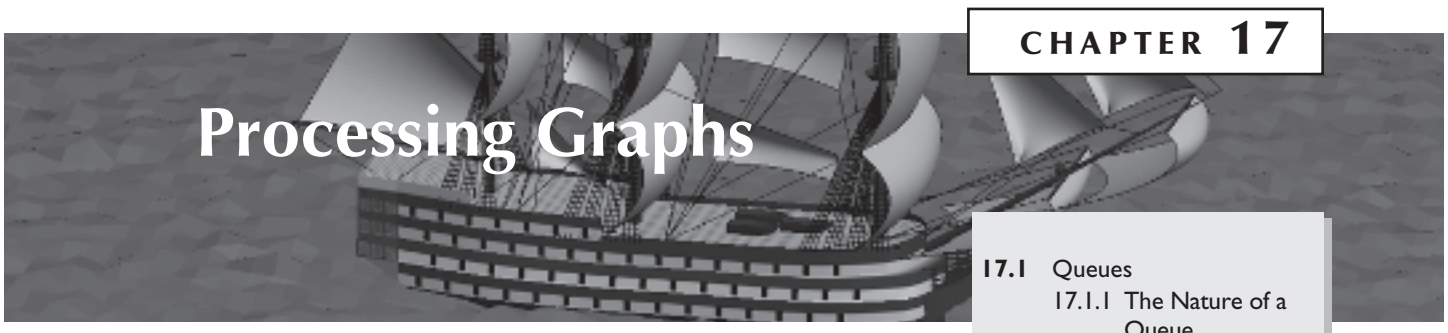
CHAPTER 17

Processing Graphs



Chapter Objectives

This chapter demonstrates algorithms that solve two problems: finding the minimum spanning tree for a graph and finding the best path 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 queues
- 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, structure arrays, and cell arrays—have essentially been collections whose elements are linearly related to each other by being organized in rows and columns. However, practical engineering frequently meets data that are not organized so easily. Graphs are one such data set. The ultimate goal of this chapter is to discuss this most general form of data structure. We need first to resolve the semantic problem of the name "graph." We typically think of a graph as a plot. However, in computer science, a **graph** is a collection of **nodes** connected by **edges**. A street map might be a useful mental model of a graph where the streets are the edges and the intersections are the nodes.

To process graphs effectively, we must first consider two simpler concepts: **queues** in general and **priority queues** in particular.

17.1 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. A cell array, therefore, would be a good underlying structure upon which to build queue behavior.

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 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 and can be extended or shortened without any apparent effort. If we establish a queue using a cell array, the implementation of the above behavior is trivial:

- `enqueue` concatenates data at the end of the cell array
- `dequeue` 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

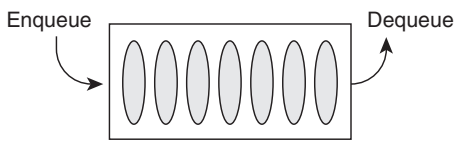


Figure 17.1 *A queue*

Listing 17.1 Enqueue and dequeue functions

```
1. function q = qEnq(q, data)
       % enqueue onto a queue
2.     q = [q {data}];
3. end
4. function [q ans] = qDeq(q)
       % dequeue
5.     ans = q{1};
6.     q = q(2:end);
7. end
```

the queue rather than the back to effectively "jump in line." There are implementations beyond the scope of this text that use object-oriented 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 in a cell (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 the printer where you wait an hour for one page while someone prints large sections of an encyclopedia and you wonder why the print queue can't put really small jobs ahead of really large jobs.

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 add in order, there must be a means of comparing two objects. Here, we use the function `is_before` that generally should be able to compare any two objects.

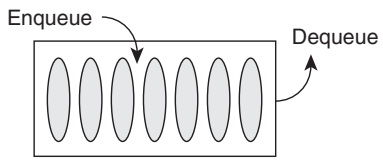


Figure 17.2 *A priority queue*

In this implementation, it is sufficient to be able to compare numbers or structures that contain either the fields `key` or `NaN`. Clearly, this can be extended as necessary to compare any two objects.

