Chapter 9

Graphs

Often we need to model relationships that can be expressed using a set of pairs. Examples include distances between points on a map, links in a communications network, precedence constraints between tasks, and compatibility of items or people. In some cases, the relationship is symmetric; e.g., if $A$ is compatible with $B$, then $B$ is compatible with $A$. In other cases, the relationship is asymmetric; e.g., the requirement that $A$ precedes $B$ is not the same as the requirement that $B$ precedes $A$. All of these relationships can be modeled using graphs. Having modeled the relationship, we can then apply graph algorithms for extracting such information as a shortest path between two points or a valid ordering of tasks.

There are two kinds of graphs, depending on whether the relationship to be modeled is symmetric or asymmetric. For symmetric relationships, we define an undirected graph to be a pair $(V, E)$, where $V$ is a finite set of vertices (or nodes) and $E$ is a set of 2-element subsets of $V$. We refer to the elements of $E$ as edges. We can represent undirected graphs pictorially as in Figure 9.1, where vertices are denoted by circles and edges are denoted by line segments or curves connecting their constituent vertices. We often say that an edge $\{u, v\}$ is incident on vertices $u$ and $v$, and that $u$ and $v$ are therefore adjacent.

For modeling asymmetric relationships, we define a directed graph to be a pair $(V, E)$, where again $V$ is a finite set of vertices or nodes, but $E$ is a set of ordered pairs of distinct elements of $V$. Again, we refer to the elements of $E$ as edges. In order to differentiate the edges of an undirected graph from the edges of a directed graph, we sometimes refer to the former as undirected edges and to the latter as directed edges. We can represent directed graphs in a manner similar to our depiction of undirected graphs, using arrows to
indicate the directions of the edges. Conventionally, we draw the edge \((u, v)\) as an arrow from \(u\) to \(v\) (see Figure 9.2). For a directed edge \((u, v)\) we say that \(v\) is adjacent to \(u\), but not vice versa (unless \((v, u)\) is also an edge in the graph).

We usually want to associate some additional information with the vertices and/or the edges. For example, if the graph is used to represent distances between points on a map, we would want to associate a distance with each edge. In addition, we might want to associate the name of a city with each vertex. In order to simplify our presentation, we will focus our attention on the edges of a graph and any information associated with them. Specifically, as we did for disjoint sets in the previous chapter, we will adopt the convention that the vertices of a graph will be designated by natural
numbers 0, . . . , n − 1. If additional information needs to be associated with vertices, it can be stored in an array indexed by the numbers designating the vertices. While some applications might require more flexibility, this scheme is sufficient for our purposes.

Although in practice it may be beneficial to define separate ADTs for directed and undirected graphs, respectively, it will simplify our presentation if we specify a single Graph ADT, as shown in Figure 9.3. This actually specifies a directed graph, but we can use it to represent an undirected graph if we make sure that whenever (i, j) is an edge, then (j, i) is an edge with the same associated information. We can therefore use the same ADT for both types of graph, though the specification itself will never guarantee that the graph is undirected.

This specification uses the data type Edge, which is implemented by three readable representation variables, source, dest, and data. We assume that it contains a constructor Edge(i, j, x), which sets source to i, dest to j, and data to x. It contains no other operations other than the tree accessor operations, so it is an immutable structure. Although we place no restrictions on the values stored in its representation variables, the specification of Graph.AllFrom ensures that any Edge in the ConsList returned by this operation will have natural numbers for source and dest and a non-nil value for data.

In the next two sections, we will consider two applications of the Graph ADT. Each of these applications will result in a graph algorithm. We will then examine two implementations of Graph and analyze the running times of their operations. We will also analyze the space usage of each implementation. Using these analyses, we will analyze the running times of the two algorithms with each of these implementations.

9.1 Universal Sink Detection

Our first example is somewhat contrived, but it serves as a useful introduction to graph algorithms. To begin, we define a sink in a directed graph $G = (V, E)$ to be a vertex $v$ with no outgoing edges. A universal sink is a sink $v$ such that for every vertex $u \neq v$, $(u, v) \in E$. In this section, we will examine the problem of finding a universal sink in a directed graph, if one exists.

We first observe that a directed graph can have at most one universal sink. Let us therefore consider the related problem of returning a universal sink in a nonempty graph if one exists, or returning an arbitrary vertex
**Graph ADT**

**Precondition:** \( n \) is a \( \text{Nat} \).

**Postcondition:** Constructs a \texttt{Graph} with vertices \( 0, \ldots, n-1 \) and no edges.

\texttt{Graph}(\( n \))

**Precondition:** true.

**Postcondition:** Returns the number of vertices in the graph.

\texttt{Graph.Size}()

**Precondition:** \( i \) and \( j \) are \texttt{Nat}s less than the number of vertices, \( i \neq j \), and \( x \) refers to a non-nil item.

**Postcondition:** Associates \( x \) with the edge \((i, j)\), adding this edge if necessary.

\texttt{Graph.Put}(i, j, x)

**Precondition:** \( i \) and \( j \) are \texttt{Nat}s less than the number of vertices.

**Postcondition:** Returns the data item associated with edge \((i, j)\), or \texttt{nil} if \((i, j)\) is not in the graph.

