Graph Algorithms

Timothy Vismor

June 11, 2011

Abstract

The network structure of many physical systems is represented by mathematical entities known as graphs. This document examines several of the computational algorithms of graph theory most relevant to network analysis.

Copyright © 1990 - 2011 Timothy Vismor

Contents

1 Graph Nomenclature **5**
1.1 Subgraph 5
1.2 Directed Graph 5
1.3 Undirected Graph 6
1.4 Paths and Connected Graphs 7
1.5 Cyclic Graph 7
1.6 Acyclic Graph 7

2 Modeling Graphs **9**
2.1 Representing Graphs as Lists 10
2.2 Representing Graphs as Adjaceny Matrices 10
2.3 Representing Sparse Graphs as Adjaceny Lists 11

3 Abstract Data Types for Graphs **13**
3.1 Adjacency List 13
3.2 Reduced Graph 14
3.3 List 14
3.4 Mapping 15

4 Creating Adjacency Lists **16**

5 Depth First Search **17**
5.1 Recursive Depth First Search 18
5.2 Non-recursive Depth First Search 19
5.3 Depth First Spanning Trees 20
5.4 Depth First Traversal 20

6 Graph Structure Analysis **21**
6.1 Connected Components of a Graph 22
6.2 Isolated Vertex Detection 22
6.3 Cycle Detection 22
6.3.1 Detecting Cycles in Undirected Graphs 23
6.3.2 Detecting Cycles in Directed Graphs 24

7 Determining the Degree of Each Vertex **25**

8 Vertex Elimination **26**
8.1 Eliminating a Single Vertex 26
8.2 Eliminating Many Vertices 27
8.3 Initializing Minimum Degree Vertex Tracking 28
8.4 Maintaining the Reduced Graph 29
8.5 Querying the Reduced Graph State 30

List of Figures

1 Example of a Directed Graph 6
2 Example of an Undirected Graph 7
3 Example of Cycles in a Digraph 8
4 Example of an Acyclic Digraph 8
5 Example of a Directed Tree 9
6 Example of a Rooted Free Tree 9
7 Adjacency List of Directed Graph in Figure 1 12
8 Adjacency List of Undirected Graph in Figure 2 12

List of Tables

1 Vertex Degree Summary for Figure 1 6

List of Algorithms

1 Create Adjacency List of Undirected Graph 16
2 Map Vertices To Labeling Order 16
3 Create Ordered Adjacency List 17
4 Depth First Search 18
5 Recursive Depth First Search 18
6 Depth First Search With Recursion Removed 19
7 Non-recursive DFS Vertex Visitation 19
8 Depth First Traversal 21
9 Extract the Connected Components of a Graph 22
10 Isolated Vertex Detection 23
11 Cycle Detection in Undirected Graphs 23
12 Cycle Detection in Directed Graphs 24
13 Determine Vertex Degree Using Adjacency List 25
14 Determine Vertex Degree of Directed Graph Using Edges . . . 25
15 Determine Vertex Degree of Undirected Graph Using Edges . 26

16 Eliminate a Vertex from a Graph 27
17 Initialize Minimum Degree Vertex Tracking 28
18 Increase the Degree of a Vertex 29
19 Decrease the Degree of a Vertex 29
20 Remove a Vertex from the Reduced Graph 30
21 Determine if a Vertex is in the Reduced Graph 30

1 Graph Nomenclature

Data structures and algorithms derived from graph theory form the basis for many network analysis techniques. A brief review of the most relevant aspects of computational graph theory follows. An excellent introduction to the subject is found in Aho, Hopcroft, and Ullman(1983) [1]. Horowitz and Sahni (1978) [2] cover similar material in a more disjointed fashion. Even (1980) [3] provides a more theoretical examination of the subject.

The graph $G = (V, E)$ consists of a finite set of vertices $V(G) = \{v_1, v_2, \cdots, v_n\}$ and a finite set of edges $E(G) = \{e_1, e_2, \cdots, e_n\}$.

Each edge corresponds to a pair of vertices. If edge e corresponds to the vertex pair (v, w), then e is incident upon vertices v and w. The number of vertices in $V(G)$ is indicated by $|V|$. The number of edges in $E(G)$ is indicated by $|E|$.

A labeling (or ordering) of the graph G is a mapping of the set $\{1, 2, \cdots, |V|\}$ onto $V(G)$.

The usual convention for drawing a graph, is to represent each vertex as a dot and each edge as a line connecting two dots. If the graph is directed, an arrow is superimposed on each edge to show its orientation.

1.1 Subgraph

A graph $G' = (V', E')$ is a subgraph of G if the following conditions apply.

- $V'(G') \subset V(G)$.
- $E'(G')$ consists of edges (v, w) in $E(G)$ where both v and w are in $V'(G')$.

If $E'(G')$ consists of all the edges in $E(G)$ for which the second condition holds, then G' is an induced subgraph of G. An induced subgraph of G that is not a proper subset of any other connected subgraph of G is called a connected component of G.

1.2 Directed Graph

If the edges of G are ordered pairs, then G is a directed graph. In a directed graph, $(v, w) \neq (w, v)$.

A directed graph is often referred to as a digraph. If edge e of a digraph is represented by (v, w), then e is an edge from v to w. Vertex w is adjacent to vertex v. Vertex v is not adjacent to vertex w unless the edge (w, v) also

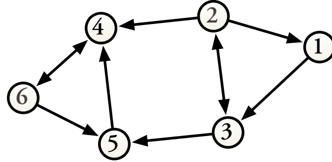

Figure 1: Example of a Directed Graph

exists in $\boldsymbol{G}$. The number of vertices adjacent to vertex v is the degree of v. In-degree indicates the number of edges incident upon v. Out-degree indicates the number of edges emanating from v.

