

# Project 2: Prolog Train Routing

---

*Due:* Friday, 6 December at 11:59 pm

**No grace days allowed—no late assignments will be accepted.**

*Weight:* 10% of your course grade.

For this assignment, you will write an important component of a railway management system: route finding. First, you will need to write a database describing the railway network. You will then use this database to ask some simple queries about the network. Finally, you will use the database to plan train routes for long trips, both for freight trains and for passenger trains.

In this assignment, you may *not* use `functor`, `arg`, `=..`, `assert`, `retract`, or `fail` (which we will most likely not talk about in class anyway). However, you may use `=`, `\+`, `;`, and parentheses for grouping. You may also find the `->` operator useful and you are encouraged to look it up. It is documented in the XSB manual (see the link on the web page), for example.

## Database

---

A database can be naturally represented in Prolog as a set of facts. In this assignment, information about a railway system can be represented so that each railway link is described by one clause. Also, information about the stop-over time at each city can also be represented by one clause.

The file `p2.data` available on the course web page lists the links that exist in our railway network, for which you will create a database. Each line in the file represents one link, and contains the name of the two cities connected by the link, the speed limit on that segment in km/h, and the length of the link (the distance between the two cities). Each link can be traversed by trains in either direction. Note that this network information is only loosely based on our regional railway system, and is not intended to be realistic. For each city on the network, the file also shows the minimum stop-over time for freight and passenger trains.

The link database for this railway network can be stored in Prolog using a predicate `link`, and possibly a helper predicate to avoid writing redundant information. The stop-over data can be stored in Prolog using a predicate `city`.

**Task 1:** Create a Prolog database to hold all the information in file `p2.data`.

Required interface: the query

```
?- link(From, To, SpeedLimit, Length).
```

must produce all the possible links in both directions (although your fact base should only list each link once) with the speed limit in km/h and the length in km; the query

```
?- city(City, MinFreightStopOver, MinPassengerStopOver).
```

must produce all the cities with the stop-over times in minutes.

Train type	Maximum Speed (km/h)
Freight	120
Passenger	200

Table 1: Train Types

Table 1 shows the types of trains we have in our system and the maximum speed at which these trains can travel. The train type information can be added to our database using a predicate `train`.

**Task 2:** Add the train type information to your database.

Required interface: the query

```
?- train(Type,MaxSpeed).
```

must produce both train types with their maximum speed in km/h.

Note: for tasks 1 and 2, you should save your work in file `p2facts.P`.

**Task 3:** For each of the following questions, write a Prolog query (*not a predicate!*) that will answer the question using the `link`, `city`, and `train` predicates with the interface defined above, and any other logic you need, except that you may not use helper predicates or the exclusions mentioned at the beginning of this handout. Not that your queries must still work if we replace your `p2facts.P` file with another describing a different train system with the same interface.

Place all your queries in file `p2queries.in`, including the exact number of lines with `;` that you need to get all the answers from Prolog. (See the sample `.in` file provided with Problem Set 4 for an example of how you can do that. The first command in this file should be `[p2facts].` in order to load `p2facts.P`) Run your queries using the following command on a CDF Linux workstation or server:

```
xsb < p2queries.in > p2queries.out
```

For this question, don't worry if an answer is produced more than once, or if Prolog prints more information than requested by the question.

- a) What links have a speed limit of 100 km/h or less?
- b) What cities are directly connected to Toronto?
- c) Which links can be traversed in less than 2 hours by a freight train?
- d) What pairs of distinct cities can be connected using exactly two links? (Only print the first five answers.)
- e) On which links can a passenger train travel at its maximum speed?
- f) Which city is the closest one to Toronto?
- g) Which cities have the shortest stop-over time for passenger trains?

## Finding Train Routes

---

You are going to write Prolog code that will find ways to plan routes for freight and passenger trains on our railway system.

### Route Path

A route will be described by the list of cities it traverses. For example, if we plan a freight train route from Toronto to Ottawa via Kingston and Brockville, we would write this route `[toronto,kingston,brockville,ottawa]`.

### Route Length

The length of a route is simply the sum of the lengths of the links traversed by this route.

### Route Duration

The duration of a route depends on the speed limit on each link, as well as the maximum speed at which a particular train can travel. A freight train never travels faster than 120 km/h, because it costs too much in fuel to travel any faster, and reducing costs is a key concern for freight trains. Passenger trains, on the other hand, are much lighter and can travel at speeds up to 200 km/h, and the priority is to get the passengers to destination as fast as possible, so the extra cost of traveling at this speed is not much of a consideration.

The speed limit on each link indicates how fast any train can travel on that link without risk of derailling. When you calculate the duration of a route, assume that trains always travel as fast as possible on each link, that is, at the least of their own maximum speed and the link's speed limit.

Thus, the duration of a route is the sum of the time it takes to traverse each link, plus the stop-over time at each city in the path, excluding the end points.

### The route predicate

When planning a route for a particular train, we have an origin city, a destination city, and the type of train we are planning this route for. Given these as input, we will find a route and give its path, length and duration.

**Task 4:** Write a predicate

```
route(Origin, Destination, TrainType, Path, Length, Duration)
```

which holds iff `Path` is a non-redundant path from `Origin` to `Destination` of length `Length` which would take `Duration` hours for a train of type `Train` to traverse.