\texttt{Graph.Get}(i, j)

**Precondition:** \( i \) is a \texttt{Nat} less than the number of vertices.

**Postcondition:** Returns a \texttt{ConsList} of \texttt{Edges} representing the edges proceeding from vertex \( i \), where an edge \((i, j)\) with data \( x \) is represented by an \texttt{Edge} with \texttt{source} = \( i \), \texttt{dest} = \( j \), and \texttt{data} = \( x \).

\texttt{Graph.AllFrom}(i)
otherwise. We will then reduce the universal sink detection problem to this variant. Suppose we consider any two distinct vertices, \( u \) and \( v \) (if there is only one vertex, clearly it is a universal sink). If \((u, v) \in E\), then \( u \) cannot be a sink. Otherwise, \( v \) cannot be a universal sink. Let \( G' \) be the graph obtained by removing from \( G \) one vertex \( x \) that is not a universal sink, along with all edges incident on \( x \). If \( G \) has a universal sink \( w \), then \( w \) must also be a universal sink in \( G' \). We have therefore transformed this problem to a smaller instance. Because this reduction is a transformation, we can implement it using a loop.

In order to implement this algorithm using the Graph ADT, we need to generalize the problem to a subgraph of \( G \) comprised of the vertices \( i, \ldots, j \) and all edges between them. If \( j > i \), we can then eliminate either \( i \) or \( j \) from the range of vertices, depending on whether \((i, j)\) is an edge. Note that by generalizing the problem in this way, we do not need to modify the graph when we eliminate vertices — we simply keep track of \( i \) and \( j \), the endpoints of the range of vertices we are considering. If there is an edge \((i, j)\), we eliminate vertex \( i \) by incrementing \( i \); otherwise, we eliminate vertex \( j \) by decrementing \( j \). When all vertices but one have been eliminated (i.e., when \( i = j \)), the remaining vertex must be the universal sink if there is one.

We can therefore solve the original universal sink detection problem for a nonempty graph by first finding a candidate vertex \( i \) as described above. We know that if there is a universal sink, it must be \( i \). We then check whether \( i \) is a universal sink by verifying that for every \( j \neq i \), \((j, i)\) is an edge but \((i, j)\) is not. The resulting algorithm is shown in Figure 9.4.

### 9.2 Topological Sort

A cycle in a directed graph \( G = (V, E) \) is a finite sequence of vertices \( v_0, \ldots, v_k \) such that \((v_k, v_0) \in E\), and for \( 0 \leq i < k \), \((v_i, v_{i+1}) \in E\). A directed graph with no cycles is said to be acyclic. These two terms apply analogously to undirected graphs as well, except that in this case all the edges in the sequence must be distinct. In either type of graph, a cycle in which all the vertices in the sequence are distinct is said to be simple.

Directed acyclic graphs are often used to model precedence relationships between objects or activities. Suppose, for example, that we have four jobs, \( A, B, C, \) and \( D \). We must schedule these jobs sequentially such that \( A \) precedes \( C \), \( B \) precedes \( A \), and \( B \) precedes \( D \). These precedence relationships can be modeled by the directed acyclic graph shown in Figure 9.5. We need to find an ordering of the vertices such that for every edge \((u, v)\), \( u \) precedes
Figure 9.4 An algorithm to find a universal sink in a directed graph

**Precondition:** $G$ refers to a Graph.
**Postcondition:** Returns the universal sink in $G$, or $-1$ if $G$ has no universal sink.

UniversalSink($G$)

if $G$.Size() = 0
    return $-1$
else
    $i \leftarrow 0$; $j \leftarrow G$.Size() - 1
    // Invariant: If $G$ has a universal sink $k$, then $i \leq k \leq j$.
    while $i < j$
        if $G$.Get($i$, $j$) = nil
            $j \leftarrow j - 1$
        else
            $i \leftarrow i + 1$
            // Invariant: For $0 \leq k < j$, if $k \neq i$, then $(k, i)$ is an edge,
            // but $(i, k)$ is not.
    for $j \leftarrow 0$ to $G$.Size() - 1
        if $j \neq i$ and ($G$.Get($i$, $j$) $\neq$ nil or $G$.Get($j$, $i$) $=$ nil)
            return $-1$
    return $i$

$v$ in the ordering. Such an ordering is called a *topological sort* of the graph. Examples of topological sorts of the graph in Figure 9.5 are $\langle B, A, C, D \rangle$ and $\langle B, D, A, C \rangle$. In this section, we will present an algorithm for finding a topological sort of a given directed acyclic graph. First, we will show that every directed acyclic graph has a topological sort.

**Lemma 9.1** Every nonempty directed acyclic graph has at least one vertex with no incoming edges.

**Proof:** By contradiction. Suppose every vertex in some nonempty directed acyclic graph $G$ has incoming edges. Then starting from any vertex, we may always traverse an incoming edge backwards to its source. Because $G$ has finitely many vertices, if we trace a path in this fashion, we must eventually repeat a vertex. We will have then found a cycle — a contradiction. □
Theorem 9.2  Every directed acyclic graph $G = (V, E)$ has a topological sort.

