CHAPTER 17

Searching Graphs

Chapter Objectives

This chapter demonstrates algorithms for finding good paths through a graph. However, first we need to understand the following:

- How to construct and use two special forms of data collection: queues and priority queue
- How to build a model of a graph
- How to traverse and search a graph

Introduction

The collections we have considered so far—vectors, arrays, structures, and cell arrays—have essentially been collections of fixed size. In practice, of course, many MATLAB operations disguise the fixed character of the collections by performing complex operations that are invisible to the user to change the size of the collection.

The ultimate goal of this chapter is to discuss the most general form of dynamic data structure: the graph. In computer science, a **graph** is a collection of **nodes** connected by **edges**, as opposed to the engineering view of a graph that is produced by plotting, for example, $f(x)$ against x. To process graphs effectively, we must first consider two simpler concepts: a **queue** and a **priority queue**.

17.1 Queues and Priority Queues

We first consider the nature and implementation of queues, special collections that enable us to process graphs efficiently. We experience the concept of a queue every day of our lives. A line of cars waiting for the light to turn green is a queue; when we stand in line at a store or send a print job to a printer, we experience typical queue behavior. In general, the first object entering a queue is the first one to exit the other end.

17.1.1 The Nature of a Queue

Formally, we refer to a queue as a first in/first out (FIFO) collection, as illustrated in Figure 17.1. The most general form of a queue is permitted to contain any kind of object, that is, an instance of any data type or class. Typically, operations on a queue are restricted to the following:

- `enqueue` puts an object into the queue
- `dequeue` removes an object from the queue
- `peek` copies the first object out of the queue without removing it
- `isempty` determines that there are no items in the queue

17.1.2 Implementing Normal Queues

Although there are many ways to implement a queue, a cell array is a good choice because it is a linear collection of objects that may be of any type. If we establish a queue using a cell array, the mechanizations are trivial as follows:

- `qEnq` concatenates data at the end of the cell array
- `qDeq` removes the item from the front of the cell array and returns that item to the user
- `peek` merely accesses the first item in the cell array
- `isempty` is the standard MATLAB test for the empty vector

Clearly, because all the cell array operations are also accessible to the programmer, nothing prevents an unscrupulous programmer from using other operations on the queue—for example, adding an item to the front of the queue rather than the back to effectively "jump in line." There are implementations beyond the scope of this text that use object-oriented

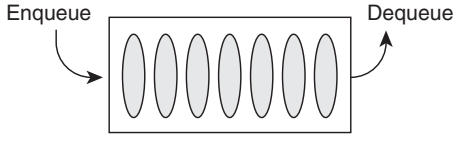


Figure 17.1 *A queue*

programming techniques to encapsulate the data and restrict the available operations on that data to those that implement the required functionality. However, for our purposes, the "open" cell array implementation is sufficient. Functions that perform the enqueue and dequeue operations for a queue are shown in Listing 17.1.

In Listing 17.1:

Line 1: Obviously, these two trivial functions should actually be in separate files. Both functions must return the updated queue because they receive copies of the original queue.

Line 2: Concatenates the enqueued item with a container (the braces) at the back of the cell array.

Line 3: The `dequeue` function must return the new queue and the item being removed.

Line 4: We return the first item on the cell array and remove it by returning the rest.

17.1.3 Priority Queues

There are times when we wish ordinary queues were priority queues. For example, at your favorite coffee bar when you simply want a cup of regular coffee and there are 10 people ahead of you ordering time-consuming steamed milk drinks, or at the printer where you wait an hour for one page while someone prints large sections of an encyclopedia. The only difference between an ordinary queue and a priority queue is in the `enqueue` algorithm. On a priority queue, the `enqueue` function involves adding the new item in order to the queue, as illustrated in Figure 17.2. For the `enqueue` function to

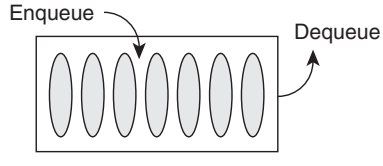


Figure 17.2 *A priority queue*

Listing 17.1 `Enqueue` and `dequeue` functions

```
1. function q = qEnq(q, data)
   % enqueue onto a queue
2.    q = [q {data}];

3. function [q ans] = qDeq(q)
   % dequeue
4.    ans = q{1};
5.    q = q(2:end);
```

add in order, there must be a means of comparing two objects. Here, we use the function `islt` that generally should be able to compare any two objects. In this implementation, it is sufficient to be able to compare numbers or structures that contain either the fields `key` or `dod`. Clearly, this can be extended as necessary to compare any two objects.

The code for `islt` is shown in Listing 17.2.

In Listing 17.2:

Line 1: Shows a function that consumes two objects and returns a Boolean result.

Line 2: Captures the data type (class) of the first object.

Line 3: Initializes the result.

Line 4: Checks that the second object is the same data type—otherwise, the `false` answer is returned.

Line 5: Decides how to compare the objects based on the data type.

Lines 6 and 7: Numbers are easily compared.

Line 8: Selects different structures to compare based on fields in the structure.

Lines 9 and 10: If the field `key` is present, compare these.

Lines 11 and 12: If the field `age` is present, compare these.

