Graph Algorithms in PROLOG

Adam C. Volk
Department of Mathematics
University of Dayton
300 College Park
Dayton, Ohio 45469
volka1@udayton.edu

ABSTRACT
We consider a PROLOG implementation of a directed graph data structure and how to solve multiple graph-theoretic problems using it. PROLOG belongs to the declarative paradigm of programming languages. Unlike imperative languages in which the programmer specifies how to solve a problem, in a declarative language, the programmer specifies what they want to find, and the system uses a search built into the language.

Keywords
graphs, logic programming, pattern matching, PROLOG

1. INTRODUCTION
Prior to this semester, the only programming languages I had studied were Java and C++. Both of these languages can be classified as object-oriented programming languages. Through languages such as LISP, ML, and Haskell, we were introduced to the functional paradigm of programming languages. For this project, I studied another programming language paradigm, logic programming. I did this by reading about, following examples, and writing programs in PROLOG.

Due to their wide range of applications, graphs are a commonly studied structure in mathematics and computer science. In this project I worked on an implementation of this data structure in PROLOG by writing various predicates to outline their characteristics.

2. A DECLARATIVE PROGRAMMING LANGUAGE

PROLOG, which stands for programming in Logic, is classified as a declarative programming language. Rather than writing code for how to find a solution, the programmer describes what the solution should look like and provides rules and facts for achieving this. The job of finding a solution is left to the interpreter. Consider the following code:

\begin{verbatim}
color(red).
color(yellow).
color(blue).
\end{verbatim}

This is treated as a set of facts that the interpreter will use in its logic. This example says that red, yellow, and blue are colors. We can test them in the interpreter by submitting queries such as the ones below.

\begin{verbatim}
?- color(red).
?- color(X).
?- color(pink).
\end{verbatim}

In this case the first line returns true while the last one returns false. Capital letters, as well as the underscore, are used to denote variables. Thus, the second line will return as many values as we request for which the query is true. Here it would be the three defined colors.

Defining facts is a starting point, but not very useful for practical purposes. The next class of predicates are called rules. Rules can extend facts and relations so that they can be applied recursively to a wider range of inputs. The following is the typical append and member functions, with sample queries and outputs.

\begin{verbatim}
append([], X, X).
append([X|L1], L2, [X|L12]) :- append(L1, L2, L12).
member(X, List) :- append(_, [X|_], List).
\end{verbatim}

\begin{verbatim}
?- append([a,b,c], [d,e,f], Y).
Y = [a, b, c, d, e, f].
?- member(4, [2,4,6,8]).
true.
\end{verbatim}

These predicates are used in the main file for this project. We now have the basic tools for PROLOG that we need, but before proceeding to the main problem, we must first discuss the objects of study, graphs.

3. DIRECTED GRAPHS IN MATHEMATICS

A directed graph is an ordered pair consisting of a set of vertices and a set of edges. A directed edge from a vertex \( u \) to vertex \( v \) is denoted by the ordered pair \( (u, v) \). In this case, we say \( u \) is adjacent to \( v \). For example, let

\[ G = \{(a, b, c, d), ((a, b), (b, c), (c, d), (d, b))\} \]

This defines the structure represented in Figure 1.
We now define a few classes of directed graphs. A chain is a list of vertices such that each vertex is adjacent to the next vertex in the list. A cycle is a chain in which the final vertex is adjacent to the first. A complete graph is a graph in which each vertex is adjacent to every other vertex. An independent set is a set of vertices with no edges between them. These two classes of graphs are complements of each other for this reason.

Note that in the above predicates, we use another predicate called member which checks a graph to see if there is a member(Edge1, Eset) graph that contains a given edge. It does this by checking if there are no duplicate edges and by making sure that the defined edges do not use vertices which are not included in the vertex set. The second predicate takes a graph and an edge, returning true or false depending on whether or not the graph is valid and the edge is a member of the graph’s edge set. The third predicate serves the same function for vertices.