Proof: By induction on the size of $V$.

Base: $V = \emptyset$. Then the empty ordering is a topological sort.

Induction Hypothesis: Suppose that for some $n > 0$, every directed acyclic graph with fewer than $n$ vertices has a topological sort.

Induction Step: Let $G = (V, E)$ be a directed acyclic graph with $n$ vertices. Because $G$ is nonempty, it must have at least one vertex $v_0$ with no incoming edges. Let $G'$ be the graph obtained from $G$ by removing $v_0$ and all of its outgoing edges. By the induction hypothesis, $G'$ has a topological sort $v_1, \ldots, v_{n-1}$. Because $v_0$ has no incoming edges in $G$, $v_0, \ldots, v_{n-1}$ must therefore be a topological sort for $G$. $\square$

The proof of Theorem 9.2 is constructive; i.e., it gives an algorithm for finding a topological sort. First, we find a vertex $v_0$ with no incoming edges. $v_0$ will come first in the topological sort. The remainder of the topological sort is obtained by removing $v_0$ and finding a topological sort of the resulting graph. We have therefore transformed the problem to a smaller instance.

The above sketch is missing a few details. For example, we need to know how to find a vertex with no incoming edges. Also, our Graph ADT provides no mechanism for removing vertices. In order to overcome these problems, we will maintain an array $\text{incount}[0..n-1]$ so that $\text{incount}[i]$ gives the number of edges to $i$ from vertices not yet in the topological sort. When we add a vertex $i$ to the topological sort, we do not need to remove it from
the graph; instead, we can simply decrement $incount[j]$ for all $j$ adjacent to $i$. To initialize $incount$, we can simply examine each edge $(i, j)$ and increment $incount[j]$.

In order to speed up finding the next vertex in the topological sort, let us keep track of all vertices $i$ for which $incount[i] = 0$. We can use a Stack for this purpose. After we initialize $incount$, we can traverse it once and push each $i$ such that $incount[i] = 0$ onto the stack. Thereafter, when we decrement an entry $incount[i]$, we need to see if it reaches 0, and if so, push it onto the stack. The algorithm is shown in Figure 9.6.

### 9.3 Adjacency Matrix Implementation

Our first implementation of Graph will have a single representation variable:

- $edges[0..n-1, 0..n-1]$.

Our structural invariant will be that $edges[i, i] = \text{nil}$ for $0 \leq i < n$. We interpret $edges[i, j]$ as giving the information associated with edge $(i, j)$, provided this value is non-nil. We interpret a nil value for $edges[i, j]$ as indicating the absence of an edge $(i, j)$. The full implementation is shown in Figure 9.7.

Each of the operations Size, Put, and Get runs in $\Theta(1)$ time. The constructor is easily seen to run in $\Theta(n^2)$ time. The AllFrom operation clearly runs in $\Theta(n)$ time, where $n$ is the number of vertices in the graph. The space usage is clearly in $\Theta(n^2)$.

We can now analyze the running times of the algorithms given in the previous two sections. We will first consider UniversalSink from Figure 9.4. Let $n$ be the number of vertices in $G$. The while loop begins with $j - i = n - 1$ and decreases $j - i$ each iteration. Because it terminates when $j - i = 0$ it iterates $n - 1$ times. Because MatrixGraph.Get runs in $\Theta(1)$ time, the entire loop runs in $\Theta(n)$ time. The for loop iterates at most $n$ times, and each iteration runs in $\Theta(1)$ time. The entire algorithm therefore operates in $\Theta(n)$ time.

Let us now consider TopSort from Figure 9.6. We will need to break the algorithm into the four for loops.

In the body of the second for loop, MatrixGraph.AllFrom runs in $\Theta(n)$ time. The body of the while loop runs in $\Theta(1)$ time. Each iteration decreases the number of elements in $L$ by 1 until $L$ is empty. Because there can be at most $n - 1$ edges from any vertex, the while loop iterates at most
Figure 9.6 Topological sort algorithm

**Precondition:** $G$ is a directed acyclic graph.

**Postcondition:** Returns an array listing the vertices of $G$ in topological order.

\[
\text{TopSort}(G) \\
  n \leftarrow G.\text{Size}(); \ s \leftarrow \text{new ARRAY}[0..n - 1] \\
  \text{incount} \leftarrow \text{new ARRAY}[0..n - 1]; \ \text{avail} \leftarrow \text{new STACK}() \\
  \text{for } i \leftarrow 0 \ \text{to } n - 1 \\
    \text{incount}[i] \leftarrow 0 \\
  \]  
// **Invariant:** For $0 \leq j < n$, incount[j] gives the number of edges to $j$
// from vertices $0, \ldots, i - 1$.