Lines 13–18: Show that an error exits.

The `enqueue` function that uses `islt` to compare objects for a priority queue is shown in Listing 17.3.

Listing 17.2 Comparing two objects

```
 1. function ans = islt(a, b)
% comparing two objects
 2. acl = class(a);
 3. ans = false;
 4. if isa(b, acl)
 5.     switch acl
 6.       case 'double'
 7.         ans = a < b;
 8.       case 'struct'
 9.         if isfield(a, 'key')
10.             ans = a.key < b.key;
11.         elseif isfield(a, 'dod')
12.             ans = age(a) < age(b);
13.         else
14.             error('comparing unknown structures')
15.         end
16.       otherwise
17.       error(['can't compare ' acl 's'])
18.     end
19. end
```

Listing 17.3 Priority queue `enqueue` function

```
 1. function pq = pqEnq(pq, item)
    % enqueue in order to a queue
 2.     in = 1;
 3.     at = length(pq)+1;
 4.     while in <= length(pq)
 5.         if islt(item, pq{in})
 6.            at = in;
 7.            break;
 8.         end
 9.         in = in + 1;
10.     end
11.     pq = [pq(1:at-1) {item} pq(at:end)];
```

In Listing 17.3:

Line 1: Shows the same signature as the `enqueue` function for ordinary queues.

Line 2 and 3: Initialize the `while` loop parameters.

Line 4: Moves the index `in` down the existing queue until it falls off the end of the cell array or finds something smaller than the `item` to insert. This second exit is implemented with a `break` statement.

Line 5: Checks whether the `item` is less than the current entry in the queue.

Lines 6 and 7: If so, mark the spot and exit the loop.

Lines 8 and 9: Otherwise, keep moving down the queue.

Line 11: Inserts the `item` in a container between the front part of the queue and the remains of the queue from `at` to the `end`.

17.1.4 Testing Queues

It is always advisable to test utility functions thoroughly before using them in complex algorithms. First we will build a simple utility for presenting the contents of any cell array, and then we will write a script to test the queues.

In order to observe the results from testing the queues, we need a function that will convert a cell array to a string for printing. Although it is tempting to try to write a single function to accomplish this, we achieve more maintainable code by separating the cell array traversal from the details of converting each item to a string. The first function, `CAToString`, which traverses the cell array, is shown in Listing 17.4.

Listing 17.4 Converting a cell array to a string

```
 1. function str = CAToString(ca)
% Traverse a cell array to make a string
 2. str = '';
 3. for in = 1:length(ca)
 4.     str = [str toString(ca{in}) 13];
 5. end
```

In Listing 17.4:

Line 1: The function consumes any cell array and returns a string.

Line 2: Initializes the string.

Line 3: Traverses the cell array.

Line 4: Extracts each item from the container, uses the `toString` utility function below to convert it to a string, and appends it to the end of the output string together with a new-line character (the ASCII value 13).

Of course, the real effort in creating this string is the second function that converts each individual item from the cell array to its string representation. This is shown in Listing 17.5.

In Listing 17.5:

Line 1: Java programmers might recognize the concept of converting an object to its string equivalent.

Lines 2 and 3: If the object is a string, surround it with single quotes.

Listing 17.5 Converting any object to a string

```
 1. function str = toString(item)
% turn any object into its string representation
 2. if isa(item, 'char')
 3.     str = [''''' item ''''];
 4. elseif isa(item, 'double')
 5.     if length(item) == 1
 6.         str = sprintf('%g', item );
 7.     else
 8.         str = '[';
 9.         for in = 1:length(item)
10.             str = [str  ...
                        sprintf(' %g', item(in) ) ];
11.         end
12.         str = [str ' ]'];
13.     end
14. elseif isa(item, 'struct')
15.     if isfield(item, 'name')
16.         str = item.name;
17.     else
18.         nms = fieldnames(item);
19.         str = [];
20.         for in = 1:length(nms)
21.             nm = nms{in};
22.            str = [str nm ': '  ...
                           toString(item.(nm))  13];
23.         end
24.     end
25. else
26.     str = 'unknown data';
27. end
```

Lines 4–6: Individual scalar numbers are printed in `%g` form.

Lines 7–12: Vectors are enclosed in braces.

Line 14: Recursively uses `toString` to print the fields of a structure.

Lines 15 and 16: A special case wherein if there is a `name` field in the structure, the value of that field is used for the string.

Lines 18–23: Extract the field names and iterate through them one at a time, creating a string by appending each field name with its value and a new line.

Listing 17.6 illustrates a test script that exercises most of the available functions for queues and priority queues using numbers. However, a queue can contain any object you can display, and a priority queue can contain any object you can display and compare to another of the same type.

In Listing 17.6:

Line 1: Initializes a queue.

Lines 2–4: Enqueue 10 numbers.

Line 5: Displays the resulting queue.

Lines 6 and 7: Dequeue and print one value and the remaining queue.

Line 8: Peeks at the head of the queue and verifies that we have not changed its contents.

Line 9: Creates a priority queue.

Lines 10–14: Enqueue 10 random integers.

Line 15: Lists the queue to show that they were enqueued in order.