Figure 1 depicts a directed graph with six vertices and ten edges. In the figure, the arrowheads indicate the direction of the edge.

Table 1 summarizes the degree of each vertex in Figure 1.

Table 1: Vertex Degree Summary for Figure 1

Vertex	Degree	In-Degree	Out-Degree
1	2	1	1
2	4	1	3
3	4	2	2
4	4	3	1
5	3	2	1
6	3	1	2

1.3 Undirected Graph

If the edges of $\boldsymbol{G}$ are unordered pairs, $\boldsymbol{G}$ is a undirected graph. In an undirected graph, $(v, w) = (w, v)$. If edge e of an undirected graph is represented by (v, w), then v is adjacent to w and w is adjacent to v. The terminology "undirected graph" is somewhat cumbersome. In this document, undirected graphs are often referred to as just "graphs".

Figure 2 depicts an undirected graph with six vertices and eight edges. The fact that the graph is undirected is indicated by the absence of directionality arrows.

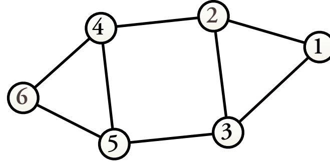

Figure 2: Example of an Undirected Graph

1.4 Paths and Connected Graphs

A path is a sequence of edges, e.g. $(v_1, v_2), (v_2, v_3), \cdots, (v_{n-1}, v_n)$. This path connects vertices v_1 and v_n. The path may also be represented by the vertex sequence $v_1, v_2, v_3, \cdots, v_{n-1}, v_n$. The path begins at vertex v_1 passes through vertices $v_2, v_3, \cdots, v_{n-1}$ and ends at vertex v_n.

The number of edges that comprise a path is its length. A path is simple if all of the vertices on a path are distinct (with the possible exception of the first and last vertices).

If some path connects each pair of vertices in $\boldsymbol{G}$, then $\boldsymbol{G}$ is a connected graph.

Connected digraphs are classified as either weakly connected or strongly connected. A strongly connected digraph has a directed path that connects all the vertices in the graph. All the vertices of a strongly connected digraph must have an in-degree of at least one.

A weakly connected digraph has has an undirected path that connects all the vertices in the graph. All the vertices of a weakly connected digraph must have either an in-degree or out-degree of at least one.

1.5 Cyclic Graph

A cycle is a simple path that begins and ends at the same vertex and has a length of at least one. We refer to any graph that contains a cycle as a cyclic graph.

Figure 3 illustrates the cycles that are present in the directed graph of Figure 1.

1.6 Acyclic Graph

A directed graph with no cycles is called directed acyclic graph or a DAG for short. An acyclic digraph has at least one vertex with an out-degree of zero.

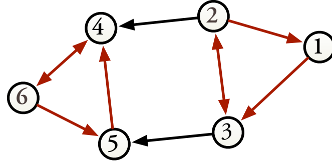

Figure 3: Example of Cycles in a Digraph

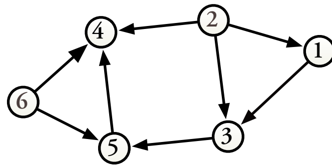

Figure 4: Example of an Acyclic Digraph

Figure 4 shows a variant of the directed graph of Figure 1 that contains no cycles. You will observe that vertex 4 has an out-degree of zero.

A directed tree is a connected DAG with the following properties:

- There is one vertex, called the root, which no edges enter.
- All vertices except the root have one entering edge.
- There is a unique path from each vertex to the root.

A DAG consisting of one or more trees is called a forest. If the graph $F = (V, E)$ is a forest and the edge (v, w) is in $E(F)$, vertex v is the parent of w and vertex w is the child of v. If there is a path from v to w, then vertex v is an ancestor of w and vertex w is a descendent of v. A vertex with no proper descendants is a leaf. A vertex v and its descendants form a subtree of F. The vertex v is the root of this subtree. The depth of vertex v is the length of the path from the root to v. The height of vertex v is the length of the longest path from v to a leaf. The height of a tree is the height of its root. The level of vertex v is its depth subtracted from the height of the tree.

Figure 5 depicts a directed tree. Its root is vertex 1. Its leaves are the set of vertices $L(G) = \{3, 4, 6, 8, 9\}$.

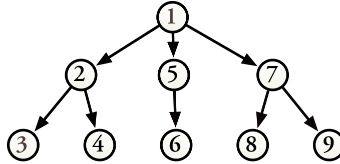

Figure 5: Example of a Directed Tree

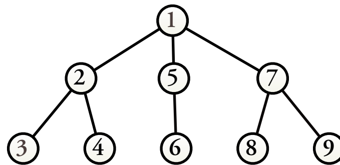

Figure 6: Example of a Rooted Free Tree

An undirected, connected, acyclic graph is called a free tree or an undirected tree. A rooted free tree is a free tree in which one vertex has been designated as the root. A directed tree is converted into a rooted free tree by discarding the orientation of the edges. A rooted free tree is converted into a directed tree by orienting each edge away from the root. The terminology which applies to directed trees also applies to rooted free trees.

Figure 6 depicts the directed tree of Figure 5 converted into a rooted free tree.

2 Modeling Graphs

A number of different data structures provide useful representations of a graph G. Different representations of G often lend themselves to specific applications. Indeed, the efficiency of graph algorithms often relies on the manner in which a graph is represented. For this reason, it may prove necessary to transform a graph from one representation to another to implement a complex series of graph operations. It can be shown that converting G between representations requires no more than $O\left(|V|^2\right)$ operations.