The code for `is_before` 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.

Listing 17.2 Comparing two objects

```
1. function ans = is_before(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
20. 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 is_before(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)];
12. end
```

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 `is_before(...)` to compare objects for a priority queue is shown in Listing 17.3.

In Listing 17.3:

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

Lines 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 that the `item` to insert goes before. 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

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
6. end
```

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.

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.

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.

Listing 17.5 Converting any object to a string

```
function str = toString(item)
% turn any object into its string representation
    if isa(item, 'char')
        str = [''' item '''];
    elseif isa(item, 'double')
        if length(item) == 1
            str = sprintf('%g', item );
        else
            str = '[';
            for in = 1:length(item)
                str = [str ...
                    sprintf(' %g', item(in) ) ];
            end
            str = [str ' ]'];
        end
    elseif isa(item, 'struct')
        if isfield(item, 'name')
            str = item.name;
        else
            nms = fieldnames(item);
            str = [];
            for in = 1:length(nms)
                nm = nms{in};
                str = [str nm ': ' ...
                    toString(item.(nm)) 13];
            end
        end
    else
        str = 'unknown data';
    end
end
```

Listing 17.6 Testing the queues

```
q = [];
for ix = 1:10
    q = qEnq(q, ix);
end
CAToString(q)
[q ans] = qDeq(q);
fprintf('dequeue -> %d leaving \n%s\n', ...
    ans, CAToString(q) );
fprintf('peek at queue -> %d leaving \n%s\n', ...
    q{1}, CAToString(q) );
pq = [];
for ix = 1:10
    value = floor(100*rand);
    fprintf(' %g:', value );
    pq = pqEnq(pq, value );
end
fprintf('\npriority queue is \n%s\n', ...
    CAToString(pq) );
```

In Listing 17.6:

Line 1: Initializes a queue.
Lines 2–4: Enqueue 10 numbers.
Line 5: Displays the resulting queue.
Lines 6–8: Dequeue and print one value and the remaining queue.
Lines 9 and 10: Peek at the head of the queue and verify that we have not changed its contents.
Line 11: Creates a priority queue.
Lines 12–16: Enqueue 10 random integers.
Lines 17 and 18: List the queue to show that they were enqueued in order.

A serious reader can verify that this indicates correct queue behavior.

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. 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 may 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

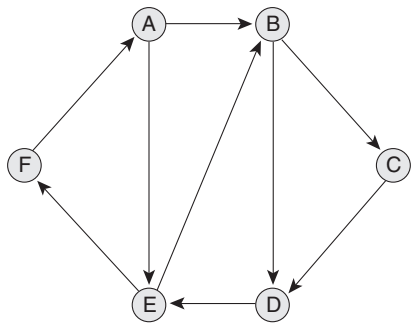


Figure 17.3 *A simple graph*

Figure 17.4 *A simple street map*

directional but not weighted. Graphs occur frequently in everyday life, as illustrated by the street map shown in Figure 17.4. Street maps can be conveniently represented as graphs where intersections are the nodes and streets are the edges. Streets can be directional (one-way), and they may have weights associated with them—either the transit time (a function of the length of the street and its speed limit) or with access to real-time traffic information, a more complex estimate of the transit time.

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 as well as the place to stop.
- 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.

For a simple, consistent example, consider the graph shown in Figure 17.5. We will use this simple example to demonstrate minimum spanning trees (MSTs) and finding paths through the graph.

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

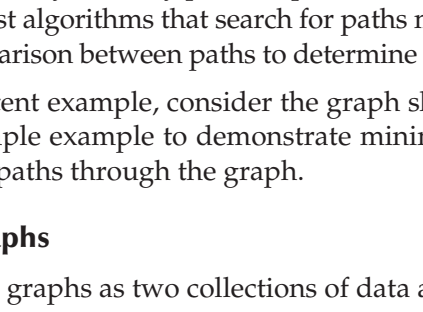


Figure 17.5 *A weighted graph*

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 is:

```
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
```

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, the MATLAB language 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 arrays of nodes, costs, and direction to the adjacency matrix form builds a sparse

Listing 17.7 Constructing a simple graph

```
   % edge weights
 1. cost = [2 2 2 2 2 3 3 3 3 1 2 1 3];
   % edge directions
 2. dir = [2 2 2 2 2 2 2 2 2 2 2 2 2];
   % connectivity
 3. node = [ 1 2 3 4 5; ...   % edges from A
 4.          1 6 7 0 0; ...   % edges from B
 5.          2 7 8 0 0; ...   % edges from C
 6.          3 8 9 0 0; ...   % edges from D
 7.          4 11 13 9 0; ... % edges from E
 8.          5 6 10 0 0; ...  % edges from F
 9.         10 11 12 0 0; ... % edges from G
10.         12 13 0 0 0];     % edges from H
   % coordinates
11. coord = [ 5 6; ... % A
12.           3 9; ... % B
13.           1 6; ... % C
14.           3 1; ... % D
15.           6 2; ... % E
16.           6 8; ... % F
17.           9 7; ... % G
18.          10 2];    % H
19. A = grAdjacency( node, cost, dir )
```

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
 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 );
32. end
```

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.

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: Iterate 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

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.

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 >
```

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.

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:

- To make sure that 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—because of the way we built the 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.

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.

Listing 17.9 Breadth-first graph traversal

```
 1. makeGraph
    % Constructs an adjacency matrix
 2. start = 5;
    % start is a node number (in this case, 'E')
    % Create a queue and
    % enqueue a path containing home
 3. q = qEnq([], start);
    % initialize the visited list
 4. visited = start;
    % initialize the result
 5. fprintf('trace: ')
    % While the queue is not empty
 6. while ~isempty(q)
        % Dequeue a path
 7.     [q thisNode] = qDeq(q);
        % Traverse the children of this node
 8.     fprintf('%s - ', char('A'+thisNode-1) );
 9.     children = find(A(thisNode,:) ~= 0);
10.     for aChild = children
            % If the child is not on the path
11.         if ~any(aChild == visited)
                % Enqueue the new path
12.             q = qEnq(q, aChild);
                % add to the visited list
13.             visited = [visited aChild];
14.         end % if ~any(eachchild == current)
15.     end % for eachchild = children
16. end % while q not empty
17. fprintf('\n');
18.
```

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 the visited list.

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 breadth-first traversal from node E outward, taking children in alphabetical order as specified.

17.2.5 Searching Graphs

Rather than traversing a graph, we frequently need to know whether a graph contains a specific node. Template 17.1 is easily modified to include a

test to see if the current node is the one sought and to exit with success when it is, leaving the existing exit for the failure case.

17.3 Minimum Spanning Trees

We will consider two practical algorithms commonly found in a large range of engineering disciplines. The MST of a graph is used, for example, to calculate the shortest cable necessary to connect all the houses in a subdivision. Unlike path search, the second algorithm to be discussed later, a spanning tree may have multiple branches essentially modeling side streets in the subdivision.

While there may be a large number of spanning trees, and there may be mutiple MSTs in different configurations, they should all have the same total length. We will consider one of the two major algorithms for computing a MST—that commonly referred to as Prim's algorithm. The other, Kruscal's, is similar and will not be covered here.

Prim's algorithm finds the subset of the edges of the graph that connect every node exactly once and whose total cost is less than that of any other spanning tree.

Technical Insight 17.1

According to Wikipedia, this algorithm was developed in 1930 by Czech mathematician Vojtech Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore, it is also sometimes called the DJP algorithm, the Jarník algorithm, or the Prim–Jarník algorithm.

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all the vertices. The resulting tree, V, is a collection of edges. It needs another collection, N, the nodes currently included in the MST.

Specifically, given a graph as defined above, Prim's algorithm proceeds as follows:

- Initialize the result V as an empty vector and N, the included nodes $= \{x\}$, where x is an arbitrary node chosen from the graph
- Repeat the following while there are available edges:
  - Choose an edge (u, v) with minimal weight such that u is in N and v is not (if there are multiple edges with the same weight, any of them may be picked)
  - Add v to N, and (u, v) to V.
- Report the contents of V as the resulting MST.

Listing 17.10 shows the code that extracts MST from our sample graph.

In Listing 17:10

Lines 1 and 2: Invoke the script that builds the graph.

Listing 17.10 Prim's Algorithm to compute a MST