Listing 17.6 Testing the queues

```
 1. q = [];
 2. for ix = 1:10
 3.     q = qEnq(q, ix);
 4. end
 5. CAToString(q)
 6. [q ans] = qDeq(q);
 7. fprintf('dequeue -> %d leaving \n%s\n', ...
        ans, CAToString(q) );
 8. fprintf('peek at queue -> %d leaving \n%s\n', ...
        q{1}, CAToString(q) );

 9. pq = [];
10. for ix = 1:10
11.     value = floor(100*rand);
12.     fprintf(' %g:', value );
13.     pq = pqEnq(pq, value );
14. end
15. fprintf('\npriority queue is \n%s\n', ...
                                    CAToString(pq) );
```

The results from this test are shown in Table 17.1.

Table 17.1 Results from testing the queues

```
ans =                          peek at queue -> 2 leaving
1                              2
2                              3
3                              4
4                              5
5                              6
6                              7
7                              8
8                              9
9                              10
10                              95: 23: 60: 48: 89: 76:
dequeue -> 1 leaving           45: 1: 82: 44:
2                              priority queue is
3                              1
4                              23
5                              44
6                              45
7                              48
8                              60
9                              76
10                             82
                               89
                               95
```

17.2 Graphs

This chapter focuses on processing a graph—the most general form of dynamic data structure, an arbitrary collection of nodes connected by edges. A street map is a good example of a graph (the intersections are the nodes and the streets are the edges). The edges may be **directional** to indicate that the graph can be traversed along that edge in only one direction (like a one-way street). The edges may also have a value associated with them to indicate, for example, the cost of traversing that edge. We refer to this as a **weighted graph**. For a street map, this cost could either be the distance, or in a more sophisticated system, the travel time—a function of the distance, the speed limit, and the traffic congestion. Graphs are not required to be completely connected, and they can contain **cycles**—closed loops in which the unwary algorithm could become trapped. Graphs also have no obvious starting and stopping points. A **path** on a graph is a connected list of edges that is the result of traversing a graph.

17.2.1 Graph Examples

A simple graph is shown in Figure 17.3. In the figure the connection points `A ... F` are the nodes and the edges are the interconnecting lines, which are

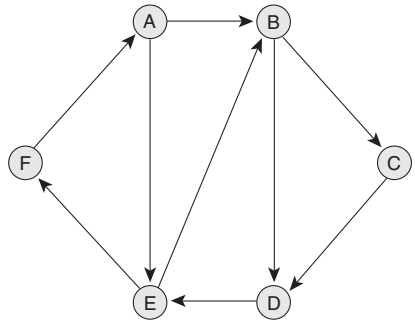


Figure 17.3 *A simple graph*

directional but not weighted. Graphs occur frequently in everyday life, as illustrated by the underground map shown in Figure 17.4. The design of the London rail system was motivated by the need to travel from any station to another while changing trains twice at most. Historically, this particular graph is notable because it was the first map constructed without strict adherence to the physical location of the nodes.

We could actually view the cell array used to construct a queue as a limited form of a graph with no cycles and only one directional edge proceeding from each node. The nodes would be the cell contents, and the edges would be the connectivity provided by the indexing. Traversing this cell array is simple because there is a natural starting place at the beginning of the array and a natural stopping place at the end.

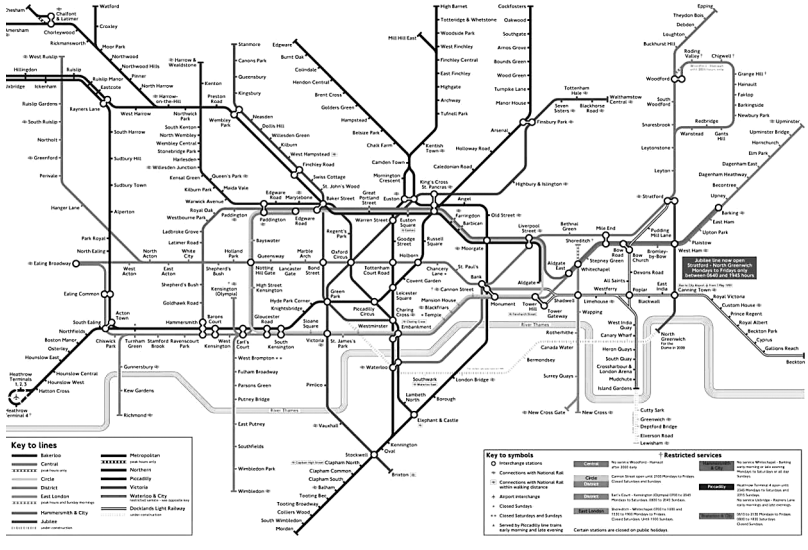


Figure 17.4 *Map of the London Underground*

17.2.2 Processing Graphs

In designing algorithms that operate on graphs in general, we need to consider the following constraints:

- With cycles permitted in the data, there is no natural starting point like the beginning of a cell array. Consequently, the user must always specify a place on the graph to start a search, as well as the place to end.
- There are no natural "leaf nodes" where a search might have to stop and back up. Consequently, an algorithm processing a graph must have a means of determining that being at a given node is the "end of the line." Typically, this is accomplished by maintaining a collection of visited nodes as it progresses around the graph. Each time a node is considered, the algorithm must check to see whether that node is already in the visited collection. If so, it refuses to return to that node. The algorithm must backtrack if it reaches a node from which there is no edge to a node that has not already been visited.
- Whereas on a cell array there is only one feasible path from one node to another, there may be many possible paths between two nodes on a graph. The best algorithms that search for paths must take into account a comparison between paths to determine the best one.

