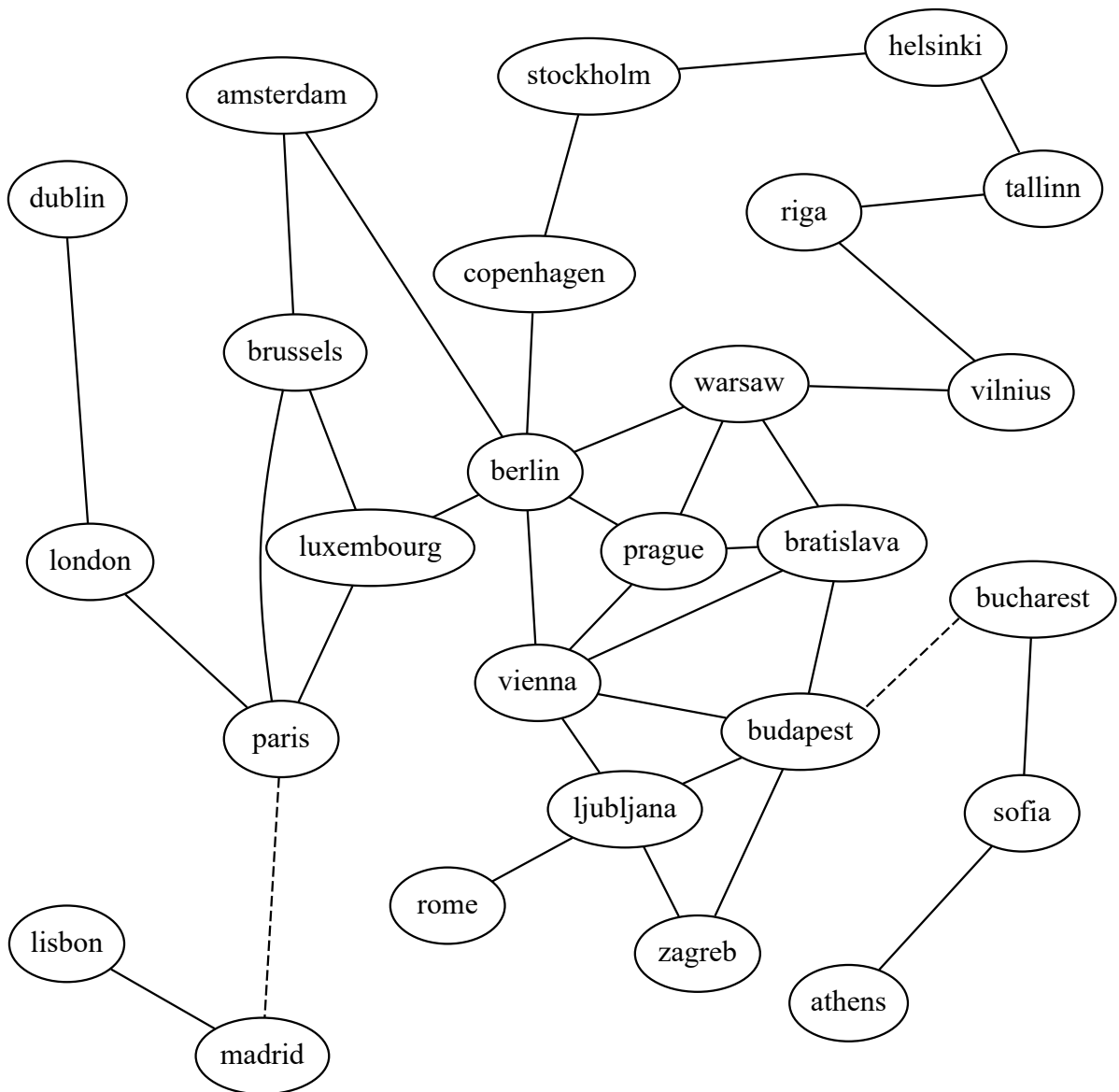


Logical reasoning and programming, lab session 12
(December 15, 2025)

Task 1: Load the map below, represented by the `europa/3` predicate, into Prolog. The map shows most of the EU capitals and London. Two capitals are connected if their respective countries are neighbours or if they are connected by a car ferry and their flight distance does not exceed 600km.

