24 Graph Algorithms

24.1 Introduction

There are many important algorithms on graphs. In this chapter we examine a few of the most fundamental and widely used. In particular, we consider graph search algorithms and several algorithms based on them.

24.2 Graph Algorithms in Ruby

The graph algorithms we study use only the operations in the Graph class, so we can implement them in the Graph class to be inherited by the ArrayGraph and LinkedGraph classes. Several methods are abstract in the Graph class but implemented in its sub-classes. Thanks to dynamic binding in Ruby, the correct operations are called when a graph algorithm invokes a particular method that is abstract in the Graph class. Placing graph algorithms in the Graph class keeps them in a good place and saves work by allowing us to write them only once while still having them available in the ArrayGraph and LinkedGraph sub-classes.

24.3 Searching Graphs

Many problems involving graphs require a systematic traversal of the graph along its edges—this is termed a **graph search**. We have already seen graph search algorithms in the form of tree traversals, but these only apply to trees, not graphs in general.

As an example of a problem illustrating the need to search a general graph, suppose that we wish to find a path through a maze. A maze can be represented by a graph as follows: map the intersections in the maze to graph vertices and map the paths in the maze to graph edges. Moving through a maze from its entry to its exit corresponds to the problem of searching a graph representing the maze to find a path from the entry vertex to the exit vertex.

There are two main approaches to graph searching.

> **Depth-first search**: A search that begins by visiting vertex v, and then recursively searches the unvisited vertices adjacent to v.
>
> **Breadth-first search**: A search that begins by visiting vertex v, then visits the vertices adjacent to v, then visits the vertices adjacent to each of v's adjacent vertices, and so on.

Vertices in a depth-first search are visited in an order that follows paths leading away from the starting vertex until the paths can be followed no further; the algorithm then backs up and follows other paths. Vertices deep in the graph (relative to the starting vertex) are visited sooner than shallow vertices. In contrast, during a breadth-first search the vertices closest to the starting vertex are visited first, then those a bit further away, and so on. Some problems are solved with one kind of search, some with the other, and in some cases it does not matter which kind of search is used.

24.4 Depth-First Search

Given its recursive characterization, it is not surprising that depth-first search is easily implemented using recursion. Consider the Ruby code in Figure 1.

```
def dfs(v)
  raise ArgumentError, "No such vertex" if v < 0 or vertices <= v
  is_visited = []
  visit = lambda do |v|
    each_edge(v) do |v,w|
      next if is_visited[w]
      yield v,w
      is_visited[w] = true
      visit.call(w)
    end
  end
  yield -1,v
  is_visited[v] = true
  visit.call(v)
end
```

Figure 1: Recursive Depth-First Search

The `dfs()` method is an iterator that yields every vertex in a graph in depth-first order as the second of the vertices in the edge used to visit the vertex. The `dfs()` function has an array of Boolean values that keeps track of whether a vertex has been visited. The real work is done by the inner `visit()` function. This recursive function takes a visited vertex `v` as its argument. It yields all the edges from `v` to each unvisited adjacent vertex `w`, marks `w` as visited, and calls itself on `w`.

Notice how similar this processing is to a preorder tree traversal. In the tree traversal, the root is visited and then its children are visited recursively. In the depth-first search, a vertex is visited and then its adjacent vertices are visited recursively. The depth-first search is more complex only because it must be able to handle cycles—this is what the `is_visited` array is for.

Just as with a binary tree traversal, we can also write a depth-first search using a stack rather than recursion. Figure 2 shows this algorithm. The stack-based algorithm uses the same strategy as the recursive algorithm: it keeps track of unvisited vertices and only processes those that have not yet been visited. Where the recursive algorithm makes recursive calls on unvisited adjacent vertices after a vertex has been visited, this algorithm places the unvisited adjacent vertices in a stack. (Actually, the edges to the adjacent vertices are placed in the stack so that they can be yielded at the right time.) Vertices are popped from the stack and processed until the stack is empty.

```
Edge = Struct.new(:v, :w) # for storing edges on the stack

def stack_dfs(v)
  raise ArgumentError, "No such vertex" if v < 0 or vertices <= v
  stack = LinkedStack.new
  is_visited = []
  stack.push(Edge.new(-1,v))
  while !stack.empty? do
    edge = stack.pop
    next if is_visited[edge.w]
    yield edge.v,edge.w
    is_visited[edge.w] = true
    each_edge(edge.w) do |w,x|
      stack.push(Edge.new(w,x)) if !is_visited[x]
    end
  end
end
```

Figure 2: Stack-Based Depth-First Search

Both versions of the depth-first search algorithm visit each vertex at most once and process each edge leaving every visited vertex exactly once. Each edge has two vertices so each edge is processed at most twice. Hence depth-first search runs in $O(v+e)$ time in the worst case, where v is the number of vertices and e is the number of edges in the graph.