The upcoming sections discuss the following:

- Constructing a graph from the standpoint of understanding the structure of the representation
- Breadth-first search that always finds the path with the least number of nodes
- Optimal search that generates the best path through a graph based on the cost of each edge

For a simple, consistent example, consider the graph shown in Figure 17.5.

We might want to find a path from node A to node H. Since there are no arrows shown on the graph, we can assume that all edges are bidirectional.

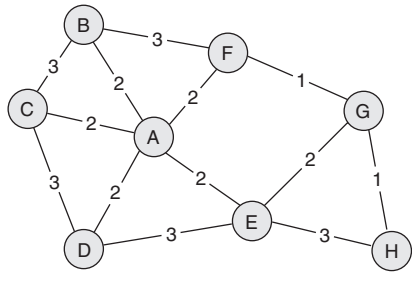


Figure 17.5 *A weighted graph*

The first step would be to create a representation of that graph. The second step would be to find a suitable way to search that graph for the best path.

17.2.3 Building Graphs

We need to consider graphs as two collections of data as follows:

- A list of `n` nodes with the properties of identity (a name) and position
- An `n` × `n` adjacency matrix that specifies the weight of the edge from each node to any other node

If one node is not reachable from another, by convention we will specify that weight as 0. This is actually a rather intimidating structure to build "by hand." In order to facilitate reliable construction of the adjacency matrix, we start with a simpler description of the graph shown in Figure 17.5. This graph can be described initially with the following data:

- `cost` a vector of size `m` × `1` containing the weights for each of `m` edges
- `dir` a vector of size `m` × `1` indicating the directionality of each edge as follows:
  - `2` two-way edge
  - `1` one way in a positive direction
  - `-1` one way in the other direction
- `node` a matrix of size `n` × `rows` containing the edge indices for each node. For example, if `node(i, j)` contains `x`, it says that the `i`th node connects to the `x`th edge. If `x` is 0, there is no connection. The value `rows` is the maximum number of nodes that can be reached from any other node.
- `coord` a matrix of size `n` × `2` containing the x-y coordinates of each node that is used only for the graphical representation of the graph.

The script shown in Listing 17.7 starts with the above representation of the graph and calls the function `grAdjacency(...)` to produce the adjacency matrix. We will save this script as the constructor script `makeGraph.m`. Again referencing Figure 17.5, the sequence of edges used in this script are as follows:

```
A-B, A-C, A-D, A-E, A-F, B-F, B-C, C-D, D-E, F-G, E-G, G-H, E-H
```

Listing 17.7 Constructing a simple graph

```
% edge weights
    cost = [2 2 2 2 2 3 3 3 3 1 2 1 3];
% edge directions
    dir =  [2 2 2 2 2 2 2 2 2 2 2 2 2];
% connectivity
```

continued on next page

```
    node = [ 1  2  3 4 5; ...      % edges from A
             1  6  7 0 0; ...      % edges from B
             2  7  8 0 0; ...      % edges from C
             3  8  9 0 0; ...      % edges from D
             4 11 13 9 0; ...      % edges from E
             5  6 10 0 0; ...      % edges from F
            10 11 12 0 0; ...      % edges from G
            12 13  0 0 0];         % edges from H
% coordinates
    coord = [ 5 6; ... % A
              3 9; ... % B
              1 6; ... % C
              3 1; ... % D
              6 2; ... % E
              6 8; ... % F
              9 7; ... % G
              10 2];   % H
    A = grAdjacency( node, cost, dir )
```

Large adjacency matrices usually contain very little data relative to their size. Consequently, to store them as a conventional `n` × `n` array is to waste most of the storage space, and may even cause memory problems for the processor. Recognizing this eventuality, MATLAB provides a special class, `sparse`, that stores a matrix as lists of row and column indices and the associated value. All normal array and matrix operations can be applied to a sparse matrix. The assumption is that any value not specifically allocated in a sparse matrix contains a zero. This is consistent with the earlier treatment of vectors and arrays where unknown values are filled with 0.

The function `grAdjacency(...)` that converts graph data from this legible form to the adjacency matrix form builds a sparse matrix by establishing three vectors of the same length—the row index, the column index, and the value of each point in the sparse matrix. The code to accomplish this is shown in Listing 17.8.