\[
\text{for } i \leftarrow 0 \ \text{to } n - 1 \\
  L \leftarrow G.\text{AllFrom}(i) \\
  \text{while not L.ISEMPTY()} \\
    k \leftarrow L.\text{HEAD}().\text{Dest}(); \ \text{incount}[k] \leftarrow \text{incount}[k] + 1; \ L \leftarrow L.\text{TAIL}() \\
  \text{for } i \leftarrow 0 \ \text{to } n - 1 \\
    \text{if } \text{incount}[i] = 0 \\
      \text{avail}.\text{Push}(i) \\
  \]  
// **Invariant:** For $0 \leq j < n$, incount[j] is the number of edges to $j$ from
// vertices not in $s[0..i - 1]$, and if incount[j] = 0, then $j$ is either in
// $s[0..i - 1]$ or avail, but not both.

\[
\text{for } i \leftarrow 0 \ \text{to } n - 1 \\
  s[i] \leftarrow \text{avail}.\text{Pop}(); \ L \leftarrow G.\text{AllFrom}(s[i]) \\
  \text{while not L.ISEMPTY()} \\
    k \leftarrow L.\text{HEAD}().\text{Dest}(); \ \text{incount}[k] \leftarrow \text{incount}[k] - 1; \ L \leftarrow L.\text{TAIL}() \\
    \text{if } \text{incount}[k] = 0 \\
      \text{avail}.\text{Push}(k) \\
  \]

return $s$
Structural Invariant: $\text{edges}[i, i] = \text{nil}$ for $0 \leq i < n$.

**MatrixGraph**($n$)

```plaintext
edges ← new Array[0..n−1, 0..n−1] 
for $i ← 0$ to $n − 1$
    for $j ← 0$ to $n − 1$
        edges[$i, j$] ← nil 
```

**MatrixGraph.Size()**

```plaintext
return SizeOf(edges[0]) // Number of columns 
```

**MatrixGraph.Put**($i, j, x$)

```plaintext
if $i = j$
    error
else
    edges[$i, j$] ← $x$ 
```

**MatrixGraph.Get**($i, j$)

```plaintext
return edges[$i, j$] 
```

**MatrixGraph.AllFrom**($i$)

```plaintext
L ← new ConsList()
for $j ← 0$ to $n − 1$
    if $\text{edges}[i, j] \neq \text{nil}$
        $L ← \text{new ConsList(new Edge}(i, j, \text{edges}[i, j]), L)$
return L 
```
n − 1 times. Its running time is therefore in \(O(n)\). The body of the second for loop therefore runs in \(\Theta(n)\) time. Because it iterates \(n\) times, its running time is in \(\Theta(n^2)\).

The first and third for loops clearly run in \(\Theta(n)\) time. Furthermore, the analysis of the fourth for loop is similar to that of the second. Therefore, the entire algorithm runs in \(\Theta(n^2)\) time.

Note that the second and fourth for loops in TopSort each contain a nested while loop. Each iteration of this while loop processes one of the edges. Furthermore, each edge is processed at most once by each while loop. The total number of iterations of each of the while loops is therefore the number of edges in the graph. While this number can be as large as \(n(n − 1) \in \Theta(n^2)\), it can also be much smaller.

The number of edges does not affect the asymptotic running time, however, because MatrixGraph.AllFrom runs in \(\Theta(n)\) time, regardless of how many edges it retrieves. If we can make this operation more efficient, we might be able to improve the running time for TopSort on graphs with few edges. In the next section, we will examine an alternative implementation that accomplishes this.

### 9.4 Adjacency List Implementation

In this section, we consider an implementation designed to improve the efficiency of the AllFrom operation. The two-dimensional array used in the adjacency matrix implementation can be thought of as an array of arrays each containing the adjacency information for a single vertex. In the adjacency list implementation, we still use an array indexed by vertices to store the adjacency information for each vertex; however, we maintain this adjacency information in a ConsList instead of an array. In such a representation, the ConsList for vertex \(i\) is exactly the ConsList that needs to be returned by AllFrom\((i)\). Furthermore, because a ConsList is immutable, we can return it without violating security.

We again use a single representation variable:

- \(\text{elements}[0..n-1]\): an array of ConsLists.

Our structural invariant will be that for \(0 \leq i < n\), \(\text{elements}[i]\) refers to a ConsList containing Edges representing at most one \((i, j)\) for each \(j\) such that \(0 \leq j < n\) and \(j \neq i\), where each Edge has a non-nil data item (see the specification for Graph.AllFrom in Figure 9.3 for an explanation of this representation). We interpret the Edges in this structure as representing
the edges in the graph, along with the information associated with each edge.

A partial implementation of \texttt{ListGraph} is shown in Figure 9.8. In addition, the \texttt{SIZE} operation returns the size of \texttt{elements}, and \texttt{ALLFROM}(i) returns \texttt{elements}[i].

It is easily seen that the \texttt{SIZE} and \texttt{ALLFROM} operations run in $\Theta(1)$ time, and that the constructor runs in $\Theta(n)$ time. Each iteration of the \texttt{while} loop in the \texttt{GET} operation reduces the size of $L$ by 1. The length of $L$ is initially the number of vertices adjacent to $i$. Because each iteration runs in $\Theta(1)$ time, the entire operation runs in $\Theta(m)$ time in the worst case, where $m$ is the number of vertices adjacent to $i$. The worst case for \texttt{GET} occurs when vertex $j$ is not adjacent to $i$. Similarly, it can be seen that the \texttt{PUT} operation runs in $\Theta(m)$ time. Note that $\Theta(m) \subseteq O(n)$. The space usage of \texttt{ListGraph} is easily seen to be in $\Theta(n + a)$, where $a$ is the number of edges in the graph.

