Extracted from:

Prolog

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

grox.io
Learning

Programmer Passport

Prolog

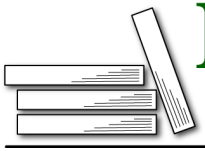
Bruce A. Tate

Edited by Jacquelyn Carter

Prolog

Bruce A. Tate

The Pragmatic Bookshelf
Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Jacquelyn Carter
Copy Editor: L. Sakhi MacMillan
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-935-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 16, 2022

As we delve deeper into Prolog, we've begun to crack some problems that may be a bit more difficult in other languages. Beginning with this chapter, we're going to be working increasingly practical problems.

We'll spend this entire chapter working on the immensely important concept called graphs. Be careful. *Graph* refers to the mathematical concept,[1] not charts based on numbers.

Graphs represent the concept of *connection*. In math, graphs are collections of nodes (also called vertices) and the connections, called edges, between them. A node might be a city on a map or a person in a social network. An edge might be the route between two subway stations, or a follows relationship in social media.

Representing connections is important, but what makes graph theory so powerful is the analysis tools a knowledgeable programmer can bring to bear. Think of it this way. Knowing all of the airports a carrier uses is interesting. Knowing the data about the flights between them is infinitely more important. Scheduling flights to minimize fuel costs and maximize customer load is the difference between profitability and bankruptcy.

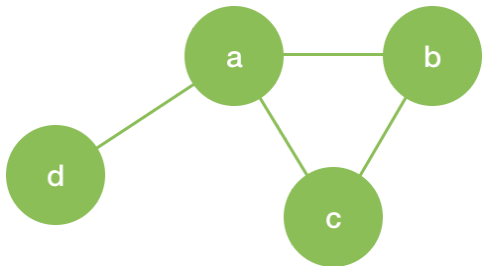
Prolog is the ideal language for dealing with graphs *because the database works that way*. You can often think of a Prolog fact as an edge in a database. That makes it easy to envision what's happening in a graph. In this chapter, we'll use Prolog to answer several kinds of questions:

- What connections exist from a given node?
- Is one node connected to another, either directly or indirectly, via other connected nodes?
- What is the shortest distance between nodes?
- How can we represent data about connections, such as the costs or value of following a connection?

In most cases, we'll write our own algorithms from scratch to answer these questions. In others, we'll use Prolog libraries to solve problems instead.

Let's take a look at a graph and then the Prolog database we might use to represent it. The following image shows a graph. The point of a graph is *not the data represented in each node*. A graph *is about the relationships between nodes*.

1. https://en.wikipedia.org/wiki/Graph_theory



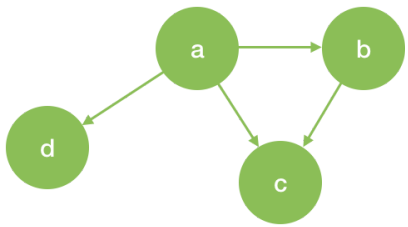
We're going to prefer the less rigid nomenclature of *nodes* instead of *vertices*, and *edges* for the connections between them. We'll label the edges in several different ways including `a-b`, `(a, b)`, or even `edge(a, b)`. In the section that follows, we'll build some basic algorithms with graphs like the previous one to reason about the graphs.

Most modern Prolog implementations, including our SWI Prolog, have libraries for dealing with graphs. We're going to spend a good deal of time focusing on basic algorithms that let us build our own versions of graph-like features. As we get further into the chapter, we'll focus on special cases of graph theory, including weighted graphs and undirected graphs.

In fact, you've already created a couple of Prolog graphs, albeit tiny ones. Both the Twitter follower database in Chapter 1, Logic Programming Basics, on page ?, and the map coloring database in Chapter 2, Logic Problem Solving, on page ?, are graphing problems. Let's create a more formal database graph, with the most basic of representations. You've used Prolog facts in the first two chapters, and we'll use a separate fact to represent each edge of a graph.

Directed Graphs with Edges as Facts

In a directed graph, each edge has a *direction*. In Prolog, edges also imply directions because `fact(a, b)` is not the same as `fact(b, a)`. Consider the same image as the previous one but with a tiny tweak—we've put an arrow on each of the lines:



That means a is connected to b, but b is not connected to a. We can describe the graph like this. a is connected to b, c, and d, and b is connected to c. Directed graphs are ideal for representing roads (which may be one way), *follows*-style relationships in a social media network, or dependencies.

We can label the edges in the graphs with Prolog facts. Create a new directory called directed and the usual db.pl database. Key this much in:

```
edge(a, b).
edge(a, c).
edge(a, d).
edge(b, c).
```

This rule represents our graph. Let's try it out in swipl:

```
?- [db].
true.
?- edge(b, d).
false.
?- edge(a, d).
true.
?- edge(d, a).
false.
```

It's working, and the last query demonstrates that our graph is connected. An edge between a and d does not show that d and a are also connected, and that's the behavior we want. If we wanted to, we could represent a two-way connection by adding an edge twice, like edge(a, b). edge(b, a).

Now, let's see what kinds of questions we can derive directly from that graph. Here are four:

- Are two specific nodes connected?
- What are the neighbors of a node?
- What are the sources leading to a node?
- What are all of the connections?

Let's use the database to answer questions like the ones above:

```
?- [db].
true.

?- edge(d, c).
false.

?- edge(a, Neighbor).
Neighbor = b ;
Neighbor = c ;
Neighbor = d.

?- edge(Source, c).
```

```
Source = a ;
Source = b.

?- edge(Source, Destination).
Source = a,
Destination = b ;
Source = a,
Destination = c ;
Source = a,
Destination = d ;
Source = b,
Destination = c.
```

Each of our edge facts represent one of the arrows on our directed graph. Look closer at the results of the four queries. The first asks the question "Is d connected to c?" We don't have to lock in both of the nodes, though. The second and third queries ask "What are the neighbors of this node?" and "What sources feed into this node?" These queries lock in one argument and let Prolog fill in the other. The final query asks Prolog to find *all* of the combinations that fulfill the query.

As with many Prolog predicates, our edge predicate is flexible. Sometimes we use it to fill in the left side or the right side. Let's build a predicate that collects all neighbors in a single list.

Find Neighbors with findall

The Prolog clauses findall, bagof, and setof are clauses that take goals. In a sense, they're like *higher order functions* in other languages, or functions that take functions as arguments. In Prolog, goals that take goals are *meta predicates*. We're going to use findall to find all of the neighbors of a node.

The purpose of findall is to find all data that satisfies a goal. Each invocation will look like this:

```
findall(Template, Goal, List).
```

This predicate finds a List of all objects in the shape of Template that satisfy a Goal. That description is a bit abstract, so let's use it to find all sources or destinations for a node. Add these rules to your database:

```
sources(Destination, List) :-
  findall(Source, edge(Source, Destination), List).

destinations(Source, List) :-
  findall(Destination, edge(Source, Destination), List).
```

In the first rule, we're finding a List of all Sources for a Destination. We're using findall. The first argument is the element we want to add to the list. The second is the goal, edge in our case. The final one is the List we're collecting.

The second rule works in exactly the same way but looks for a Destination instead. Our solution works like this:

```
?- [db].
true.

?- sources(c, Set).
Set = [a, b].

?- destinations(a, Set).
Set = [b, c, d].
```

Nice! We can use our edge facts to set individual goals, and our other rules to get a whole list of solutions. Let's crank up the complexity, and the power.

edges can show whether two nodes are connected through an edge but can't show whether two nodes are connected through a series of intermediate hops. Let's solve that next.