Listing 17.8 Creating an adjacency matrix

```
1. function A = grAdjacency( node, cost, dir )
   % compute an adjacency matrix.
   % it should contain the weight from one
   % node to another (0 if the nodes
   %                  are not connected)
2. [m cols] = size(node);
3. n = length(cost);
4. k = 0;
   % iterate across the edges
   %    finding the nodes at each end of the edge
5. for is = 1:n
6.     iv = 0;
7.     for ir = 1:m
8.         for ic = 1:cols
```

continued on next page

```
 9.              if node(ir, ic) == is
10.                  iv = iv + 1;
11.                  if iv > 2
12.                      error(...
                         'bad intersection matrix');
13.                  end
14.                  ij(iv) = ir;
15.              end
16.          end
17.      end
18.      if iv ~= 2
19.          error(sprintf( ...
             'didn't find both ends of edge %d', is));
20.      end
21.      t = cost(is);
22.      if dir(is) ~= -1
23.         k = k + 1;
24.         ip(k) = ij(1); jp(k) = ij(2); tp(k) = t;
25.      end
26.      if dir(is) ~= 1
27.         k = k + 1;
28.         ip(k) = ij(2); jp(k) = ij(1); tp(k) = t;
29.      end
30. end
31. A = sparse( ip, jp, tp );
```

In Listing 17.8:

Line 1: Shows a function consuming the `node`, `cost`, and `direction` arrays defined above. The locations of the nodes are needed only for plotting.

Lines 2–4: Show initial parameters, where `k` is the number of entries in the sparse matrix.

Line 5: Iterates down the list of edges.

Line 6: Initializes the number of nodes found connected to the edge.

Lines 7 and 8: Iterates across the nodes and columns of the node array, looking for the nodes connected to the edge.

Lines 9 and 10: When we find the edge value, we want to save that node index.

Lines 11–13: There can be only two ends to an edge; any more indicates a bad data set.

Line 14: Saves each end in the local variable `ij`.

Lines 18–20: When we finish the traversal, there must be a node at each end of the edge.

Line 21: Retrieves the cost of this edge.

Lines 22–25: Since bidirectional edges must be in the matrix twice, we check to see if the edge is bidirectional or forward, and enter the forward path in the sparse matrix.

Lines 26–28: Similarly, the reverse path is entered only if the edge is not forward.

Line 31: Constructs the sparse adjacency matrix.

17.2.4 Traversing Graphs

We first develop an iterative template for traversing a graph, then adapt this template to deal with some practical issues of path search, and finally, we mechanize this template for two search techniques. In its simplest form, the template for graph traversal is shown in Template 17.1.

In Template 17.1:

Line 1: This algorithm uses a queue to serialize the nodes to be considered. The first in/first out behavior of the queue causes the nearest nodes to emerge before the nodes farther away.

Line 2: Since all nodes have equal status on a graph, graph traversal must be provided with the node from which to begin the traversal. We enqueue that node to begin the traversal.

Lines 3 and 4: Show the typical `while` loop traversal, initializing a result.

Lines 5 and 6: Extract and process one node.

Lines 7 and 8: Traverse the edges from this node. There must be an indication for each problem of which order to use in selecting the edges to the children of the current path.

Lines 9 and 10: Because the graph can contain cycles, the mechanism for preventing the algorithm from becoming trapped requires that we enqueue only those nodes that have not already been visited.

Template 17.1 Template for graph traversal

```
 1. < create a queue >
 2. < enqueue the start node >
 3. < initialize the result >
 4. while < the queue is not empty >
 5.     < dequeue a node >
 6.     < operate on the node >
 7.     < for each edge from this node >
 8.         < retrieve the other node >
 9.         if < not already used >
10.               < enqueue the other node >
11.         end
12.     end
13. end
14. < return the result >
```

The choice of queue type governs the behavior of the traversal. If a simple queue is used, the traversal will happen like ripples on a pond from the starting node, touching all the nearest nodes before touching those farther away.

To illustrate the use of Template 17.1, we will print the names of all the nodes of the graph in Figure 17.5 in breadth-first order starting from node E, assuming that all edges are bidirectional. When choosing the edges to the next child node, the child nodes should be taken in alphabetical order as follows:

- Fortunately, because the adjacency matrix was built with the nodes in alphabetical order, the request will naturally produce them in alphabetical order.
- To make sure a node is not revisited, we will keep a list of the visited nodes, beginning with the start node.
- We find the children from the non-zero entries in the adjacency matrix—they are already in alphabetical order.
- We then traverse these children, adding to the queue those not found on the visited list and adding each to the visited list.

The script for this is shown in Listing 17.9.

Listing 17.9 Breadth-first graph traversal

```
makeGraph
   % Constructs an adjacency matrix
start = 5;
   % start is a node number (in this case, 'E')
   % Create a queue and
   % enqueue a path containing home
q = qEnq([], start);
   % initialize the visited list
visited = start;
   % initialize the result
fprintf('trace: ')
   % While the queue is not empty
while ~isempty(q)
   % Dequeue a path
    [q thisNode] = qDeq(q);
   % Traverse the children of this node
    fprintf('%s - ', char('A'+thisNode-1) );
    children = find(A(thisNode,:) ~= 0);
    for aChild = children
    % If the child is not on the path
        if ~any(aChild == visited)
        % Enqueue the new path
            q = qEnq(q, aChild);
        % add to the visited list
            visited = [visited aChild];
        end % if ~any(eachchild == current)
    end % for eachchild = children
end % while q not empty
fprintf('\n');
```

In Listing 17.9:

Line 1: Invokes the script in Listing 17.7 to build the adjacency matrix.