Let us now revisit the analysis of the running time of \texttt{TopSort} (Figure 9.6), this time assuming that $G$ is a \texttt{ListGraph}. Consider the second \texttt{for} loop. Note that running time of the nested \texttt{while} loop does not depend on the implementation of $G$; hence, we can still conclude that it runs in $O(n)$ time. We can therefore conclude that the running time of the second \texttt{for} loop is in $O(n^2)$. However, because we have reduced the running time of \texttt{ALLFROM} from $\Theta(n)$ to $\Theta(1)$, it is no longer clear that the running time of this loop is in $\Omega(n^2)$. Indeed, if there are no edges in the graph, then the nested \texttt{while} loop will not iterate. In this case, the running time is in $\Theta(n)$.

We therefore need to analyze the running time of the nested \texttt{while} loop more carefully. Notice that over the course of the \texttt{for} loop, each edge is processed by the inner \texttt{while} loop exactly once. Therefore, the body of the inner loop is executed exactly $a$ times over the course of the entire outer loop, where $a$ is the number of edges in $G$. Because the remainder of the outer loop is executed exactly $n$ times, the running time of the outer loop is in $\Theta(n + a)$.

We now observe that the fourth loop can be analyzed in exactly the same way as the second loop; hence, the fourth loop also runs in $\Theta(n + a)$ time. In fact, because the structure of these two loops is quite common for graph algorithms, this method of calculating the running time is often needed for analyzing algorithms that operate on \texttt{ListGraphs}.

To complete the analysis of \texttt{TopSort}, we observe that the first and third loops do not depend on how $G$ is implemented; hence, they both run in $\Theta(n)$ time. The total running time of \texttt{TopSort} is therefore in $\Theta(n + a)$. For graphs in which $a \in o(n^2)$, this is an improvement over the $\Theta(n^2)$
Figure 9.8 ListGraph implementation (partial) of Graph

**Structural Invariant:** For $0 \leq i < n$, $\text{elements}[i]$ refers to a ConsList containing Edges representing at most one $(i, j)$ for each $j$ such that $0 \leq j < n$ and $j \neq i$, where each Edge has a non-nil data item.

```
ListGraph(n)
  elements ← new Array[0..n - 1]
  for i ← 0 to n - 1
    elements[i] ← new ConsList()

ListGraph.Put(i, j, x)
  if j ≥ SIZEOF(elements) or j < 0 or j = i or x = nil
    error
  else
    elements[i] ← ADDEdge(new Edge(i, j, x), elements[i])

ListGraph.Get(i, j)
  L ← elements[i]
  while not L.ISEmpty()
    if L.HEAD().DEST() = j
      return L.HEAD().DATA()
    else
      L ← L.TAIL()
  return nil
```

--- Internal Functions Follow ---

**Precondition:** $e$ refers to an Edge and $L$ refers to a ConsList of Edges each having the same source as $e$.

**Postcondition:** Returns a ConsList with the contents of $L$, plus the edge $e$. If the edge represented by $e$ is already in $L$, it is replaced by $e$.

```
ListGraph AddEdge(e, L)
  if L.ISEmpty()
    return new ConsList(e, L)
  else if L.HEAD().DEST() = e.DEST()
    return new ConsList(e, L.TAIL())
  else
    return new ConsList(L.HEAD(), ADDEdge(e, L.TAIL()))
```
Let us now consider the impact of the ListGraph implementation on the analysis of UniversalSink (Figure 9.4). Due to the increased running time of Get, the body of the while loop runs in $\Theta(m)$ time, where $m$ is the number of vertices adjacent to $i$. This number cannot be more than $n-1$, nor can it be more than $a$. Because this loop iterates $\Theta(n)$ times, we obtain an upper bound of $O(n \min(n, a))$. Likewise, it is easily seen that the for loop runs in $O(n \min(n, a))$ time.

To see that this bound is tight for the while loop, let us first consider the case in which $a \leq n(n-1) - \lfloor n/2 \rfloor$. Suppose that from vertex 0 there is an edge to each of the vertices $1, \ldots, \min(a, \lfloor (n-1)/2 \rfloor)$, but no edge to any other vertex. From vertices other than 0 we may have edges to any of the other vertices. Note that with these constraints, we can have up to $n(n-1) - \lfloor n/2 \rfloor$ edges. For such a graph, the first $\lfloor n/2 \rfloor$ iterations of the while loop will have $i = 0$, while $j$ ranges from $n-1$ down to $\lfloor (n-1)/2 \rfloor + 1$. For each of these iterations, Get($i, j$) runs in $\Theta(\min(a, n))$ time, because there are $\Theta(\min(a, n))$ vertices adjacent to 0, but $j$ is not adjacent to 0. Because the number of these iterations is in $\Theta(n)$, the total time is in $\Theta(n \min(n, a))$.