24.5 Breadth-First Search

The stack-based version of depth-first search places vertices adjacent to the current vertex in a stack, then goes on to process the vertex on the top of the stack, placing its adjacent vertices on the stack, and so forth. The effect of this strategy is that vertices adjacent to the initial vertex are at the bottom of the stack and therefore get processed after vertices further away. If we use a queue instead of a stack, then adjacent vertices are processed first, then those adjacent to those next, and so on. In short, we can make a breadth-first search algorithm from the stack-based depth-first search algorithm by simply replacing the stack with a queue. Such an algorithm is shown in Figure 3.

This algorithm visits each vertex exactly once and follows each edge at most twice, so its performance is in $O(e+v)$, just like its stack-based peer. The only difference is that is visits vertices in a different order.

```
def bfs(v)
  raise ArgumentError, "No such vertex" if v < 0 or vertices <= v
  queue = LinkedQueue.new
  is_visited = []
  queue.enter(Edge.new(-1,v))
  while !queue.empty? do
    edge = queue.leave
    next if is_visited[edge.w]
    yield edge.v,edge.w
    is_visited[edge.w] = true
    each_edge(edge.w) do |w,x|
      queue.enter(Edge.new(w,x)) if !is_visited[x]
    end
  end
end
```

Figure 3: Queue-Based Breadth-First Search

24.6 Paths in a Graph

We can use graph searching algorithms to determine properties of graphs. Recall that two vertices are connected if and only if there is a path between them. We can check whether vertices are connected using either depth-first or breadth-first search by starting a search at one vertex and determining whether the search ever reaches the other vertex. Ruby code for such a method appears in Figure 4.

```
def path?(v1,v2)
 return false if v1 < 0 or vertices <= v1
 return false if v2 < 0 or vertices <= v2
 dfs(v1) do |v,w|
 return true if w == v2
 end
 false
end
```

Figure 4: Determining Whether Two Vertices are Connected

This method first checks whether the argument vertices are even in the graph—if one is not, then there is no path between them. It then uses depth-first search from the first vertex to check whether the second is ever reached and returns the result.

If two vertices are connected, there may be more than one path between them, and often it is useful to know the shortest path (the one with the fewest edges). The method in Figure 5 finds the shortest path between two vertices.

```
def shortest_path(v,w)
  raise ArgumentError unless path?(v,w)
  to_edge = []
  bfs(w) { |v1,v2| to_edge[v2] = v1 }
  result = []
  x = v
  while x != w
    result << x
    x = to_edge[x]
  end
  result << x
end
```

Figure 5: Finding the Shortest Path Between Connected Vertices

This method first makes sure that there is a path between the argument vertices and raises an exception if there is not. The core of the algorithm is a search that takes a source vertex `v` and constructs an array that contains, for each vertex except `v`, the vertex next on the shortest path back to `v`. This array, called `to_edge` in Figure 5, must be constructed using a breadth-first search. A vertex `x` adjacent to `v` has its `to_edge` entry set to `v` because the shortest path from `x` back to `v` is obviously the edge to `v`. The vertices adjacent to `x` have their `to_edge` entries set to `x` because the shortest path back to `v` goes first to `x` and then to `v` (a shorter path could only exist if these vertices were adjacent to `v`). The `to_edge` entries are constructed in like fashion for vertices further from `v`. Clearly, a breadth-first search is needed to visit vertices in the order necessary to make this work. Once the `to_edge` array is constructed, it is an easy task to traverse it from the target vertex back to the source vertex to generate a shortest path between the two. In Figure 5, argument `w` is treated as the source vertex and `v` as the target so that the array returned lists the path from `v` to `w`.

24.7 Connected Graphs and Spanning Trees

Besides determining whether two vertices are connected, we can also determine whether an entire graph is connected, and in much the same way. In this case, we can do a depth-first or breadth-first search from any vertex in the graph and check whether every other vertex is visited. The graph is connected if and only if every other vertex is visited by either a breadth-first or depth-first search starting at a source vertex. We leave the code for this algorithm as an exercise.