Line 2: The user-defined starting node—E.

Line 3: Enqueues the starting node on a new queue.

Line 4: Initializes the visited list.

Line 5: Initializes the result—in this case, a printout.

Line 6: Shows the `while` loop.

Line 7: Dequeues a node.

Line 8: In this case, processing the node involves printing its label and a dash.

Line 9: The non-zero values from the row in the adjacency matrix corresponding to this edge give us the children of this node.

Line 10: Traverses the children.

Lines 11–13: If they are not already on the visited list, enqueue them and put them on it.

Line 17: Completes the result when the queue is empty.

The results from this script are as follows:

```
trace: E - A - D - G - H - B - C - F –
```

which, referring to Figure 17.5, is a "ripple" traversal from node E outward, taking children in alphabetical order as specified.

17.2.5 Searching Graphs

In order to search a graph rather than traverse it, we have to make the following changes to Template 17.1:

- Since we need to return the complete path between the start and target, the queue has to contain that path
- Rather than use a global visited list, we can use the path taken from the queue to determine whether a child node is causing a cycle
- The order of the nodes on the path has the starting node at the front of the path and the new node at the end

17.2.6 Breadth-First Search (BFS)

Frequently, we actually need the path with the smallest number of nodes between the starting and ending nodes. For example, because changing trains involves walking and waiting, the best path on a railway map (such as the one shown in Figure 17.4) is that with the fewest changes, even if the resulting path is longer.

To search for the path with the least nodes, we need a function that performs a Breadth-First Search (BFS) on a graph. In order to be able to use MATLAB's built-in graph plotting program, the answer returned should be an adjacency matrix showing the computed path. The function to perform this search is shown in Listing 17.10.

In Listing 17.10:

Line 1: Shows a function consuming an adjacency matrix, the starting and destination node indices.

Line 2: Initializes the queue.

Line 3: Shows the usual `while` loop.

Listing 17.10 Breadth-first graph search

```
 1. function D = grBFS(A, home, target)
% A - an adjacency matrix
% home — starting node index
% target — ending node index
% produce the adjacency matrix D of the
% breadth-first path from home to target.
% If no such path exists, return []
% Create a queue and enqueue a path containing home
 2. q = qEnq([], home);
% While the queue is not empty
 3. while ~isempty(q)
    % Dequeue a path
 4.     [q current] = qDeq(q);
        % If this path reaches the destination, stop
 5.     if current(end) == target   % success exit
 6.         D = sparse([0]);
            % build a new adjacency matrix
            % from the path
 7.         for ans = 1:length(current)-1
 8.             D(current(ans), current(ans+1)) = 1;
 9.         end
10.         return;   % exit the function
11.      end  % if current == target
        % Traverse the children of the last node
12.     thisNode = current(end);
13.     children = find(A(thisNode,:) ~= 0);
14.     for thisChild = children
            % If the child is not on the path
15.         if ~any(thisChild == current)
                % Enqueue the new path
16.             q = qEnq(q, [current thisChild]);
17.         end % if ~any(thisChild == current)
18.     end % for thisChild = children
19. end % while q not empty
% If we reach here we never found a path
20. D = [];
```

Line 4: The queue now contains a vector of the node indices on the current path.

Line 5: If the node dequeued (`current(end)`) is the target, the function creates a new adjacency matrix representing the path from the home node to the target.

Line 6: Creates an empty sparse matrix.

Lines 7–9: Add to it the edges between each node in the path.

Line 10: Exits the function.

Line 12: Otherwise, recovers the last node.

Line 13: Retrieves its children.

Lines 14–18: Traverse the children as before, checking for their presence on the current path. When a child is enqueued, it is appended to the end of the current path and the whole path is enqueued.

The BFS path from A to H is shown in Figure 17.6. Note that it found the path with the least number of nodes.

17.2.7 Dijkstra's Algorithm

Although the minimal number of nodes is sometimes the right answer, frequently there is a path that uses more nodes but has a smaller overall cost. This is evident from a quick glance at Figure 17.5: the path A–F–G–H has a lower cost than the A–E–H path found by the BFS algorithm, which actually ignores the edge weights. Many algorithms exist for finding the optimal path through a graph. Here we illustrate the algorithm attributed to the Dutch computer scientist Dr. Edsger Dijkstra. Perhaps it is not the most efficient algorithm, but for our purposes, this approach has the virtue of being a minor extension to the `while` loop algorithms used for BFS.

- The major differences arise from the use of a priority queue in place of the normal queue used in the BFS algorithm. As previously noted, priority queues differ from basic queues only to the extent that the `enqueue` method puts the data in order, rather than at the tail.

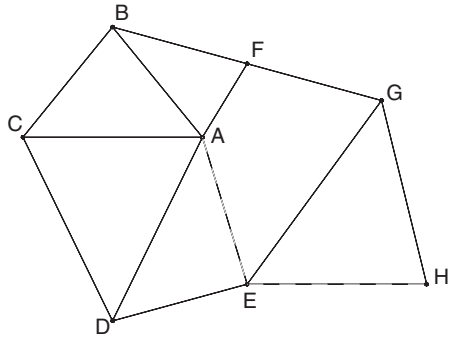


Figure 17.6 *Breadth-first result*

- The ordering criterion required by the algorithm is to place the paths in increasing order of path cost (total weight).