2.1 Representing Graphs as Lists

The most obvious data structure for representing a graph arises directly from its definition. G is described by two simple lists:

- A vertex list $V(G)$, and
- An edge list $E(G)$ which represents each edge as a pair of vertices.

This data structure often forms the basis of the interface between the user and graph algorithms. The reasons are twofold:

- For many people, it is the most "natural" way to describe the connections in a graph.
- It is a relatively efficient way to represent sparse graphs.

As an example, consider the directed graph of Figure 1. It has six vertices. Its vertex list is simply

$$V(G)_{digraph} = \{1, 2, 3, 4, 5, 6\}$$

The directed graph in Figure 1 has ten edges. Its edge list follows.

$$E(G)_{digraph} = \{(1,3),(2,1)(2,3),(2,4),(3,2),(3,5),(4,6),(5,4),(6,4),(6,5)\}$$

The corresponding undirected graph shown in Figure 2 has the same six vertices.

$$V(G)_{undirected} = \{1, 2, 3, 4, 5, 6\}$$

However, it has 16 edges.

$$\begin{aligned} E(G)_{undirected} = \{&(1,2),(1,3),(2,1)(2,3),(2,4),(3,1),(3,2),(3,5),\\ &(4,2),(4,5),(4,6),(5,3),(5,4),(5,6),(6,4),(6,5)\} \end{aligned}$$

2.2 Representing Graphs as Adjaceny Matrices

An alternate representation of G is known as an adjacency matrix. In this data structure, a $|V| \times |V|$ matrix $\mathbf{A}$ is established such that

$$a_{ij} = \begin{cases} 1 & \text{when vertex } i \text{ is adjacent to vertex } j \\ 0 & \text{when vertex } i \text{ is not adjacent to vertex } j \end{cases} \tag{1}$$

The storage required to implement an adjacency matrix is proportional to $|V|^2$. When G is represented as an adjacency matrix, the best time complexity you can expect for graph algorithms is $O\left(|V|^2\right)$. This is the time required to access each element of the matrix exactly one time.

Once again consider the directed graph of Figure 1. Its adjacency matrix follows.

$$A_{digraph} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

The adjacency matrix of the undirected version of the graph (Figure 2) is as follows.

$$A_{undirected} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Observe that the adjacency matrix of an undirected graph is symmetric.

2.3 Representing Sparse Graphs as Adjaceny Lists

An adjacency list is a data structure for representing adjacency relationships in sparse graphs. The adjacency list of G is actually a set of $|V|$ linked lists. Each vertex v is the head of a list. Vertices adjacent to v form the body of each list. If a vertex is not adjacent to any other vertices its list is empty.

The storage required to implement an adjacency list is proportional to $|V| + |E|$. When G is represented as an adjacency list, the best time complexity you can expect for graph algorithms is $O(|V| + |E|)$. The time required to access each vertex and each edge exactly one time. An algorithm whose time complexity is $O(|V| + |E|)$ is sometimes referred to as linear in the size of G.

Figure 7 depicts the adjacency list for the directed graph in Figure 1.

Figure 8 depicts the adjacency list for the undirected graph in Figure 2.

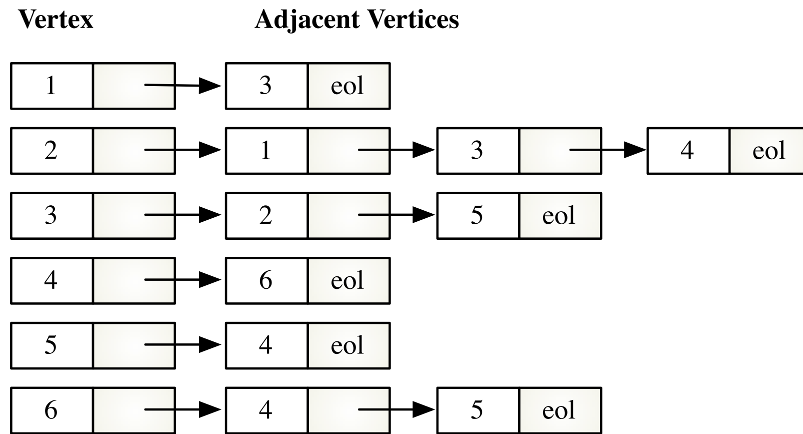

Figure 7: Adjacency List of Directed Graph in Figure 1

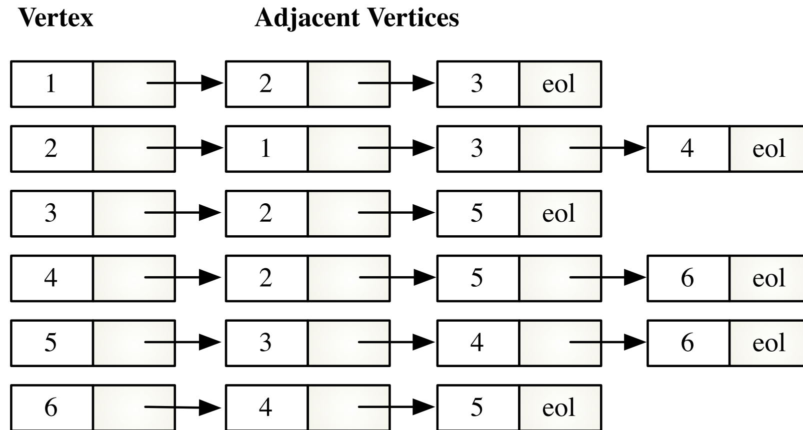

Figure 8: Adjacency List of Undirected Graph in Figure 2

3 Abstract Data Types for Graphs

Graph algorithms in this document are described using an abstract data type paradigm. That is, data sets and operators are specified, but the actual data structures used to implement them remain undefined. Any data structure that efficiently satisfies the constraints imposed in this section is suited for the job.