Now let us consider the case in which $a > n(n-1) - \lfloor n/2 \rfloor$. In this case, we make sure that from each of the vertices $0, \ldots, \lfloor n/2 \rfloor - 1$, there is an edge to every other vertex. Furthermore, we make sure that in each of the ConsLists of edges from these first $\lfloor n/2 \rfloor$ vertices, the edge to vertex $n-1$ occurs last. From the remaining vertices we may have any edges listed in any order. For such a graph, the first $\lfloor n/2 \rfloor$ iterations of the while loop will have $j = n-1$, while $i$ ranges from 0 to $\lfloor n/2 \rfloor - 1$. For each of these iterations, Get($i, j$) runs in $\Theta(n)$ time, because there are $\Theta(n)$ vertices adjacent to $i$, and $n-1$ is the last of these. Because the total number of iterations is in $\Theta(n)$, the total time is in $\Theta(n^2)$. Because $a \geq n$, this is the same as $\Theta(n \min(a, n))$.

Based on the analyses of the two algorithms, we can see that neither implementation is necessarily better than the other. If an algorithm relies more heavily on Get than on AllFrom, it is better to use MatrixGraph. If an algorithm relies more heavily on AllFrom, it is probably better to use ListGraph, particularly if there is a reasonable expectation that the graph will be sparse — i.e., that it will have relatively few edges. Note also that for sparse graphs, a ListGraph will use considerably less space.
9.5 Multigraphs

Let us briefly consider the building of a ListGraph. We must first construct a graph with no edges, then add edges one by one using the Put operation. The constructor runs in $\Theta(n)$ time. The Put operation runs in $\Theta(m)$ time, where $m$ is the number of vertices adjacent to the source of the edge. It is easily seen that the time required to build the graph is in $O(n + a \min(n, a))$, where $a$ is the number of edges. It is not hard to match this upper bound using graphs in which the number of vertices with outgoing edges is minimized for the given number of edges. An example of a sparse graph (specifically, with $a \leq n$) that gives this behavior is a graph whose edge set is

$$\{(0, j) \mid 1 \leq j < a\}.$$  

A dense graph giving this behavior is a complete graph — a graph in which every possible edge is present. Note that in terms of the number of vertices, the running time for building a complete graph is in $\Theta(n^3)$.

Building a ListGraph is expensive because the Put operation must check each edge to see if it is already in the graph. We could speed this activity considerably if we could avoid this check. However, the check is necessary not only to satisfy the operation’s specification, but also to maintain the structural invariant. If we can modify the specification of Put and weaken the structural invariant so that parallel edges (i.e., multiple edges from a vertex $i$ to a vertex $j$) are not prohibited, then we can build the graph more quickly.

We therefore extend the definitions of undirected and directed graphs to allow parallel edges from one vertex to another. We call such a structure a multigraph. We can then define the Multigraph ADT by modifying the following postconditions in the specification of Graph:

- **Put($i, j, x$)**: Adds the edge $(i, j)$ and associates $x$ with it.

- **Get($i, j$)**: Returns the data item associated with an edge $(i, j)$, or nil if $(i, j)$ is not in the graph.

We could also specify additional operations for retrieving all edges $(i, j)$ or modifying the data associated with an edge, but this specification is sufficient for our purposes.

We will represent a Multigraph using adjacency lists in the same way as in the ListGraph implementation. The structural invariant will be modified to allow parallel edges $(i, j)$ for the same $i$ and $j$. The implementation of the operations remains the same except for Put, whose implementation is
Figure 9.9 The Put operation in the ListMultigraph implementation of Multigraph

```
ListMultigraph.Put(i, j, x)
    if j ≥ SizeOf(elements) or j < 0 or x = nil
        error
    else
        elements[i] ← new ConsList(new Edge(i, j, x), elements[i])
```

shown in Figure 9.9. It is easily seen that a ListMultigraph can be built in $\Theta(n + a)$ time, where $n$ is the number of vertices and $a$ is the number of edges.

Because it is more efficient to build a ListMultigraph than to build a ListGraph, it may be advantageous to represent a graph using a ListMultigraph. If we are careful never to add parallel edges, we can maintain an invariant that the ListMultigraph represents a graph. This transfers the burden of maintaining a valid graph structure from the Put operation to the code that invokes the Put operation.

Although we can use a ListMultigraph to represent a graph, an interesting problem is how to construct a ListGraph from a given ListMultigraph. Specifically, suppose we wish to define a ListGraph constructor that takes a ListMultigraph as input and produces a ListGraph with the same vertices and edges, assuming the ListMultigraph has no parallel edges. We may be able to do this more efficiently than simply calling ListGraph.Put repeatedly.