Note the two dashed lines (commented lines in the code). Those connections are longer than 600km and are thus removed from the program, which causes our graph to be disconnected. All your solutions should account for that fact, although you may include the connections when debugging.



```

europe(lisbon, madrid, 503).
% europe(madrid, paris, 1053).
europe(dublin, london, 464).
europe(london, paris, 342).
europe(paris, brussels, 263).
europe(paris, luxembourg, 287).
europe(brussels, luxembourg, 187).
europe(brussels, amsterdam, 174).
europe(luxembourg, berlin, 600).
europe(amsterdam, berlin, 577).
europe(berlin, copenhagen, 355).
europe(berlin, warsaw, 516).
europe(berlin, prague, 280).
europe(berlin, vienna, 523).
europe(copenhagen, stockholm, 521).
europe(stockholm, helsinki, 396).
europe(helsinki, tallinn, 82).
europe(tallinn, riga, 280).
europe(riga, vilnius, 261).
europe(vilnius, warsaw, 393).
europe(prague, warsaw, 514).
europe(prague, bratislava, 291).
europe(prague, vienna, 250).
europe(bratislava, warsaw, 533).
europe(bratislava, budapest, 161).
europe(bratislava, vienna, 55).
europe(vienna, budapest, 214).
europe(vienna, ljubljana, 279).
europe(ljubljana, rome, 490).
europe(ljubljana, zagreb, 117).
europe(ljubljana, budapest, 380).
europe(budapest, zagreb, 299).
% europe(budapest, bucharest, 643).
europe(bucharest, sofia, 296).
europe(sofia, athens, 526).

```

```

way(X, Y, Dist) :- europe(X, Y, Dist).
way(X, Y, Dist) :- europe(Y, X, Dist).

```

Task 2: Implement a depth-first-search (DFS) procedure with properties:

1. It should connect any two cities, e.g. both `lisbon` to `madrid` and `madrid` to `lisbon`. Please note that the provided map only connects cities in one direction. Use the `way/3` predicate!
2. The procedure is finite, it never ends in an infinite loop. You can try it by *disproving* a path between `lisbon` 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,berlin,Journey,Reverse,Length).
Journey = [dublin, london, paris, brussels, luxembourg, berlin],
Reverse = [berlin, luxembourg, brussels, paris, london, dublin],
Length = 1856 ;
Journey = [dublin, london, paris, brussels, amsterdam, berlin],
Reverse = [berlin, amsterdam, brussels, paris, london, dublin],
Length = 1820 ;
...
```

```
dfs(From, To, Journey, Reverse, Length):-
    dfs_with_cl(From, To, Journey, Reverse, Length, [From], [From]).

dfs_with_cl(To, To, [To], Acc, 0, _, Acc).
dfs_with_cl(From, To, [From|Journey], Reverse, Length, Cl, Acc):-
    way(From, X, Len1),
    not(member(X, Cl)),
    dfs_with_cl(X, To, Journey, Reverse, Len2, [X|Cl], [X|Acc]),
    Length is Len1 + Len2.
```

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

```
?- findall(Dest, europe(berlin, Dest, _), Neighbours).
Neighbours = [copenhagen, warsaw, prague, vienna].
```

You may refer to the documentation.

Task 4: Using `findall/3`, find the longest path in the map. It should be 6513km long.

Hint: Find lengths of all journeys using `findall/3` and your `dfs` predicate. Then extract its largest element using the built-in `max_member` predicate.

```
findall(D, dfs(rome, dublin, _, _, D), Bag), max_member(X, Bag).
```

Next step: Verify that the longest journey connects `dublin` and `rome`.

Hint: 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), List).
```

Task 5: Implement the breadth-first-search (BFS) procedure.

If you don't know how to start, here are some suggestions:

1. Define the current state using a structured term `s(CurrentNode, PathSoFar)`. For example, starting in Dublin, the currently explored city can be `s(brussels, [paris, london, dublin])`.
2. Note that in DFS, you don't implement the *open list* (Prolog keeps it on the stack automatically). For BFS, you will have to implement it as a separate argument.

3. Initialize the open list to 1 city, where the journey starts.
4. In every recursive call, merely *pop* the first city from the open list, find its neighbours using `findall/3` and append them to the *end* of the open list.
5. Please note that appending `findall`'s result before other items in the open list gives you a DFS procedure. Try it!

```

bfs_call(From, To, Path):-
    bfs(To, [s(From, [])], Path).

bfs(To, [s(To, Path)|_], [To|Path]).
bfs(To, [s(To, _)|OpenList], Path):-
    !,
    bfs(To, OpenList, Path) .
bfs(To, [s(CurrentNode, CurrentPath)|OpenList], Path):-
    member(CurrentNode, CurrentPath),
    !,
    bfs(To, OpenList, Path).
bfs(To, [s(CurrentNode, CurrentPath)|OpenList], Path):-
    findall(s(X, [CurrentNode|CurrentPath]),
           way(CurrentNode, X, _), Bag),
    append(OpenList, Bag, NewOpenList),
    bfs(To, NewOpenList, Path).