All signals emitted by operators defined in this section are used to navigate through data, not to indicate errors. Error processing is intentionally omitted from the algorithms in this document. The intent is to avoid clutter that obscures the nature of the algorithms.

3.1 Adjacency List

An adjacency list, A, is a data type for representing adjacency relationships of the sparse graph $G = (V, E)$. Adjacency lists are described in more detail in Section 2.3 of this document.

An adjacency list is typically stored in a dynamic data structure that identifies the edge from vertex i to vertex j as an ordered pair of vertex labels (i, j). Descriptive information is usually associated with each edge.

More specifically, the following operations are supported on an adjacency list A:

- **`Insert`** adds an arbitrary edge (i, j) to A. If edge (i, j) is not already in the list, **`insert`** signals a successful insertion.

- **`Get`** retrieves an arbitrary edge (i, j) from A. When edge (i, j) is in A, **`get`** signals a successful lookup.

- **`Scan`** permits sequential access to all edges incident upon vertex i. Vertex scans are bounded. More specifically, a vertex scan finds all edges (i, j) such that $j_{min} \leq j \leq j_{max}$. When **`scan`** finds edge (i, j), it returns j. When a vertex scan has exhausted all entries in its range, a *finished* signal is emitted.

  A **`scan`** has two support operations: **`push`** and **`pop`**. A **`push`** suspends the scan at its current position. A **`pop`** resumes a suspended scan. The **`push`** and **`pop`** operations permit scans to be nested.

- **`Put`** updates the information associated with an arbitrary edge (i, j) in A.

The algorithms assume that read operations (**`get`** and **`scan`**) make edge information available in a buffer (this buffer is usually denoted by the symbol

e). Update operations (**insert** and **put**) modify the description of an edge based on the current contents of the communication buffer.

Algorithms for creating adjacency lists are examined in Section 4.

3.2 Reduced Graph

A reduced graph, $G' = (V', E')$, is a data structure that supports the systematic elimination of all vertices from the graph $G = (V, E)$. The vertices of the reduced graph are denoted as $V'(G')$ and its edges as $E'(G')$. A crucial attribute of the reduced graph is efficient identification of the vertex in $V'(G')$ with the minimum degree.

A reduced graph supports the following operations:

- **Increase_degree** increases the degree of vertex v in $V'(G')$ by one.
- **Decrease_degree** decreases the degree of vertex v in $V'(G')$ by one.
- **Remove** excises vertex v from $\$V'(G')$.
- **In_graph** tests to see whether vertex v is in $V'(G')$.
- **Minimum_degree** finds the vertex v in $V'(G')$ with the smallest degree.

The topic of vertex elimination and efficient techniques for facilitating the elimination of many vertices are examined in detail in Section 8 of this document.

3.3 List

A simple list L is an ordered set of elements. If the set $\{l_1, \cdots, l_i, l_{i+1}, \cdots, l_n\}$ represents L, then the list contains n elements. Element l_1 is the first item on the list and l_n is the last item on the list. Element l_i precedes l_{i+1} and element l_{i+1} follows l_i. Element l_i is at position i in L. Descriptive information may accompany each item on a list. Lists associated with graph algorithms support the following operations:

- **Link** adds an element x to a list at position i. Inserting element x position i results in an updated list: $\{l_1, \cdots, l_{i-1}, x, l_i, l_{i+1}, \cdots, l_n\}$ An insertion at position *eol* appends x to the end of the list.
- **Unlink** removes the element at position i from the list. Deleting element i results in the list $\{l_1, \cdots, l_{i-1}, l_{i+1}, \cdots, l_n\}$.

- **`Find`** looks for an element on the list and returns its position. If the element is not a member of the list, *eol* is returned.

- **`First`** returns the position of the first item on the list. When the list is empty, *eol* is returned.

- **`Next`** returns position $i + 1$ on the list if position i is provided. When l_i is the last item on the list, *eol* is returned.

- **`Previous`** returns position $i - 1$ on the list if position i is provided. If i is one, *eol* is returned.

A linked list refers to a list implementation that does not require its members to reside in contiguous storage locations. In this environment, an efficient implementation of the **`previous`** operator dictates the use of a doubly linked list.

Communicating with a simple list is analogous to adjacency list communication. Read operations (**`find`**, **`first`**, **`next`**, and **`previous`**) make list information available in a buffer. Update operations (**`link`**, **`unlink`**) modify the list based on the current contents of the buffer.

If it is necessary to distinguish discrete elements in the communication buffer, the structure notation of the C programming language is used. For example, consider a set of edges E maintained in a list whose communication buffer is e. Each edge in E is defined by the pair of vertices (v, w) that constitute its endpoints. In this situation, the endpoints of edge e are referenced as $e.v$ and $e.w$.

3.4 Mapping

A mapping μ relates elements of its domain d to elements of its range r as follows.

$$\mu(d) = r$$

A mapping resides in a data structure that supports two operations:

- **`Map`** links an element r in the range of μ to an arbitrary element d in the domain of μ, i.e. sets $\mu(d)$ to r.

- **`Evaluate`** evaluates the mapping μ for an arbitrary element d in its domain, i.e. returns $\mu(d)$.