One approach would be to convert the ListMultigraph to a MatrixGraph, then convert the MatrixGraph to a ListGraph. In fact, we do not really need to build a MatrixGraph — we could simply use a two-dimensional array as a temporary representation of the graph. As we examine each edge of the ListMultigraph, we can check to see if it has been added to the array, and if not, add it to the appropriate adjacency list. If we ever find parallel edges, we can immediately terminate with an error. Each edge can therefore be processed in $\Theta(1)$ time, for a total of $\Theta(a)$ time to process the edges. Unfortunately, $\Theta(n^2)$ time is required to initialize the array, and the space usage of $\Theta(n^2)$ is rather high, especially for sparse graphs. However, the resulting time of $\Theta(n^2)$ is still an improvement (in
most cases) over the $\Theta(a \min(n, a))$ worst-case time for repeatedly calling ListGraph.Put.

In the above solution, the most natural way of processing the edges of the ListMultigraph is to consider each vertex $i$ in turn, and for each $i$, to process all edges proceeding from $i$. As we are processing the edges from vertex $i$, we only need row $i$ of the array. We could therefore save a significant amount of space by replacing the two-dimensional array with a singly-dimensioned array $A[0..n - 1]$. Before we consider any vertex $i$, we initialize $A$. For each edge $(i, j)$, we check to see if it has been recorded in $A[j]$. If so, we have found a parallel edge; otherwise, we record this edge in $A[j]$. Thus, we have reduced our space usage to $\Theta(n)$. However, because $A$ must be initialized each time we consider a new vertex, the overall running time is still in $\Theta(n^2)$.

Note that if we ignore the time for initializing $A$, this last solution runs in $\Theta(n + a)$ time in the worst case. We can therefore use the virtual initialization technique of Section 7.1 to reduce the overall running time to $\Theta(n + a)$. However, the technique of virtual initialization was inspired by the need to avoid initializing large arrays, not to avoid initializing small arrays many times. If we are careful about the way we use the array, a single initialization should be significant. Thus, if we can find a way to avoid repeated initializations of $A$, we will be able to achieve $\Theta(n + a)$ running time without using virtual initialization.

Consider what happens if we simply omit every initialization of $A$ except the first. If, when processing edge $(i, j)$, we find edge $(i', j)$, for some $i' < i$, recorded in $A[j]$, then we know that no other edge $(i, j)$ has yet been processed. We can then simply record $(i, j)$ in $A[j]$, as if no edge had been recorded there. If, on the other hand, we find that $(i, j)$ has already been recorded in $A[j]$, we know that we have found a parallel edge.

As a final simplification to this algorithm, we note that there is really no reason to store EDGES in the array. Specifically, the only information we need to record in $A[j]$ is the most recent (i.e., the largest) vertex $i$ for which an edge $(i, j)$ has been found. Thus, $A$ can be an array of integers. We can initialize $A$ to contain only negative values, such as $-1$, to indicate that no such edge has yet been found. The resulting ListGraph constructor is shown in Figure 9.10. It is easily seen to run in $\Theta(n + a)$ time and use $\Theta(n)$ temporary space in the worst case, where $n$ and $a$ are the number of vertices and unique edges, respectively, in the given ListMultigraph.
Figure 9.10 Constructor for building a ListGraph from a ListMultigraph

**Precondition:** $G$ refers to a ListMultigraph with no parallel edges.

**Postcondition:** Constructs a ListGraph with the same vertices and edges as $G$.

```plaintext
ListGraph(G)
size ← G.SIZE(); elements ← new ARRAY[0..size - 1]
A ← new ARRAY[0..size - 1]
// Invariant: For 0 ≤ k < i, elements[k] = G.ALLFROM(k) and
for i ← 0 to size - 1
    elements[i] ← G.ALLFROM(i); A[i] ← -1
// Invariant: For 0 ≤ j < size, A[j] < i. For 0 ≤ k < i and 0 ≤ j < size,
// elements[k] has at most one edge to j.
for i ← 0 to size - 1
    L ← elements[i]
    // Invariant: For 0 ≤ j < size, A[j] ≤ i. If A[j] = i, then elements[i]
    // contains one more edge to j than does L; otherwise, they contain
    // the same number of edges to j.
while not L.ISEMPTY()
    j ← L.HEAD().DEST()
    if A[j] < i
        A[j] ← i; L ← L.TAIL()
    else
        error
```

9.6 Summary

Graphs are useful for representing relationships between data items. Various algorithms can then be designed for manipulating graphs. As a result, we can often use the same algorithm in a variety of different applications.

Graphs may be either directed or undirected, but we can treat undirected graphs as directed graphs in which for every edge \((u, v)\), there is a reverse edge \((v, u)\). We then have two implementations of graphs. The adjacency matrix implementation has \text{GET} and \text{PUT} operations that run in \(\Theta(1)\) time, but its \text{ALLFROM} operation runs in \(\Theta(n)\) time, where \(n\) is the number of vertices in the graph. Its space usage is in \(\Theta(n^2)\). On the other hand, the adjacency list implementation has an \text{ALLFROM} operation that runs in \(\Theta(1)\) time, but its \text{GET} and \text{PUT} operations run in \(\Theta(m)\) time in the worst case, where \(m\) is the number of vertices adjacent to the given source vertex. Its space usage is in \(\Theta(n + a)\) where \(n\) is the number of vertices and \(a\) is the number of edges.