```

Task 6 (optional): Reimplement the *closed list* using red-black trees. Don't worry, there is a built-in library in SWI Prolog:

`https://eu.swi-prolog.org/pldoc/doc/_SWI_/library/rbtrees.pl`

Runtime of the *member* operation will drop from $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$. However, *insert* will also take $\mathcal{O}(\log(n))$, whereas without an RB tree, it can be done in $\mathcal{O}(1)$ time.

```

dfs_2(From, To, Journey, Reverse, Length):-
    rb_empty(Tree),
    dfs_with_rb(From, To, Journey, Reverse, Length, Tree, [From]).

dfs_with_rb(To, To, [To], Acc, 0, _, Acc).
dfs_with_rb(From, To, [From|Journey], Reverse, Length, Rb, Acc):-
    way(From, X, Len1),
    not(rb_lookup(X, X, Rb)),
    rb_insert(Rb, X, X, NewRb),
    dfs_with_rb(X, To, Journey, Reverse, Len2, NewRb, [X|Acc]),
    Length is Len1 + Len2.

```

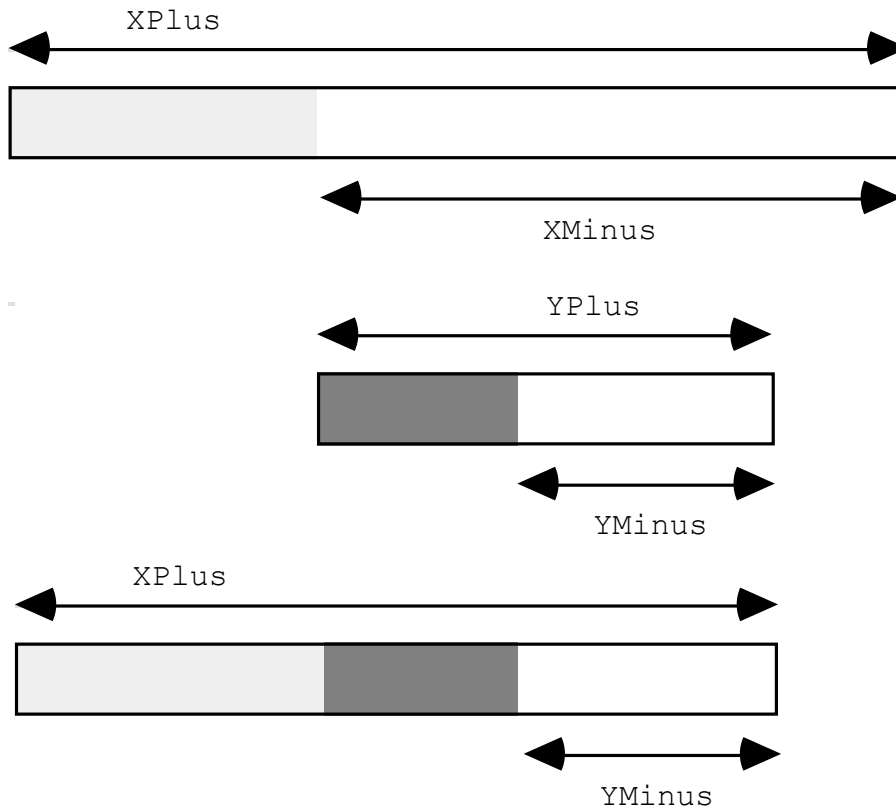
Task 7 (optional): Reimplement the *open list* in BFS using a difference list. Difference list represents a regular list using two lists, namely a *plus list* and a *minus list*, such as

```
[a,b,c,d,e]-[d,e] = [a,b,c]
```

To see the advantage, consider the following two difference lists in Prolog:

```
L1 = [a,b,c|X]-X
L2 = [d,e|Y]-Y
```

How could we append L2 to L1 using Prolog's unification procedure?



```
append_dl(XPlus-XMinus, YPlus-YMinus, XPlus-YMinus) :- XMinus = YPlus.
% or shorter
append_dl(XPlus-YPlus, YPlus-YMinus, XPlus-YMinus).
```

Can you see it? Difference lists can do **append** in constant $\mathcal{O}(1)$ time! (At the cost of increasing memory requirements for the unification procedure.)

Note: The “-” sign here is just a regular functor in Prolog. We could have used e.g. `dl([a,b,c,X], X)` just as efficiently!

Hint: Check for emptiness by

```
empty_dl(X-Y) :- X == Y.
```