Graphs Data Structures

Introduction

We looked previously at the binary tree data structure, which provides a useful way of storing data for efficient searching. In a binary tree, each node can have up to two child nodes. More general tree structures can be created in which different numbers of child nodes are allowed for each node.

All tree structures are hierarchical. This means that each node can only have one parent node. Trees can be used to store data which has a definite hierarchy; for example a family tree or a computer file system.

Some data need to have connections between items which do not fit into a hierarchy like this. Graph data structures can be useful in these situations. A graph consists of a number of data items, each of which is called a vertex. Any vertex may be connected to any other, and these connections are called edges. Some problems that can be represented by a graph:

- computer networks
- airline flights
- road map
- course prerequisite structure
- tasks for completing a job
- flow of control through a program

The following figure shows a graph in which the vertices are the names of cities in North America. The edges could represent flights between these cities, or possibly Wide Area Network links between them.
Describing graphs

A graph \( G = (V,E) \) is composed of:
- \( V \): set of vertices
- \( E \): set of edges connecting the vertices in \( V \)

An edge \( e = (u,v) \) is a pair of vertices

Example:

\[
V = \{a,b,c,d,e\}
\]
\[
E = \\
\{(a,b),(a,c),(a,d),
\ (b,e),(c,d),(c,e),
\ (d,e)\}
\]

The graph in the figure above is known as an undirected graph.

An undirected graph is complete if it has as many edges as possible – in other words, if every vertex is joined to every other vertex. The graph in the figure is not complete. For a complete graph with \( n \) vertices, the number of edges is \( n(n-1)/2 \). A graph with 6 vertices needs 15 edges to be complete.

Two vertices in a graph are adjacent if they form an edge. For example, Anchorage and Corvallis are adjacent, while Anchorage and Denver are not. Adjacent vertices are called neighbors.

A path is a sequence of vertices in which each successive pair is an edge. For example:

**Anchorage-Billings-Denver-Edmonton-Anchorage**

A cycle is a path in which the first and last vertices are the same and there are no repeated edges. For example:

**Anchorage-Billings-Denver-Flagstaff**

An undirected graph is connected if, for any pair of vertices, there is a path between them. The graph above is connected, while the following one is not, as there are no paths to Corvallis.
A tree data structure can be described as a connected, acyclic graph with one element designated as the root element. It is acyclic because there are no paths in a tree which start and finish at the same element.

**Directed Graphs**

In a directed graph, or digraph, each edge is an ordered pair of vertices – it has a direction defined. The direction is indicated by an arrow:

A **path** in a directed graph **must follow the direction of the arrows**. Note that there are two edges in this example between Denver and Flagstaff, so it is possible to travel in either direction. The following is a path in this graph

*Billings-Denver-Flagstaff*

while the following is not, because there is no edge from Denver to Billings:
A directed graph is connected if, for any pair of vertices, there is a path between them. The following example graph is not connected – can you see why? What single edge could you change to make it connected?

The Adjacency Matrix
An adjacency matrix is a two-dimensional array in which the elements indicate whether an edge is present between two vertices. If a graph has N vertices, the adjacency matrix is an NxN array. Table 1 shows the adjacency matrix for the graph in Figure 1a.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1 Connected and non-connected graphs.

TABLE 1 Adjacency Matrix
The vertices are used as headings for both rows and columns. An edge between two vertices is indicated by a 1; the absence of an edge is a 0. (You could also use Boolean true/false values.) As you can see, vertex A is adjacent to all three other vertices, B is adjacent to A and D, C is adjacent only to A, and D is adjacent to A and B. In this example, the “connection” of a vertex to itself is indicated by 0, so the diagonal from upper left to lower right, A-A to D-D, which is called the identity diagonal, is all 0s. The entries on the identity diagonal don’t convey any real information, so you can equally well put 1s along it, if that’s more convenient in your program.

The Adjacency List

The other way to represent edges is with an adjacency list. The list in adjacency list refers to a linked list of the kind of, “Linked Lists.” Actually, an adjacency list is an array of lists (or sometimes a list of lists). Each individual list shows what vertices a given vertex is adjacent to. Table 2 shows the adjacency lists for the graph of Figure 1a.