In order to improve the running time of the \text{PUT} operation — and hence of building a graph — when using an adjacency list, we can relax our definition to allow parallel edges. The resulting structure is known as a multigraph. We can always use a multigraph whenever a graph is required, though it might be useful to maintain an invariant that no parallel edges exist. Furthermore, we can construct a \text{ListGraph} from a \text{ListMultigraph} with no parallel edges in \(\Theta(n + a)\) time and \(\Theta(n)\) space, where \(n\) is the number of vertices and \(a\) is the number of edges. Figure 9.11 shows a summary of the running times of these operations for each of the implementations of \text{Graph}, as well as for \text{ListMultigraph}.

9.7 Exercises

Exercise 9.1 Prove that \text{UniversalSink}, shown in Figure 9.4, meets its specification.

Exercise 9.2 Prove that \text{TopSort}, shown in Figure 9.6, meets its specification.

Exercise 9.3 Give an algorithm that takes as input a directed graph \(G = (V, E)\) and returns a directed graph \(G' = (V, E')\), where

\[
E' = \{(u, v) \mid (v, u) \in E\}.
\]
Figure 9.11 Comparison of running times for two implementations of Graph, along with ListMultigraph

<table>
<thead>
<tr>
<th></th>
<th>constructor</th>
<th>Get</th>
<th>Put</th>
<th>AllFrom</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixGraph</td>
<td>Θ(n^2)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>ListGraph</td>
<td>Θ(n)</td>
<td>Θ(m)</td>
<td>Θ(m)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>ListMultigraph</td>
<td>Θ(n)</td>
<td>Θ(m)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>

Notes:

- *n* is the number of vertices in the graph, and *m* is the number of vertices adjacent to the given source vertex.
- The constructor referenced above is the one that constructs a graph with no edges — the ListGraph constructor that takes a ListMultigraph as input runs in Θ(n+a) worst-case time, where *n* is the number of vertices and *a* is the number of edges in the graph, assuming there are no parallel edges.
- The Size operation runs in Θ(1) worst-case time for all implementations.
- All running times are worst-case.

Thus, $G'$ contains the same edges as does $G$, except that they are reversed in $G'$. Express the running time of your algorithm as simply as possible using Θ-notation in terms of the number of vertices *n* and the number of edges *a*, assuming the graphs are implemented using

a. MatrixGraph
b. ListGraph
c. ListMultigraph.

Exercise 9.4 Give an algorithm to compute the number of edges in a given graph. Express the running time of your algorithm as simply as possible using Θ-notation in terms of the number of vertices *n* and the number of edges *a*, assuming the graph is implemented using

a. MatrixGraph
CHAPTER 9. GRAPHS

b. ListGraph

Exercise 9.5 A directed graph is said to be transitivity closed if whenever
(u, v) and (v, w) are edges, then (u, w) is also an edge. Give an \( O(n^3) \)
algorithm to determine whether a given directed graph is transitivity closed.
You may assume the graph is implemented as a MatrixGraph.

Exercise 9.6 An undirected graph is said to be connected if for every pair
of vertices u and v, there is a path of edges leading from u to v. A tree
is a connected undirected acyclic graph. Prove each of the following for an
undirected graph \( G \) with \( n \) vertices:

a. If \( G \) is a tree, then \( G \) has exactly \( n - 1 \) edges.

b. If \( G \) is connected and has exactly \( n - 1 \) edges, then \( G \) is a tree.

c. If \( G \) is acyclic and has exactly \( n - 1 \) edges, then \( G \) is a tree.

Exercise 9.7 Prove that MatrixGraph, shown in Figure 9.7, meets its
specification.

Exercise 9.8 Prove that the ListGraph constructor shown in Figure 9.10
meets its specification.

Exercise 9.9 Give an algorithm that takes as input a graph \( G \) and returns
true iff \( G \) is undirected; i.e., if for every edge \((u, v)\), \((v, u)\) is also an edge.
Give the best upper bound you can on the running time, expressed as simply
as possible using big-\( O \) notation in terms of the number of vertices \( n \) and
the number of edges \( a \), assuming the graph is implemented using Matrix-
Graph.

* Exercise 9.10 An Euler path in a connected undirected graph \( G \) is a
path that contains every edge in \( G \) exactly once. Give an efficient algorithm
to find an Euler path in a connected undirected graph, provided one exists.
Your algorithm should return some representation of the Euler path, or nil if
no Euler path exists. Your algorithm should run in \( O(a) \) time, where \( a \) is the
number of edges in \( G \), assuming that \( G \) is implemented as a ListGraph.
9.8 Chapter Notes

The study of graph theory began in 1736 with Leonhard Euler’s famous study of the Königsberg Bridge Problem [36], which is simply the problem of finding an Euler path in a connected undirected graph (see Exercise 9.10). Good references on graph theory and graph algorithms include Even [37] and Tarjan [104]. In the early days of electronic computing, graphs were typically implemented using adjacency matrices. Hopcroft and Tarjan [62] first proposed using adjacency lists for sparse graphs. The topological sort algorithm of Section 9.2 is due to Knuth [76].