```
 1. makeGraph
 2. start = 1;
 3. gplot(A, coord, 'ro-')
 4. hold on
 5. for index = 1:length(coord)
 6.      str = char('A' + index - 1);
 7.      text(coord(index,1) + 0.2, ...
 8.           coord(index,2) + 0.3, str);
 9. end
10. axis([0 11 0 10]); axis off; hold on
11. N = start;
12. running = true;
13. result = sparse([0]);
14. while running
        % find the smallest edge
15.     best = 10000;
16.     running = false;
17.     for ndx = 1:length(N)
18.         node = N(ndx);
19.         next = find(A(node,:) > 0);
20.         for nxt = 1:length(next)
21.             nxtn = next(nxt);
22.             if ~any(N == nxtn)
23.                 running = true;
24.                 if A(node, nxtn) < best
25.                     best = A(node, nxtn);
26.                     from = node;
27.                     to = nxtn;
28.                 end
29.             end
30.         end
31.     end
32.     if running
33.         N = [N to];
34.         result(from, to) = 1;
35.     end
36. end
37. gplot(result, coord, 'gx--')
```

Line 2: Sets the starting node (A).

Line 3: Plots the basic graph structure using the built-in `gplot(...)` function.

Lines 4–10: Add plot axes and labels for all the nodes.

Lines 11–13: Initialize the visited node list, `N`, a sparse matrix to store the result and the while condition.

Lines 14–37: Repeat as long as there are nodes to process.

Line 16: Establishes a large initial best node.

Line 17: Assumes failure to find a node.

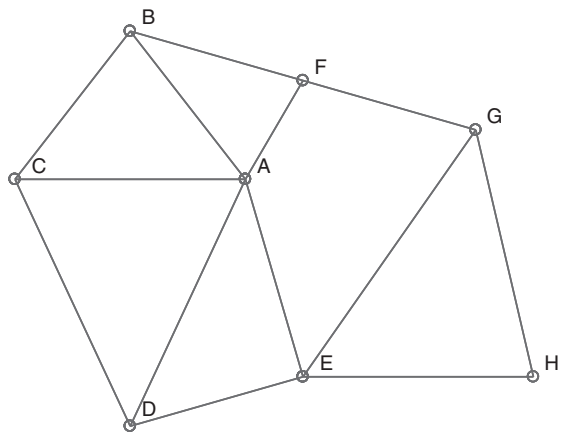


Figure 17.6 *MST result*

Line 18: Looks through all the visited nodes.

Line 19: Extracts a node.

Line 20: Extracts from the adjacency matrix the edges from that node. The indices with non-zero entries are the nodes to which the edge connects; the value in the adjacency matrix is the weight or cost of that edge.

Lines 21 and 22: Examine each of the edges for the node it reaches (`nxtn`).

Line 23: Only continues if this is not already on the visited list.

Lines 24–29: Report that an edge has been found, determine if it is shorter than the previous best, and if so store the new best value and the nodes at each end of the edge.

Lines 33–36: Add the new node to the visited list and the edge to the result array.

Line 38: Plots the MST as green dashed lines.

Figure 17.6 shows the MST resulting from this script. Note that if a different starting node is used, the specific tree might be different but its total edge length will be the same.

17.4 Finding Paths through a Graph

This section discusses three algorithms for finding a path from one node on the graph to another. The first two algorithms exhaustively search the graph to find the absolute best path between node pairs by different criteria. The third is one of many approximation algorithms typically used to compute a good enough route in circumstances where an exact solution is not feasible.

17.4.1 Exact Algorithms

In order to find a path 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

17.4.2 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 street map in Figure 17.4) is that with the fewest changes, even if the resulting path is longer. The algorithm is based on Template 17.1 with the changes noted above.

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 the 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 17.11.

Listing 17.11 Breadth-first graph search