Recall that a spanning tree is a sub-graph of a graph *g* that is a tree and contains every vertex of *g*. If we visit every vertex in a connected graph *g* from a source vertex and add the edge along which each vertex is visited to a new graph *h*, then *h* will be a spanning tree for *g* when we are done. This is the strategy used to construct a spanning tree in the Ruby code in Figure 6.

```
def spanning_tree
  raise ArgumentError unless connected?
  result = (self.class.to_s == "ArrayGraph") ?
           ArrayGraph.new(vertices) :
           LinkedGraph.new(vertices)
  dfs(0) { |v,w| result.add_edge(v,w) if 0 <= v }
  result
end
```

Figure 6: Making a Spanning Tree for a Connected Graph

This method returns creates a new graph for the spanning tree that has the same class as the host class.

24.8 Summary and Conclusion

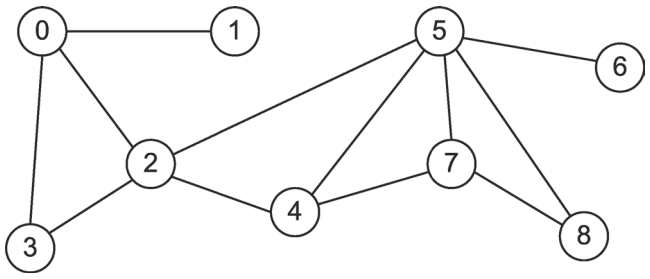
Depth-first and breadth-first search are two ways to visit the vertices of a graph by following its edges in an organized way. Both algorithms work in time proportional to the number of edges and vertices in the graph. Depth-first and breadth-first search are the basis for many algorithms that process graphs in various ways, including determining whether two vertices are connected, determining whether a graph is connected, finding the shortest path between two vertices, and finding a spanning tree for a connected graph.

24.9 Review Questions

1. What is the relationship between graph search and graph traversal?
2. How are graph searching algorithms related to stacks and queues?
3. Under what circumstances is one graph searching algorithm preferable to the other?
4. Does it matter whether we use depth-first or breadth-first search to find the shortest path between two vertices?
5. Does it matter whether we use depth-first or breadth-first search to generate a spanning tree for a connected graph?

24.10 Exercises

Use the graph below to do the exercises.



1. List the order in which the vertices are visited in a depth-first search from vertex 0 in the graph above. Assume that adjacent vertices are visited in order from smallest to largest.
2. List the order in which the vertices are visited in a breadth-first search from vertex 0 in the graph above. Assume that adjacent vertices are visited in order from smallest to largest.
3. Trace the execution of the shortest-path function in generating a shortest path between vertices 0 and 8 in the graph above. What is the path?
4. Trace the execution of the spanning tree function in generating a spanning tree for the graph above. What is the spanning tree?

5. Write a Ruby method `connected?()` in the `Graph` class to determine whether the graph is connected.
6. Write a Ruby method `component_count()` in the `Graph` class to count the number of connected components in the graph.
7. The *degree* of a vertex is the number of edges connected to it. Write a Ruby method `max_degree()` in the `Graph` class to find the maximum degree of the vertices in the graph.
8. The following problems can be solved by modeling the problem with a graph and then applying graph functions to the model. Explain how to solve these problems using graphs and graph algorithms.
   a) A path through a maze.
   b) The smallest set of phone calls that must be made to transmit a message amongst a group of friends.
   c) Determining whether it is possible to get from point A to point B using only main highways.
   d) Finding the degree of separation between two people in a community (the *degree of separation* between two people who know each other is one; the degree of separation between two people who do not know each other but have a mutual friend is two, and so on).

24.11 Review Question Answers

1. Graph search is another name for graph traversal.
2. The depth-first search algorithm uses a stack (or recursion), while the breadth-first search algorithm uses a queue. Otherwise, they are virtually identical.
3. The depth-first and breadth-first search algorithms run in the same amount of time and use the same amount of space, so there is no basis for preferring one over the other in general. However, some algorithms require one or the other to work properly.
4. The shortest path algorithm is an example of an algorithm that requires one of the search algorithms to work properly; in particular, the shortest path algorithm requires a breadth-first search to work properly.
5. The spanning tree algorithm is an example of an algorithm where is does not matter which search algorithm is used: either depth-first or breadth-first search can be used to generate a spanning tree.