<table>
<thead>
<tr>
<th>Vertex</th>
<th>List Containing Adjacent Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B--&gt;C--&gt;D</td>
</tr>
<tr>
<td>B</td>
<td>A--&gt;D</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>A--&gt;B</td>
</tr>
</tbody>
</table>

In this table, the —> symbol indicates a link in a linked list. Each link in the list is a vertex. Here the vertices are arranged in alphabetical order in each list, although that’s not really necessary. Don’t confuse the contents of adjacency lists with paths. The adjacency list shows which vertices are adjacent to—that is, one edge away from—a given vertex, not paths from vertex to vertex.

Traversing a graph

Traversal is the facility to move through a structure visiting each of the vertices once. We looked previously at the ways in which a binary tree can be traversed. Two possible traversal methods for a graph are breadth-first and depth-first.

- Depth First Search
  - Once a possible path is found, continue the search until the end of the path
- Breadth First Search
  - Start several paths at a time, and advance in each one step at a time

Depth-First Search

The depth-first search uses a stack to remember where it should go when it reaches a dead end.

An Example

We’ll discuss the idea behind the depth-first search in relation to Figure 2. The numbers in this figure show the order in which the vertices are visited.
To carry out the depth-first search, you pick a starting point—in this case, vertex A. You then do three things:

- Visit this vertex,
- Push it onto a stack so you can remember it, and
- Mark it so you won’t visit it again.

Next, you go to any vertex adjacent to A that hasn’t yet been visited. We’ll assume the vertices are selected in alphabetical order, so that brings up B. You visit B, mark it, and push it on the stack.

Now what? You’re at B, and you do the same thing as before: go to an adjacent vertex that hasn’t been visited. This leads you to F. We can call this process Rule 1.

**RULE 1**
If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

Applying Rule 1 again leads you to H. At this point, however, you need to do something else because there are no unvisited vertices adjacent to H. Here’s where Rule 2 comes in.

**RULE 2**
If you can’t follow Rule 1, then, if possible, pop a vertex off the stack.

Following this rule, you pop H off the stack, which brings you back to F. F has no unvisited adjacent vertices, so you pop it. Ditto B. Now only A is left on the stack.

A, however, does have unvisited adjacent vertices, so you visit the next one, C. But C is the end of the line again, so you pop it and you’re back to A. You visit D, G, and I, and then pop them all when you reach the dead end at I. Now you’re back to A. You visit E, and again you’re back to A.

This time, however, A has no unvisited neighbors, so we pop it off the stack. But now there’s nothing left to pop, which brings up Rule 3.
**RULE 3**
If you can’t follow Rule 1 or Rule 2, you’re done.

Table 3 shows how the stack looks in the various stages of this process, as applied to Figure 2.

**TABLE 3 Stack Contents During Depth-First Search**

<table>
<thead>
<tr>
<th>Event</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visit A</td>
<td>A</td>
</tr>
<tr>
<td>Visit B</td>
<td>AB</td>
</tr>
<tr>
<td>Visit F</td>
<td>ABF</td>
</tr>
<tr>
<td>Visit H</td>
<td>ABFH</td>
</tr>
<tr>
<td>Pop H</td>
<td>ABF</td>
</tr>
<tr>
<td>Pop F</td>
<td>AB</td>
</tr>
<tr>
<td>Pop B</td>
<td>A</td>
</tr>
<tr>
<td>Visit C</td>
<td>AC</td>
</tr>
<tr>
<td>Pop C</td>
<td>A</td>
</tr>
<tr>
<td>Visit D</td>
<td>AD</td>
</tr>
<tr>
<td>Visit G</td>
<td>ADG</td>
</tr>
<tr>
<td>Visit I</td>
<td>ADGI</td>
</tr>
<tr>
<td>Pop I</td>
<td>ADG</td>
</tr>
<tr>
<td>Pop G</td>
<td>AD</td>
</tr>
<tr>
<td>Pop D</td>
<td>A</td>
</tr>
<tr>
<td>Visit E</td>
<td>AE</td>
</tr>
<tr>
<td>Pop E</td>
<td>A</td>
</tr>
<tr>
<td>Pop A</td>
<td></td>
</tr>
<tr>
<td>Done</td>
<td></td>
</tr>
</tbody>
</table>

The contents of the stack is the **route** you took from the starting vertex to get where you are. As you move away from the starting vertex, you push vertices as you go. As you move back toward the starting vertex, you pop them. **The order in which you visit the vertices is ABFHCDGIE.**

You might say that the depth-first search algorithm likes to get as far away from the starting point as **quickly as possible and returns only when it reaches a dead end.** If you use the term **depth to mean the distance from the starting point,** you can see where the name depth-first search comes from.