```
function D = grBFS(A, home, target)
    q = qEnq([], home);
    while ~isempty(q)
        [q current] = qDeq(q);
        if current(end) == target % success exit
            D = sparse([0]);
            for ans = 1:length(current)-1
                D(current(ans), current(ans+1)) = 1;
            end
            return; % exit the function
        end % if current == target
        thisNode = current(end);
        children = find(A(thisNode,:) ~= 0);
        for thisChild = children
            if ~any(thisChild == current)
                q = qEnq(q, [current thisChild]);
            end % if ~any(thisChild == current)
        end % for thisChild = children
    end % while q not empty
    % if we reach here we never found a path
    D = [];
end
```

In Listing 17.11:

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

Line 2: Initializes the queue.

Line 3: Repeats to Line 19 until the queue is empty.

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 17.7. Note that it found the path with the least number of nodes.

17.4.3 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

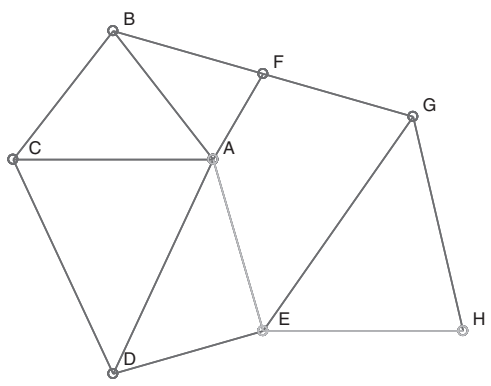


Figure 17.7 *Breadth-first result*

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 algorithm described in Template 17.1.

- 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.
- 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 helper functions to build a structure with a key and extract the key of the last path entry are shown in Listing 17.12.

In Listing 17.12:

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

Lines 2 and 3: Build the structure.

Line 5: Shows a function to retrieve the last node from a path.

Line 6: 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 17.13.

In Listing 17.13:

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

Listing 17.12 Helper functions for Dijkstra's algorithm

```
1. function ret = Path(nodes, len)
   % Path constructor
2.     ret.nodes = nodes;
3.     ret.key = len;
4. end
5. function ret = pthGetLast(apath)
   % Returns number of last node on a path
6.     ret = apath.nodes(end);
7. end
```

Listing 17.13 Code for Dijkstra's algorithm