The objects contained in the priority queue need to contain not only the path used for BFS, but also the total path weight. For this we will use a structure with fields `nodes` and `key`, and implement a small collection of helper functions. The priority queue and its helper function to compare two structures with a key are shown in Listings 17.2 and 17.3; `Path(...)` to construct a new entry and `pthGetLast(...)` to get the index of the last node on the path are shown in Listing 17.11.

In Listing 17.11:

Line 1: Shows a function to construct a path structure from its components.

Lines 2 and 3: Build the structure.

Line 4: Shows a function to retrieve the last node from a path.

Line 5: Since the path nodes start at the path origin, the last entry is the node we need.

The function that performs Dijkstra's algorithm is shown in Listing 17.12.

Listing 17.11 Helper functions for Dijkstra's algorithm

```
1. function ret = Path(nodes, len)
   % Path constructor
2.       ret.nodes = nodes;
3.       ret.key = len;

4. function ret = pthGetLast(apath)
   % Returns number of last node on a path
5.     ret = apath.nodes(end);
```

Listing 17.12 Code for Dijkstra's algorithm

```
 1. function D = grDijkstra(A, home, target)
% A - an adjacency matrix
% home and target - node indices.
% Create a priority queue
% Enqueue a path with containing home with length 0
 2. pq = pqEnq([], Path(home, 0));
 3. while ~isempty(pq)
    % Dequeue a path
 4.     [pq current] = qDeq(pq);
    % If this path is to the destination quit
 5.     if pthGetLast(current) == target
 6.         D = sparse(0);
 7.         answer = current.nodes;
 8.         for ans = 1:length(answer)-1
 9.             D(answer(ans), answer(ans+1)) = 1;
10.         end
```

continued on next page

```
11.          return;
12.      end  % if last(current) == target
    % put the children of the last node in the path
13.      endnode = pthGetLast(current);
14.      children = A(endnode,:);
15.      children = find(children ~= 0);
16.      for achild = children
17.          len = A(endnode, achild);
         % If the child is not on the path
18.          if ~any(achild == current.nodes)
             % Add it to the path
19.              clone = Path( [clone.nodes achild] ...
20.                           current.key + len;
             % Enqueue the new path to that length
21.              pq = pqEnq(pq, clone);
22.          end % if ~any child == current.nodes
23.      end % for achild = children
24. end % if pq not empty
% If we reach here we never found a path
25. D = [];
```

In Listing 17.12:

Line 1: Shows a function consuming an adjacency matrix, the starting and destination node indices.

Line 2: Initializes the priority queue with a starting node and zero cost.

Line 3: Shows the usual `while` loop.

Line 4: Shows that the queue now contains a path structure.

Lines 5–12: If the node dequeued is the target, the function creates a new adjacency matrix representing the path from the home node to the target.

Line 13: Otherwise, it recovers the last node.

Line 14: Retrieves its children.

Lines 15–23: Traverse the children as before, checking for their presence on the current path. When a child is enqueued, it is appended to the end of the current path, and the whole path is enqueued.

The optimal path from A to H is shown in Figure 17.7. Note that it found the path with the least cost.

17.2.8 Testing Graph Search Algorithms

The script that develops both search path solutions is shown in Listing 17.13. It requests the starting and ending node letters from the user and then incrementally plots the original graph, the BFS solution, and the optimal solution. The pause between plots allows the individual paths to be examined. Without a parameter, `pause` waits for any keyboard character.

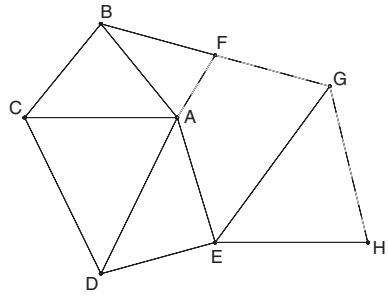


Figure 17.7 *Dijkstra's result*

In Listing 17.13:

Line 1: Constructs the adjacency matrix.

Line 2: Forces your way into the `while` loop once.

Line 3: Keeps going until the user enters an illegal starting node.

Lines 4 and 5: Plot the graph using the adjacency matrix and the coordinates stored for the nodes.

Lines 6–10: Put letters by the nodes.

Line 11: Sets the axes.

Lines 12 and 13: Get the starting node.

Lines 14–16: If valid, get the target node.

Line 17: Shows the original graph and waits for a character.

Lines 18–20: Compute and plot the BFS solution.

Lines 21–24: Compute and plot the optimal solution.

Lines 25 and 26: Repeat as necessary.