Algorithm 1: Create Adjacency List of Undirected Graph

```
i = first(V)
while k ≠ eol
    insert(A, v, header)
    k = next(V, k)
k = first(E)
while k ≠ eol
    insert(A, e.v, e.w)
    insert(A, e.w, e.v)
    k = next(E, k)
```

Algorithm 2: Map Vertices To Labeling Order

```
i = 0
k = first(V)
while k ≠ eol
    i = i + 1
    map(μ, v, i)
    k = next(V, k)
```

4 Creating Adjacency Lists

An undirected graph $G = (V, E)$ is represented by the sets $V(G)$ and $E(G)$. If these sets are maintained as simple lists where the buffer v communicates with the vertex list and e communicates with the edge list, Algorithm 1 creates an adjacency list A for G.

The first loop in Algorithm 1 creates a header for each vertex. This operation is not strictly necessary given the current data definition; however, it is often required in practical implementations.

You should observe that the algorithm inserts each edge in $E(G)$ into the adjacency list twice: once in the stated direction and once in the reverse direction. If G is a directed graph, the reverse insertion is inappropriate (i.e. omit the final insert statement in Algorithm 1). It should be apparent that the algorithm has a time complexity of $O(|V| + |E|)$ if all operators have time complexity of $O(1)$.

It is often beneficial to create an adjacency list based on an alternate labeling of the vertices of G. This operation requires a mapping μ of the integers from 1 to $|V|$ onto the set $V(G)$. Algorithm 2 maps $V(G)$ to the order in which it is enumerated.

Algorithm 3: Create Ordered Adjacency List

```
i = 0
k = first(V)
while k ≠ eol
    i = i + 1
    map(μ, v, i)
    insert(A, v, header)
    k = next(V, k)
k = first(E)
while k ≠ eol
    i = evaluate(μ, e.v)
    j = evaluate(μ, e.w)
    insert(A, i, j)
    insert(A, j, i)
    k = next(E, k)
```

Algorithm 3 combines these two procedures. It labels $V(G)$ according to its order of enumeration and creates an adjacency list based on this labeling.

If **map** and **evaluate** have constant time complexity, the overall algorithm retains its linear complexity.

Note. In a practical implementation, **map** will insert the pair (d, r) into a data structure that is optimized for searching and **evaluate** will look up d in this data structure. Efficient operations of this sort have a time complexity of $O(\log_2 n)$. Therefore, adjacency list creation will have complexity $O(|V| \log_2 |V| + |E| \log_2 |V|)$.

5 Depth First Search

Given a graph $G = (V, E)$ and a vertex $r \in V(G)$, a depth-first search finds all the vertices $v \in V(G)$ for which there is a path from r to v. In other words, a depth-first search finds the connected component of G that contains r. It does so by starting at r and systematically searching G until no more vertices can be reached. After a vertex is reached by the search, it is referred to as visited. A vertex is explored after all adjacent vertices are visited. The vertex r that anchors the search is referred to as its root.

When a depth-first search reaches vertex v, it immediately suspends its exploration of v and explores any unvisited vertex w adjacent to v. The exploration of v resumes only after the exploration of w has finished. The order in which

Algorithm 4: Depth First Search

```
while next(V) ≠ finished
    map(γ, v, unvisited)
depth = dfn = 0
dfs(r)
```

Algorithm 5: Recursive Depth First Search

```
dfs(v)
    dfn = dfn + 1
    map(γ, v, dfn)
    map(δ, v, depth)
    depth = depth + 1
    for all vertices w adjacent to v
        if evaluate(γ, w) is unvisited
            map(ρ, w, v)
            dfs(w)
    depth = depth − 1
```

a depth-first search visits a graph G is a generalization of the order in which a preorder traversal visits the vertices of a tree. If G is a tree, a depth-first search produces a preorder traversal of G.

5.1 Recursive Depth First Search

Algorithm 4 performs a depth-first search of G rooted at r. The counters dfn and $depth$ are global. It relies upon the recursive procedure **dfs**. Algorithm 5 sketches the implementation of the **dfs** procedure.

Observe that Algorithm 5 produces three mappings as it performs the DFS: the depth first ordering γ, the vertex depth map δ, and the predecessor map ρ. These mappings provide useful information concerning the structure of the graph. However, only γ is actually required by the algorithm. It is initialized with all vertices marked as *unvisited* and updated as each node is visited.

The time complexity of a depth-first search is linear in the size of G, i.e. $O(|V| + |E|)$. To achieve this linear time complexity, the implementation of the adjacency list is crucial. More specifically, some indication of the most recently visited neighbor (call it vertex w) must be maintained for each active vertex v. This permits the scan of v's adjacency list to resume at the point where it

Algorithm 6: Depth First Search With Recursion Removed

```
dfs(r)
    visit(r, noparent)
    v = r
    next edge
    for all vertices w adjacent to v
        if evaluate(γ, w) is unvisited
            depth = depth + 1
            visit(w, v)
            v = w
    goto next edge
    if v ≠ r
        v = evaluate(ρ, v)
        depth = depth − 1
        goto next edge
```

Algorithm 7: Non-recursive DFS Vertex Visitation

```
visit(v, predecessor)
    map(δ, v, depth)
    map(ρ, v, predecessor)
    dfn = dfn + 1
    map(γ, v, dfn)
```

was suspended when the exploration of w is finished. If you have to rescan the adjacency list of v to pick up where you left off, linearity is lost.

5.2 Non-recursive Depth First Search

Since the procedure described in Section 5.1 has a depth of recursion of $|V|$ in the worst case, Algorithm 5 may have unacceptable execution characteristics (such as stack overflow) for large graphs. Algorithm 6 removes the recursion from the **dfs** function.

The vertex visitation procedure **visit** is straightforward and is specified in Algorithm 7.