```
 1. function D = grDijkstra(A, home, target)
 2.     pq = pqEnq([], Path(home, 0));
 3.     while ~isempty(pq)
 4.         [pq current] = qDeq(pq);
 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
11.             return;
12.         end % if last(current) == target
13.         endnode = pthGetLast(current);
14.         children = A(endnode,:);
15.         children = find(children ~= 0);
16.         for achild = children
17.             len = A(endnode, achild);
18.             if ~any(achild == current.nodes)
19.                 clone = Path( [clone.nodes achild] ...
20.                     current.key + len;
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 = [];
26. end
```

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

Line 3: Continues repeating until the queue is empty.

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 17.8. Note that it found the path with the least cost.

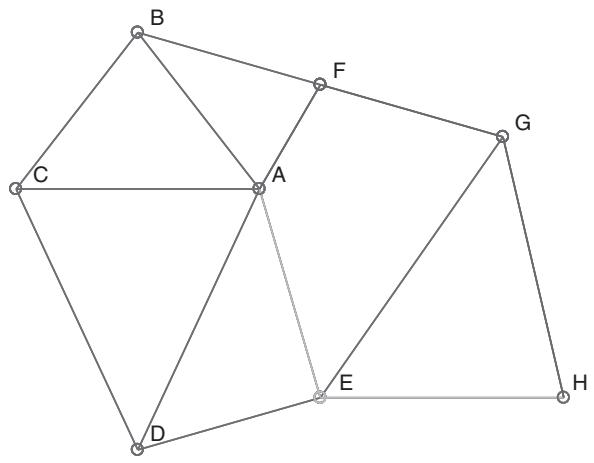


Figure 17.8 *Dijkstra's result*

17.4.4 Approximation Algorithm

When a graph is very large, the computation complexity of the exact solutions (roughly $O(N^2)$) becomes unmanageable. The A* algorithm is one of many popular approximation techniques that will produce a solution, but is not guaranteed to produce the best, and its computational complexity is roughly O(N). This algorithm is quite simple:

1. Beginning at the starting node, it evaluates the result of traveling along each of the feasible edges to an adjacent node (eliminating cyclic paths). The evaluation takes the form of summing the cost of that edge and an estimate of the cost from that node to the destination. On a street map, for example, the estimated cost of each step would be the length of the edge and the straight-line distance from the new node to the destination.
2. It selects the step with the least cost, adds the node reached to the path, and repeats step 1 until the destination is reached.
3. Back-tracking is sometimes necessary if a node is reached from which there are no feasible paths, such as driving into a dead end street.
4. Complete failure is also possible, as it is for the other algorithms, if no physical path exists between the origin and destination nodes.

Listing 17.14 shows the code that implements the A* algorithm. Notice that some additional information is necessary to effectively compute the estimated cost from a node to the destination. In our example, we can use the location of each node, but in general, that location may not be readily available.

Listing 17.14 Code for A* algorithm

```
 1. function D = A_Star(A, home, target, coord)
        % initial path
 2.     current = home;
 3.     visited = home;
 4.     while current(end) ~= target
 5.         thisNode = current(end);
            % get possible paths from here
 6.         children = find(A(thisNode,:) ~= 0);
 7.         best = inf;
 8.         node = -1; % no node seleected yet
 9.         for thisChild = children
10.             if ~any(thisChild == visited)
11.                 edgeCost = A(thisNode, thisChild);
12.                 estimate = dist(thisChild, target, coord);
13.                 cost = edgeCost + estimate;
14.                 if cost < best
15.                     best = cost;
16.                     node = thisChild;
17.                 end
18.             end % if ~any(thisChild == current)
19.         end % for thisChild = children
20.         if node == -1
                % dead end -> back up one
21.             current = current(1:end-1);
22.             if length(current == 0)
23.                 error('path failed')
24.             end
25.         else
26.             current = [current node];
27.             visited = [visited node]; %
28.         end
29.     end
30.     D = sparse([0]);
31.     for it = 1:length(current)-1
32.         D(current(it), current(it+1)) = 1;
33.     end
34. end
35. function res = dist(a, b, coord)
36.     from = coord(a,:);
37.     to = coord(b,:);
38.     res = sqrt((from(1)-to(1)).^2 + (from(2)-to(2)).^2);
39. end
```

In Listing 17.14:

Lines 2 and 3: We will maintain two lists—the current path and the visited list indicating all the nodes that have been visited. This provides for the case when back-tracking is necessary to avoid revisiting the dead end.

Lines 4–29: Continue until the target node is reached.

Lines 5 and 6: Find the nodes that can be reached from the current node.

Lines 7 and 8: Initialize the storage for the best next step.

Lines 9–19: Iterate across all possibilities.

Line 10: Only considers nodes not on the visited list.

Lines 11–13: The cost of this step is the sum of the actual cost of one step and the estimate of the remaining cost given in this case by the distance between the nodes (invoking the helper function at lines 35–39).

Lines 14–17: Check for improvement in the best cost.

Lines 20 and 21: Check for a dead end.

Lines 22 and 23: Check for total failure—we have backed up beyond the starting node.

Lines 26 and 27: Add a successful node to the current path (from which it might later be remove by backing up) and the visited nodes from which it is never removed.

Lines 30–33: Prepare the results as a sparse matrix for plotting.

Lines 35–39: Helper function calculating the distance between the specified points.

The A* path from A to H is shown in Figure 17.9. Note that in this simple case, it found the same path as the BFS, but that is not necessarily the case in a more complex test.

17.4.5 Testing Graph Search Algorithms

The script that develops both search path solutions is shown in Listing 17.15. It requests the starting and ending node letters from the user and then

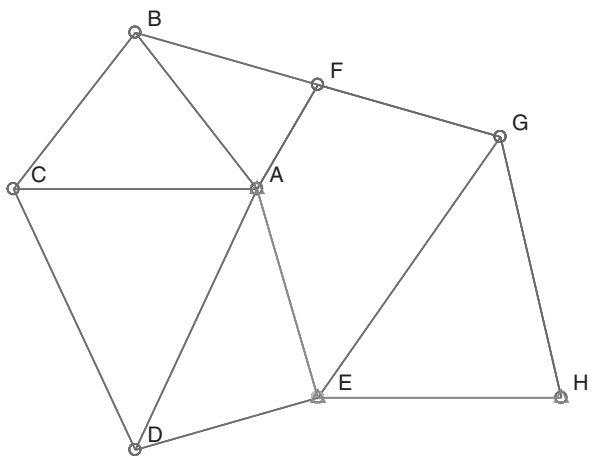


Figure 17.9 *A* result*

Listing 17.15 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);
19.         gplot(D, coord, 'go-')
20.         disp('BFS result'); pause
21.         D = grDijkstra( A, start, target);
22.         gplot(D, coord, 'bo-')
23.         disp('Optimal result'); pause
24.         D = A_Star( A, start, target, coord);
25.         gplot(D, coord, 'm^-')
26.         disp('A* result'); pause
27.         hold off
28.     end
29. end
```