Listing 17.13 Testing graph search algorithms

```
1.     makeGraph;  % call script to make the graph:
2.     start = 1;
3.     while start > 0
4.         gplot(A, coord, 'ro-')
5.         hold on
6.         for index = 1:length(coord)
7.             str = char('A' + index - 1);
8.             text(coord(index,1) + 0.2, ...
9.                  coord(index,2) + 0.3, str);
10.        end
11.        axis([0 11 0 10]); axis off; hold on
12.        ch = input('Starting node: ','s');
13.        start = ch - 'A' + 1;
14.        if start > 0
15.            ch = input('Target node: ','s');
16.            target = ch - 'A' + 1;
17.            disp('original graph'); pause
18.            D = grBFS( A, start, target);
```

continued on next page

```
            gplot(D, coord, 'go-')
            disp('BFS result'); pause
            D = grDijkstra( A, start, target);
            gplot(D, coord, 'bo-')
            disp('Optimal result'); pause
            hold off
        end
    end
```

17.3 Engineering Application—Route Planning

One obvious application is to use these algorithms for route planning in a city. Figure 17.8 shows the results from a Dijkstra's path search on a small city street map. Although this particular search used street length as the cost of each edge, it is not difficult to visualize a system architecture whereby individual vehicles could receive real-time data feeds indicating the actual speed of vehicles on the streets and wait times at intersections. The cost of using a street in terms of travel times could then be factored into the calculation. An advanced mapping system in vehicles would then be able to recommend alternate routes to avoid bottlenecks on city streets, thereby saving significant amounts of time and energy now lost idling in city traffic.

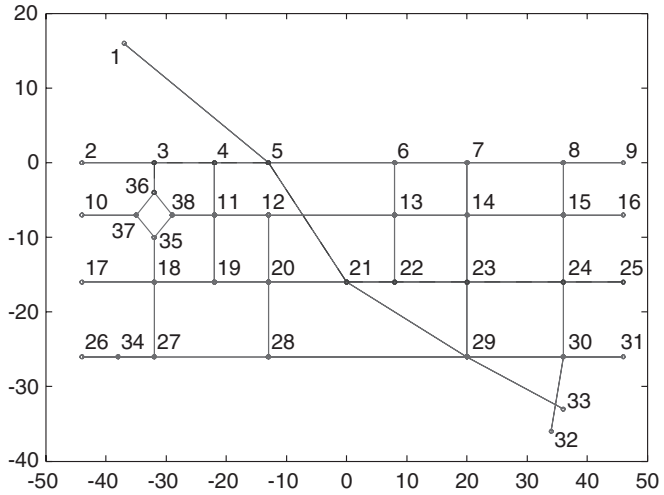


Figure 17.8 *Street map results*

Chapter Summary

This chapter demonstrated effective algorithms for finding good paths through a graph, and included the following:

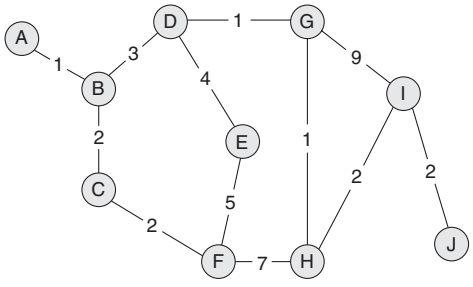
- How to construct and use queues and priority queues as the underlying mechanism for graph traversal
- The basic use of an adjacency matrix for defining a graph
- How to use a queue to develop traversal and search results on any kind of graph

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`break`	A command within a loop module that forces control to the statement following the innermost loop	17.1.3
`any()`	True if any of the values in `a`, a logical vector, is true	17.2.4
`class(obj)`	Determines the data type of an object	17.1.3
`gplot(A,coord)`	Plots a graph from an adjacency matrix and an array of coordinates	17.2.8
`isa(obj, str)`	Tests for a given data type	17.1.3
`sparse(r,c,value)`	Builds a sparse matrix from vectors of row and column indices and the associated value	17.2.3

Programming Projects

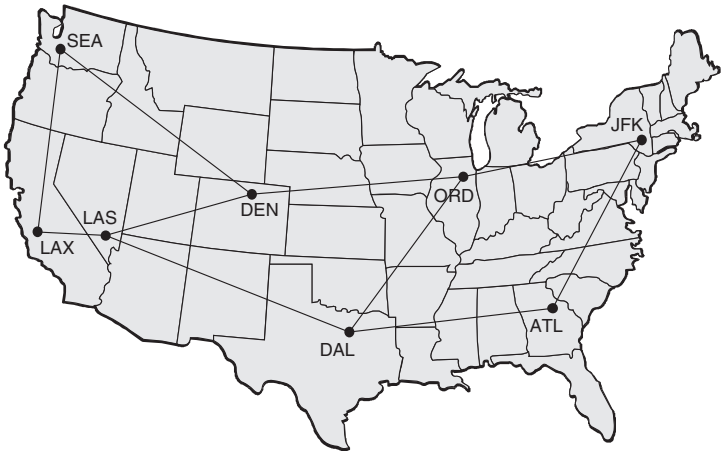
1. Consider the graph shown in the figure below.



   a. Write a script to construct this graph.

b. Compute the shortest path and the path with least nodes between nodes A and J.
c. Verify the answer by manually executing the BFS and Dijkstra's algorithms.

2. Consider the graph shown in the figure below.



a. Obtain the route pattern for a small airline.
b. Write the code to build a model of their route map.
c. Write a script that asks the user for an origin and destination and computes the shortest route and the route with the fewest connections.