**Breadth-First Search**
As we saw in the depth-first search, the algorithm acts as though it wants to get as far away from the **starting point** as quickly as possible. In the breadth-first search, on the other hand, the algorithm likes to **stay as close as possible to the starting point.** It visits all the vertices adjacent to the starting vertex, and only then goes further afield. This kind of search is **implemented using a queue instead of a stack.**
**An Example**
Figure 3 shows the same graph as Figure 2, but here the **breadth-first search** is used. Again, the **numbers indicate the order** in which the vertices are visited.

![Figure 3: Breadth-first search.](image)

**A** is the starting vertex, so you visit it and make it the current vertex. Then you follow these rules:

**RULE 1**
Visit the **next unvisited vertex** (if there is one) that’s adjacent to the current vertex, **mark** it, and **insert it into the queue**.

**RULE 2**
If you can’t carry out Rule 1 because there are **no more unvisited vertices**, remove a vertex from the queue (if possible) and make it the current vertex.

**RULE 3**
If you can’t carry out Rule 2 because the queue is empty, you’re done.

Thus, you first visit all the vertices adjacent to A, inserting each one into the queue as you visit it. Now you’ve visited A, B, C, D, and E. At this point the queue (from front to rear) contains BCDE.

There are no more unvisited vertices adjacent to A, so you **remove B** from the queue and look for vertices adjacent to it. You **find F**, so you **insert it in the queue**. There are no more unvisited vertices adjacent to B, so you **remove C** from the queue. It has no adjacent unvisited vertices, so you **remove D** and **visit G**. D has no more adjacent unvisited vertices, so you **remove E**. Now the queue is FG. You remove F **and visit H**, and then you **remove G and visit I**. Now the queue is HI, but when you’ve removed each of these and found no adjacent unvisited vertices, the queue is empty, so you’re done. Table 2 shows this sequence.
### TABLE 4 Queue Contents During Breadth-First Search

<table>
<thead>
<tr>
<th>Event</th>
<th>Queue (Front to Rear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visit A</td>
<td></td>
</tr>
<tr>
<td>Visit B</td>
<td>B</td>
</tr>
<tr>
<td>Visit C</td>
<td>BC</td>
</tr>
<tr>
<td>Visit D</td>
<td>BCD</td>
</tr>
<tr>
<td>Visit E</td>
<td>BCDE</td>
</tr>
<tr>
<td>Remove B</td>
<td>CDE</td>
</tr>
<tr>
<td>Visit F</td>
<td>CDEF</td>
</tr>
<tr>
<td>Remove C</td>
<td>DEF</td>
</tr>
<tr>
<td>Remove D</td>
<td>EF</td>
</tr>
<tr>
<td>Visit G</td>
<td>EFG</td>
</tr>
<tr>
<td>Remove E</td>
<td>FG</td>
</tr>
<tr>
<td>Remove F</td>
<td>G</td>
</tr>
<tr>
<td>Visit H</td>
<td>GH</td>
</tr>
<tr>
<td>Remove G</td>
<td>H</td>
</tr>
<tr>
<td>Visit I</td>
<td>HI</td>
</tr>
<tr>
<td>Remove H</td>
<td>I</td>
</tr>
<tr>
<td>Remove I</td>
<td></td>
</tr>
<tr>
<td>Done</td>
<td></td>
</tr>
</tbody>
</table>

At each moment, the **queue contains the vertices** that have been visited but whose neighbors have not yet been fully explored. (Contrast this breadth-first search with the depth-first search, where the contents of the stack is the route you took from the starting point to the current vertex.) The nodes are visited in the order **ABCDEFGHI**.

**Weighted Directed Graph**

Sometimes the edges in a graph have numbers, or weights, associated with them. Weights in the example below could be based on, for example, costs of flights, or on WAN bandwidth. A graph like this is called a weighted graph, or a network.
In a network, each path has a total weight. For example, the path Anchorage-Billings-Edmonton has a total weight of 4 + 10 = 14. This is in fact a shorter path than the direct path Anchorage-Edmonton which has a weight of 15.

Finding the shortest path between two vertices is often important in answering questions like “What is the cheapest way to fly from Anchorage to Flagstaff?” or “What is the best way to route WAN traffic between Billings and Edmonton?”