Observe that vertex visitation (Algorithm 7) generates the three mappings (γ, δ, and ρ) that were produced by the recursive procedure. Note that the predecessor mapping ρ is required by the unwound **dfs** procedure. It was not

required by the recursive implementation since equivalent information is implicitly retained on the stack during recursion.

The computational complexity of this unwound procedure is similar to that of the recursive implementation.

When implementing non-recursive depth first search, don't forget that the **`dfs`** function of Algorithm 6 is imbedded in the overall procedure defined by Algorithm 4.

See Section 5.1 for more information on these issues.

5.3 Depth First Spanning Trees

Any edge (v, w) that leads to the discovery of an unvisited vertex during a depth-first search is referred to as a tree edge of $\boldsymbol{G}$. Collectively, the tree edges of $\boldsymbol{G}$ form a depth-first spanning tree of $\boldsymbol{G}$. Tree edges are detected as follows:

- In Algorithm 5, the recursive depth-first search of Section 5.1, only the tree edges will cause **`map`**(ρ, w, v) to execute.
- In Algorithm 6, the non-recursive depth-first search of Section 5.2, only tree edges cause **`visit`**(w, v) to execute.

If $\boldsymbol{G}$ is a directed graph, an edge that connects a vertex to one of its ancestors in the spanning tree is referred to as a back edge. An edge that connects a vertex to one of its descendants in the spanning tree is referred to as a forward edge. If w is neither an ancestor nor descendant of v, then (v, w) is called a cross edge.

If $\boldsymbol{G}$ is an undirected graph, the depth-first spanning tree simplifies as follows:

- Back edges and forward edges are not distinguished.
- Cross edges do not exist.

Therefore, only two types of edge are defined for undirected graphs. We will call them tree edges and back edges.

5.4 Depth First Traversal

Following the lead of Horowitz and Sahni (1978) [2], a distinction is made between a depth-first search (Sections 5.1 and 5.2) and a depth-first traversal. Given a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, a series of depth-first searches that visit all the vertices in $\boldsymbol{V}(\boldsymbol{G})$ is referred to as a depth-first traversal. If $\boldsymbol{G}$ is a connected graph, a

Algorithm 8: Depth First Traversal

```
k = first ( V )
while  k ≠ eol
    map ( γ, v, unvisited )
    k = next ( V, k )
dfn = 0
k = first ( V )
while  k ≠ eol
    if  evaluate ( γ, v )  is unvisited
        depth = 0
        dfs ( v )
  k = next ( V, k )
```

single depth-first search produces a depth-first traversal. Otherwise, a depth-first traversal will discover all of the connected components of G.

Algorithm 8 determines a depth-first traversal of G. It assumes the buffer v is used to communicate with the vertex list.

The time complexity of a depth-first traversal is linear if **dfs** is linear.

6 Graph Structure Analysis

A depth-first search can be used to preprocess a graph or it can be directly embedded in a more complex algorithm. In the latter case, the action taken when a vertex is visited depends on the algorithm in which the search is embedded. In the former case, the normal objective of the search is to gather information about the structure of the graph (as seen from the vantage point of the search).

Recall that Algorithms 5 and 6 gathered three bits of structural information as it proceeded:

- The depth-first labeling $\gamma(v)$ is the order in which v was visited during the search. Depth-first numbers range from 1 to $|V|$. The root is mapped to one.
- The predecessor (or parent) mapping $\rho(v)$ identifies the vertex from which v was discovered during the search. The root has no parent.
- The depth mapping $\delta(v)$ is the length of the path from v to its root in the depth-first spanning tree of G.

Algorithm 9: Extract the Connected Components of a Graph

$k =$ **first** (V)
while $k \neq eol$
    map $(\gamma, v, unvisited)$
    $k =$ **next** (V, k)
$dfn = 0$
$k =$ **first** (C)
while $k \neq eol$
    if **evaluate** (γ, v) is *unvisited*
        $depth = 0$
        dfs (v)
    $k =$ **next** (C, k)

Subsequent sections provide algorithmic examples of structural information about a graph that is obtained by making small alterations to the depth-first search and depth-first traversal algorithms.

6.1 Connected Components of a Graph

If $C(G) \subset V(G)$, a similar procedure (Algorithm 9) extracts a set of components from G whose roots are defined by $C(G)$. It assumes that $C(G)$ and $V(G)$ are maintained in simple lists (Section 3.3 examines the list data type).

6.2 Isolated Vertex Detection

Isolated vertices are detected during a depth-first traversal of a graph. The detection scheme is based on the observation that an isolated vertex processed by **dfs** increases the depth-first number by exactly one. Algorithm 10 implements this observation in the main loop of a depth-first traversal (Algorithm 8). It assumes the buffer v provides communication with the vertex list.

Since the isolated vertex test is an O(1) operation, it does not increase the complexity of a depth-first traversal.

6.3 Cycle Detection

It can be shown that each back edge detected during a depth-first search of the graph $G = (V, E)$ corresponds to a cycle in G. Recall from Section 5.3 that a back edge connects a vertex to one of its ancestors in the spanning tree.

Algorithm 10: Isolated Vertex Detection

```
dfn = 0
k = first (V)
while k ≠ eol
    if evaluate (γ, v)  is unvisited
        depth = 0
        i = dfn
        dfs (v)
        if dfn is i + 1
            ISOLATED VERTEX
    k = next (V, k)
```

Algorithm 11: Cycle Detection in Undirected Graphs

```
next edge
for all vertices w adjacent to v
    if evaluate (γ, w)  is unvisited
        depth = depth + 1
        visit (w, v)
        v = w
    else
        CYCLE DETECTED
    goto next edge
```

A minor modification of the depth-first search will produce a test for cycles. We will examine the case of cycle detection in undirected graphs first. It is simpler.