incrementally plots the original graph, the BFS solution, the optimal solution, and the A* solution. The pause between plots allows the individual paths to be examined. Without a parameter, `pause` waits for any keyboard character.

In Listing 17.15:

Lines 1–11: Initialize the experiment as before.
Lines 12 and 13: Get the starting node.
Lines 15 and 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 21–23: Compute and plot the optimal solution.
Lines 24–26: Compute and plot the A* solution.
Lines 27 and 28: Repeat as necessary.

17.5 Engineering Applications

Many practical engineering problems can be characterized as graph search problems.

17.5.1 Simple Applications

MSTs are used by utility companies to find the least amount of cable that must be used to wire a subdivision.

Approximate path finding is used, for example, in navigation systems that use GPS to find the current position of the vehicle and an approximate algorithm like A* to determine the route to a destination.

Exact path finding is used to optimize the flight profile of commercial aircraft outside FAA-managed air space and can save as much as 10% of the fuel burned on every flight.

17.5.2 Complex Extensions

In addition to the obvious examples above, consider these examples:

- designing printed circuit boards is a complex extension of path finding
- stresses in a redundant structure like an aircraft wing seek a path that is in some sense optimal, and
- the "traveling salesperson problem" is an unpleasant extension of path finding in which the objective is to find the shorted linear path that connects all of the nodes of a graph visiting each exactly once. For example, designing routes for garbage collection or school buses.

Each of these belongs to a large class of problems called N-P Complete problems, a continued topic of research in many communities.

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
- Prim's algorithm for finding the minimum spanning tree of a graph
- Breadth-first and Dijkstra's algorithms for finding exact paths through a graph
- The A* algorithm for finding approximate paths that are "good enough"

Programming Project

1. We would like to validate the assertion that the street map is designed to have at most two train changes between any pair of stations. Using the methodology of Section 17.2.3 and the picture in Figure 17.4, construct a graph representing the major routes in that system. You will not need all the stations identified for this exercise—only one station per track segment between transfer stations.
   a. Write a function that will determine the number of train changes to travel between any pair of stations using a breadth-first search to minimize the number of changes.
   b. Iterate across every pair of stations and find the station pair with the maximum number of train changes.