A non-redundant path is one which has no unnecessary loops, that is, it never goes through the same city more than once from the origin to the destination.

Preconditions: `Length` is either uninstantiated or a number. `Duration` is either uninstantiated or a number.

For example, the query

```
?- route(toronto, montreal, passenger, Path, Length, Duration).
```

would yield, as one of a number of answers:

```

Path = [toronto, northbay, ottawa, cornwall, montreal]
Length = 880
Duration = 6.1333

```

and the query

```
?- route(montreal, montreal, freight, Path, Length, Duration).
```

would yield, as the only possible answer:

```

Path = [montreal]
Length = 0
Duration = 0

```

## Optimizing Routes

---

The `route` predicate above is useful in finding a route between two cities, but it has a severe limitation for actual use, in that it doesn't discriminate between fast and slow routes. In our railway planning, we want to find the fastest route, or at least a reasonably fast route. We could find all the possible paths and choose the fastest one, but that would be computationally much too expensive. Instead, we'll employ an iterative search technique which uses Bounded Depth-First Search (BDFS).

BDFS (sometimes also called Depth-Limited Search) is based on Depth-First Search (DFS), the search strategy used by Prolog, except that in BDFS, we set a limit to the allowable depth of a solution. When a partial solution reaches or exceeds this limit, we discard it immediately. BDFS typically uses the depth of the search tree as the limit, but any limiting criterion can be used instead. In our case, the time required to travel a route, or its duration, is the appropriate criterion since we evaluate a route by its duration, not by the number of links it traverses.

Our iterative search technique is based on the following idea: given a known route, we can repetitively use BDFS to find faster routes until we find the fastest one. The `route` predicate will yield a first known route. Its duration becomes the maximum allowable duration for the BDFS algorithm, which we use to find a second, faster route. The duration of that route becomes the duration limit for the next, and so forth, until no faster route is found, at which point we know that the last route we found was the fastest one.

**Task 5:** Write a predicate

```
bdfsRoute(Origin, Destination, TrainType, DurationLimit, Path, Length, Duration)
```

which works like `route` but only holds if `Duration` is *less than* `DurationLimit`. In order to be efficient, every time it considers adding a link to the path so far, it should test that the duration of the resulting route is still less than `DurationLimit` before proceeding with the search, or backtrack immediately otherwise.

Preconditions: `DurationLimit` is a number. `Length` is either uninstantiated or a number. `Duration` is either uninstantiated or a number.

**Task 6:** Write a predicate

`fastestRoute(Origin, Destination, TrainType, Path, Length, Duration)` which holds if `Path`, `Length` and `Duration` describe the fastest route from `Origin` to `Destination` for a train of type `TrainType`. This predicate should use the iterative search technique described above.

Preconditions: `Origin` and `Destination` are instantiated. `Length` is either uninstantiated or a number. `Duration` is either uninstantiated or a number.

### Important Notes:

- All your predicates should be generic, so that they could be used with databases describing any train system, not just the one you are working with.
- For all the predicates you write, the user must be able to use any combination of variables and constants that doesn't violate the specified preconditions, and your system should simply say "no" (*and not crash!*) if the user makes a nonsensical query.

For example,

```
?- route(A,B,C,3,E,F).
```

should simply say "no", since `Path` should be a list, not a number,

```
?- route(A,B,C,D,[a,b,c],F).
```

should say "no", since the `Length` should be a number, but

```
?- route(A,B,C,D,400,F).
```

should produce all 400 km long routes using any train type, and

```
?- bdfsRoute(A,B,C,1,D,E,F).
```

should produce all possible routes from any city to any other city using any train type, but taking less than 1 hour.

## Testing

---

As usual, you should test your predicates thoroughly, making sure they work with all inputs that don't violate your preconditions.

With your assignment, you must submit a test suite for the `link`, `route`, `bdfsRoute`, and `fastestRoute` predicates. Note that your test suite should be complete even if the code you submit doesn't work completely. For each of these four predicates, you must save your test suite in file `predicate.in`, following the instructions we provided with problem set 4. Run each of your test suites using the following command on a CDF Linux workstation or server:

```
xsb < predicate.in > predicate.out
```

Include brief comments in your test scripts to outline your testing strategy and the purpose of each test case.

Note that, as was the case for project 1, testing will be an important part of your project grade.

## Silent Policy

---

The silent policy will take effect 27 hours before this assignment is due. This means that no question asked after 9pm on Thursday, 5 December will be answered, whether it is asked on the newsgroup, by e-mail or in person.

## Plagiarism will not be tolerated

---

As with all assignments in this course, the work you submit must be **entirely** your own. You may not share your code or design with other students. You may not get ideas for your code by looking at someone else's screen in a CDF lab. If you discuss aspects of the assignment with another student, you may not leave from the discussion with written notes, not even on a napkin. All detected plagiarism cases will be prosecuted.

## Handing It In

---

You must submit this entire project electronically.

- Place your database for tasks 1 and 2, *i.e.*, the `link`, `city` and `train` predicates, in file `p2facts.P`.
- Place all your queries for task 3 in file `p2queries.in`, and the resulting output in file `p2queries.out`.
- Place your code for tasks 4, 5 and 6 in a single file called `p2route.P`.
- Place all your test input and output in `.in` and `.out` files as specified.
- Submit all your files using the following command:

```
submit -c csc324h -a Prolog filename(s)
```