6.3.1 Detecting Cycles in Undirected Graphs

Algorithm 11 implements a cycle test for undirected graphs that is illustrated on a fragment of Algorithm 6, the non-recursive **dfs** procedure.

Since the cycle test is an $O(1)$ operation, it does not increase the complexity of the depth-first search.

Algorithm 12: Cycle Detection in Directed Graphs

```
next edge
for all vertices w adjacent to v
    if evaluate(γ, w) is unvisited
        depth = depth + 1
        visit(w, v)
        v = w
    else
        if evaluate(γ, v)>evaluate(γ, w) and
           evaluate(γ, r)<evaluate(γ, w)
              CYCLE DETECTED
    goto next edge
```

6.3.2 Detecting Cycles in Directed Graphs

Testing for cycles in directed graphs is still a matter of looking for back edges. However, the process is complicated by the fact that the edges of a digraph that are not part of its spanning forest are not necessarily back edges either. The following observations permit the forward edges, cross edges, and back edges of a depth-first search to be distinguished efficiently.

It can be shown that the forward edges of a depth-first search connect vertices with low depth-first numbers to vertices with high depth-first numbers. Therefore, an edge (v, w) of G that is not part of its spanning forest is a forward edge when $\gamma(v) < \gamma(w)$.

Conversely, (v, w) is a back edge or cross edge of G when $\gamma(v) > \gamma(w)$.

Distinguishing between back edges and cross edges relies on the observation that a cross edge connects the current spanning tree to a spanning tree that was explored at an earlier stage of a depth-first traversal. Since the root r of the current spanning tree must have a larger depth-first number than any vertex in any previously processed tree in the spanning forest, the condition $\gamma(r) > \gamma(w)$ must hold for a cross edge (v, w).

Therefore, an edge (v, w) of G that is not part of its spanning forest is a back edge when the following conditions apply: $\gamma(v) > \gamma(w)$ and $\gamma(r) < \gamma(w)$.

This observation leads to Algorithm 12 for detecting cycles in a directed graph, it is a modification to Algorithm 6, the non-recursive depth-first search.

Obviously, the cycle test does not increase the complexity of the algorithm.

Algorithm 13: Determine Vertex Degree Using Adjacency List

```
k = first(V)
while k ≠ eol
    map(λ, v, 0)
    k = next(V, k)
k = first(V)
while k ≠ eol
    for all vertices w adjacent to v
        n = evaluate(λ, w) + 1
        map(λ, v, n)
    k = next(V, k)
```

Algorithm 14: Determine Vertex Degree of Directed Graph Using Edges

```
k = first(V)
while k ≠ eol
    map(λ, v, 0)
    k = next(V, k)
k = first(E)
while k ≠ eol
    n = evaluate(λ, e.v) + 1
    map(λ, e.v, n)
    k = next(E, k)
```

7 Determining the Degree of Each Vertex

If the graph $G = (V, E)$ is represented by its adjacency list $A(G)$ and vertex list $V(G)$, Algorithm 13 develops a mapping λ whose domain is $V(G)$ and whose range is the degree of $V(G)$. It assumes the buffer v provides communication with the vertex list.

If G is a directed graph, represented by its vertex list $V(G)$ and edge list $E(G)$, Algorithm 14 maps each vertex in $V(G)$ to its degree. It is assumed that each edge is represented by an ordered pair of vertices (v, w) and the buffer e provides communication with the edge list.

If G is an undirected graph, Algorithm 15 performs the same operation. It is assumed that each edge is represented by an unordered pair of vertices (v, w).

If all the elementary operations are $O(1)$, creating λ is linear, i.e. has complexity $O(|V| + |E|)$.

Algorithm 15: Determine Vertex Degree of Undirected Graph Using Edges

```
k = first ( V )
while  k ≠ eol
    map ( λ, v, 0 )
    k = next ( V, k )
k = first ( E )
while  k ≠ eol
    n = evaluate ( λ, e.v ) +1
    map ( λ, e.v, n )
    n = evaluate ( λ, e.w ) +1
    map ( λ, e.w, n )
    k = next ( E, k )
```

8 Vertex Elimination

Eliminating a vertex v from the graph $G = (V, E)$ creates a reduced graph $G' = (V', E')$ with the following characteristics:

- The set of vertices V' is the subset of $V(G)$ that does not contain v.
- The set of edges E' does not contain any edges that were incident upon v.
- The set of edges E' contains edges that connect all vertices in $V(G)$ that were adjacent to v.

In other words, you get G' from G by

1. Removing vertex v from $V(G)$.
2. Removing all edges that were incident upon v from $E(G)$.
3. Adding edges to $E(G)$ that connect all the vertices that were adjacent to v.

8.1 Eliminating a Single Vertex

Assuming that the graph G is represented by a list of vertices $V(G)$ and a list of edges $E(G)$, Algorithm 16 eliminates vertex v from G. It assumes the buffer e provides communication with the edge list.

The operation **adjacency_list** creates an adjacency list from $V(G)$ and $E(G)$. In Section 4, Algorithms 1 and Algorithm 3 were presented to solve this

Algorithm 16: Eliminate a Vertex from a Graph

```
adjacency_list(V, E)
for all vertices w adjacent to v
    for all vertices z adjacent to v
        if w ≠ z and find(E, w, z) is eol
            e.v = w
            e.w = z
            link(E, 1)
    i = find(E, w, z)
    unlink(E, i)
i = find(V, v)
unlink(V, i)
```

problem. New edges created during vertex elimination are arbitrarily inserted at the beginning of the edge list. For the current purposes, the insertion point is irrelevant.