These rules serve as a way of introducing graphs in Prolog and are building blocks from which we can consider more interesting questions. For example, we can check if one graph is a subgraph of another one. We can also check whether or not a graph has a cycle in general or a cycle containing a given vertex. We do this with the following Prolog predicates:

```prolog
subgraph ([Vset1, Eset1], [Vset2, Eset2]) :-
  graph ([Vset1, Eset1]),
  graph ([Vset2, Eset2]),
  subset (Vset1, Vset2),
  subset (Eset1, Eset2).

has_cycle (Graph, Vertex) :-
  chain (Graph, Vertex, Vertex, _).

cycle_vertices (G, [V1|Vset]) :-
  has_cycle (G, V1),
  cycle_vertices (G, Vset).

has_cycle ([[V1|Vset], Eset]) :-
  cycle_vertices ([[V1|Vset], Eset], [V1|Vset]).
```

Note that in the above predicates, we use another predicate called chain which checks a graph to see if there is a path from a start vertex to an end vertex. As mentioned above, a cycle is a path where the start vertex and end vertex are the same vertex. For one final example of a test we can run on graphs using this programming language, consider the question of whether or not a graph is complete. We first consider the complement: a graph with no edges. As stated previously this is called an independent set. To check if a valid graph falls in this class, we check if the edge set is empty. A complete directed graph has no loops, but all other possible edges. A complete directed graph with n vertices has precisely n × (n – 1) edges. We can check if a graph is complete by verifying that a valid graph has no self-edges and that the number of edges satisfies this condition. Below are the required rules and helper predicates for implementing this test:

```prolog
/* count number of elements in a list */
count([] , 0).
count([X|Y], Y) :- count(X, Z), Y is Z+1.

/* for graph with X vertices, checks if X(X-1) edges */
proper(Y, X) :-
  Z is Y - X*(X-1), Z == 0.
```

As stated previously, a graph is an ordered pair consisting of a set of vertices and a set of edges. To implement this structure in Prolog, we treat these two sets as lists with each edge being a list of two elements. Using this format, we write the previously defined graph as:

*Figure 1: G with vertices labeled.*

![Figure 1: G with vertices labeled.](image)

\[ G = [[a, b, c, d], [[a, b], [b, c], [c, d], [d, b]]] \]

As stated previously, a graph is an ordered pair consisting of a set of vertices and a set of edges. To implement this structure in Prolog, we treat these two sets as lists with each edge being a list of two elements. Using this format, we write the previously defined graph as:

\[ G = [[a, b, c, d], [[a, b], [b, c], [c, d], [d, b]]] \]

Note that for the purposes of this project, we allow loops, directed edges from a vertex to itself, however, we do not allow duplicate edges. That is, there can be no more than one directed edge from one vertex to another. It is okay for there to be directed edges going both ways though.

4. GRAPHS IN PROLOG

As stated previously, a graph is an ordered pair consisting of a set of vertices and a set of edges. To implement this structure in Prolog, we treat these two sets as lists with each edge being a list of two elements. Using this format, we write the previously defined graph as:

\[ G = [[a, b, c, d], [[a, b], [b, c], [c, d], [d, b]]] \]

4.1 Implementation

Now that we know a little bit about Prolog, graphs, and how graphs look in Prolog and are building blocks from which we can consider more interesting questions. For example, we can check if one graph is a subgraph of another one. We can also check whether or not a graph has a cycle in general or a cycle containing a given vertex. We do this with the following Prolog predicates:

```prolog
graph ([Vertices, Edges]) :-
  checkDuplicateEdge (Edges),
  my_flatten (Edges, X),
  makeset (X, Y),
  subset (Vertices, Y).
```

edge ([Vset, Eset], Edge1) :-
  graph ([Vset, Eset]),
  member (Edge1, Eset).

vertex ([Vset, Eset], Vertex1) :-
  graph ([Vset, Eset]),
  member (Vertex1, Vset).
```

This first predicate tests whether or not a given input represents a valid graph. It does this by checking if there are no duplicate edges and by making sure that the defined edges do not use vertices which are not included in the vertex set. The second predicate takes a graph and an edge, returning true or false depending on whether or not the graph is valid and the edge is a member of the graph’s edge set. The third predicate serves the same function for vertices.

These rules serve as a way of introducing graphs in Prolog and are building blocks from which we can consider more interesting questions. For example, we can check if one graph is a subgraph of another one. We can also check whether or not a graph has a cycle in general or a cycle containing a given vertex. We do this with the following Prolog predicates:

```prolog
subgraph ([Vset1, Eset1], [Vset2, Eset2]) :-
  graph ([Vset1, Eset1]),
  graph ([Vset2, Eset2]),
  subset (Vset1, Vset2),
  subset (Eset1, Eset2).

has_cycle (Graph, Vertex) :-
  chain (Graph, Vertex, Vertex, _).

cycle_vertices (G, [V1|Vset]) :-
  has_cycle (G, V1),
  cycle_vertices (G, Vset).

has_cycle ([[V1|Vset], Eset]) :-
  cycle_vertices ([[V1|Vset], Eset], [V1|Vset]).
```

Note that in the above predicates, we use another predicate called chain which checks a graph to see if there is a path from a start vertex to an end vertex. As mentioned above, a cycle is a path where the start vertex and end vertex are the same vertex. For one final example of a test we can run on graphs using this programming language, consider the question of whether or not a graph is complete. We first consider the complement: a graph with no edges. As stated previously this is called an independent set. To check if a valid graph falls in this class, we check if the edge set is empty. A complete directed graph has no loops, but all other possible edges. A complete directed graph with n vertices has precisely n × (n – 1) edges. We can check if a graph is complete by verifying that a valid graph has no self-edges and that the number of edges satisfies this condition. Below are the required rules and helper predicates for implementing this test:

```prolog
/* count number of elements in a list */
count([], 0).
count([X|Y], Y) :- count(X, Z), Y is Z+1.

/* for graph with X vertices, checks if X(X-1) edges */
proper(Y, X) :-
  Z is Y - X*(X-1), Z == 0.
```
*/ checks if graph is an independent set */
independent([Vset,[]]) :-
    graph([[Vset],[]]).

/* checks if graph is complete (infinite recursion problem) */
complete([Vset,Eset]) :-
    graph([Vset,Eset]),
    not(member([X,X],Eset)),
    count(Vset,X), count(Eset,Y),
    proper(Y,X).

5. RESULTS
These methods were tested using SWI-PROLOG as an interpreter. The file consulted includes all of the aforementioned predicates as well as helpers excluded from this paper for the sake of space. Each method provided the correct output for all cases tested. One issue that arises is that of infinite recursion. In particular, in some cases, when a query is checked, the program correctly outputs true; however, if the user presses the semicolon rather than the period, it will return true what appears to be an infinite number of times. We now provide a few examples of input with their corresponding output.

?- graph([[a,b,c],[a,b],[b,c],[d,a]]). false.
?- edge([[a,b,c],[a,b],[b,c],[d,a]],[a,b]). true.
?- has_cycle([[a,b,c,d],[a,b],[b,c],[c,d],[d,b]]). true.
?- has_cycle([[a,b,c,d],[a,b],[b,c],[c,d],[d,b]],[a]). false.
?- has_cycle([[a,b,c,d],[a,b],[b,c],[c,d],[d,b]],[d]). true.
?- subgraph([[a,b,c],[a,b],[a,c],[b,c]],[a,b,c],[[a,b],[a,c]]). true.
?- complete([[],[[]]]). true.
?- complete([[a,b,c],[a,b],[a,c],[b,a],[b,c],[c,a],[c,b]]).

6. FUTURE WORK
Future work for this project could be checking base cases to see if they can be adjusted to account for the infinite loops in output. This project also included just a couple of questions that can be asked about graphs. This problem can be extended to deal with a wide range of important topics in the mathematical branch of graph theory. Such tests might include chromatic numbers, graph decomposition, checking for hamilton cycles. I am not sure if PROLOG would be an appropriate language for dealing with multigraphs or infinite graphs, but that is something that could be considered as well.

7. CONCLUSION
Whether it is easier or harder to implement graphs in PROLOG compared to imperative languages is a question that I do not feel suited to answer. What I will say is that it is very different and requires another way of thinking about the problem at hand. I believe the same can be said about most problems using this logic programming language. Taking on this project with no prior background using PROLOG was a challenge, but one that was manageable. One difficulty was trying to apply imperative ideas in a declarative language. PROLOG required elements studied in other languages, most of all pattern matching. I believe that this project served as a fitting end to a course in which I was exposed to not only a wide range of languages that I had never seen before such as LISP, Go, Haskell, and Elixir, but an entirely new paradigm of programming as well.

8. REFERENCES