8.2 Eliminating Many Vertices

The vertex elimination procedure outlined in Algorithm 16 of Section 8.1 is primarily illustrative. When multiple vertices are eliminated, it is usually quite inefficient to create an adjacency list each time you want to eliminate a vertex from a graph.

Many of the algorithms discussed in the companion document Matrix Algorithms[1] require an efficient vertex elimination model that supports the sequential elimination of all vertices $V(G)$ from the graph $G = (V, E)$. The procedures are often constrained such that each stage of the process eliminates the minimum degree vertex v from the reduced vertex set $V'(G')$. Additionally, the vertex elimination procedures are often prohibited the modifying of the adjacency list of the graph's $A(G)$. Recall from Section 3.2 that vertex elimination data structures must also support the following operations:

- **Increase_degree** increases the degree of vertex v by one.
- **Decrease_degree** decreases the degree of vertex v by one.
- **Remove** excises vertex v from \$$V'(G')$.

[1] https://vismor.com/documents/network_analysis/matrix_algorithms/

Algorithm 17: Initialize Minimum Degree Vertex Tracking

```
compute_degree ( A, V, λ )
k = first ( V )
while  k ≠ eol
    x = v
    link ( L, eol )
    k = next ( V, k )
merge_sort ( L, λ )
```

- **In_graph** determines whether vertex v is in $V'(G')$.
- **Minimum_degree** finds the vertex v in $V'(G')$ with the smallest degree.

Subsequent sections of this document examine the implementation of these operations.

8.3 Initializing Minimum Degree Vertex Tracking

The following data structure is proposed to support efficient tracking the minimum degree vertex during sequential vertex elimination:

- A mapping λ between each vertex in $V(G)$ and its degree.
- A list L which orders $V'(G')$ by ascending degree.

Algorithm 17 initializes this data structure for the graph G based on its adjacency list $A(G)$ and vertex list $V(G)$. Observe that, per the comments in Section 8.2, $A(G)$ is unchanged by this operation.

The procedure **compute_degree** creates the vertex degree mapping λ based on $A(G)$ and $V(G)$. Algorithm 13 is a candidate for **compute_degree**.

The procedure **merge_sort** sorts the list L based on the mapping λ. Its name suggests that a merge sort may be the appropriate sorting algorithm. The minimum number of comparisons required to sort a set of n keys is $n \ln n$. The merge sort algorithm is $O(n \ln n)$ in both its best and worst case and is particularly suited for use with linked lists. Furthermore, merge sort does not suffer from the potentially catastrophic worst case stack growth that is common in recursive sorting algorithms. The depth of recursion of a merge sort is bounded by $\log n$. This implies that sorting 1,000,000 items would require no more than 20 nested function calls. An efficient implementation of merge sort needs at most 4 to 8 bytes of local stack space for each level of recursion.

Algorithm 18: Increase the Degree of a Vertex

```
increase_degree(v)
    n = evaluate(λ, v) + 1
    map(λ, v, n)
    i = k = find(L, v)
    while k ≠ eol and n > evaluate(λ, k)
        k = next(L, k)
    link(L, k, v)
    unlink(L, i)
```

Algorithm 19: Decrease the Degree of a Vertex

```
decrease_degree(v)
    n = evaluate(λ, v) − 1
    map(λ, v, n)
    i = k = find(L, v)
    while k ≠ eol and n < evaluate(λ, k)
        k = previous(L, k)
    link(L, k, v)
    unlink(L, i)
```

8.4 Maintaining the Reduced Graph

With the vertex elimination data structure established, maintenance and query operations are straightforward. Algorithm 18 implements **increase_degree**. It assumes the list communication buffer is x.

In words, Algorithm 18 increases the degree of v and scans the degree list L in ascending order until the new position of v is located. It then adds v into L at this position and deletes v from the old position. In the worst case, the loop in Algorithm 18 is an $O(|V'|)$ operation. This degenerate case only occurs when all vertices in $V'(G')$ have the same degree and v is the first item in L.

Algorithm 19 implements the **decrease_degree** operation. It decreases the degree of v then scans the degree list L in descending order until the new position of v is located. It then adds v into L at this position and deletes v from the old position. Once again, Algorithm 19 is an $O(|V'|)$ in the worse case. In the average case, it is much closer $O(1)$.

The **remove** operator (Algorithm 20) excises v from the degree list L and makes sure its degree count is less than or equal to zero in the mapping λ.

Algorithm 20: Remove a Vertex from the Reduced Graph

```
remove(v)
    i = find(L, v)
    unlink(L, i)
    map(λ, v, 0)
```

Algorithm 21: Determine if a Vertex is in the Reduced Graph

```
in_graph(v)
    if evaluate(λ, v) > 0
        v IS IN GRAPH
    else
        v IS NOT IN GRAPH
```

8.5 Querying the Reduced Graph State

A couple of reduced graph state operations are also specified in Section 8.1. Their implementations tend to fall out trivially from the data structures used for minimum degree vertex tracking.

The **`in_graph`** operator is realized by the mapping λ. If the degree of vertex v is greater than zero, then v is in $V'(G')$. Algorithm 21 formalizes this observation.

In a similar vein, the **`minimum_degree`** operation is realized by the list operation **`first`**. By definition, **`first`**(L) gets the minimum degree vertex. Recall that the minimum degree vertex list L orders the vertices by ascending degree.

References

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1983. 5

[2] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, Maryland, 1978. 5, 20

[3] S. Even, *Graph Algorithms*, Computer Science Press, Rockville, Maryland, 1980. 5