Algorithm for finding the shortest path

DIJKSTRA’S ALGORITHM

Dijkstra's algorithm is called the single-source shortest path. It is also known as the single source shortest path problem. It computes length of the shortest path from the source to each of the remaining vertices in the graph.

The single source shortest path problem can be described as follows:

Let $G= \{V, E\}$ be a directed weighted graph with $V$ having the set of vertices.
The special vertex $s$ in $V$, where $s$ is the source and let for any edge $e$ in $E$, $\text{EdgeCost}(e)$ be the length of edge $e$. All the weights in the graph should be non-negative.

![Directed graph](image)

Directed-weighted graph is a directed graph with weight attached to each of the edge of the graph.
**Dijkstra’s –A Greedy Algorithm**

Greedy algorithms use problem solving methods based on actions to see if there’s a better long term strategy. **Dijkstra’s algorithm** uses the **greedy approach** to solve the single source shortest problem. It repeatedly selects from the unselected vertices, vertex v nearest to source s and declares the distance to be the actual shortest distance from s to v. The edges of v are then checked to see if their destination can be reached by v followed by the relevant outgoing edges.

**DESCRIPTION OF THE ALGORITHM**

Dijkstra’s algorithm works by solving the sub-problem k, which computes the shortest path from the source to vertices among the k closest vertices to the source. For the dijkstra’s algorithm to work it should be **directed-weighted** graph and the edges should be **non-negative**. If the edges are negative then the actual shortest path cannot be obtained.

At the kth round, there will be a set called Frontier of k vertices that will consist of the vertices closest to the source and the vertices that lie outside frontier are computed and put into New Frontier. The shortest distance obtained is maintained in \( sDist[w] \). It holds the estimate of the distance from s to w. Dijkstra’s algorithm finds the next closest vertex by maintaining the New Frontier vertices in a **priority-min queue**.

The algorithm works by keeping the shortest distance of vertex v from the source in an array, sDist. The shortest distance of the source to itself is zero. \( sDist \) for all other vertices is set to infinity to indicate that those vertices are not yet processed. After the algorithm finishes the processing of the vertices sDist will have the shortest distance of vertex w to s. two sets are maintained Frontier and New Frontier which helps in the processing of the algorithm. Frontier has k vertices which are closest to the source, will have already computed shortest distances to these vertices, for paths restricted upto k vertices. The vertices that resides outside of Frontier is put in a set called New Frontier.
**PSEUDO-CODE OF THE ALGORITHM**

The following pseudo-code gives a brief description of the working of the Dijkstra’s algorithm.

Procedure Dijsktra (V: set of vertices 1... n {Vertex 1 is the source}
    Adj[1…n] of adjacency lists;
    EdgeCost(u, w): edge – cost functions;)
Var: sDist[1…n] of path costs from source (vertex 1); {sDist[j] will be equal to
the length of the shortest path to j}
Begin:
Initialize
{Create a virtual set Frontier to store i where sDist[i] is already fully solved}
Create empty Priority Queue New Frontier;
sDist[1]←0; {The distance to the source is zero}
forall vertices w in V – {1} do {no edges have been explored yet}
sDist[w]←∞
end for;
Fill New Frontier with vertices w in V organized by priorities sDist[w];
endInitialize;
repeat
v←DeleteMin{New Frontier}; {v is the new closest; sDist[v] is already correct}
forall of the neighbors w in Adj[v] do
if sDist[w]>sDist[v] +EdgeCost(v,w) then
sDist[w]←sDist[v] +EdgeCost(v,w)
update w in New Frontier {with new priority sDist[w]}
endif
endfor
until New Frontier is empty
endDijkstra;

**EXAMPLE**
The example will briefly explain each step that is taken and how \( s\text{Dist} \) is calculated.
Consider the following example:

![Figure: Weighted-directed graph](image-url)
The above weighted graph has 5 vertices from A-E. The value between the two vertices is known as the edge cost between two vertices. For example the edge cost between A and C is 1.

Using the above graph the Dijkstra’s algorithm is used to determine the shortest path from the source A to the remaining vertices in the graph.

The example is solved as follows:

• **Initial step**
  
  - sDist[A]=0; the value to the source itself
  - sDist[B]=∞, sDist[C]=∞, sDist[D]=∞, sDist[E]=∞; the nodes not processed yet

• **Step 1**
  
  - Adj[A]={B,C}; computing the value of the adjacent vertices of the graph
  - sDist[B]=4;
  - sDist[C]=1;

![Diagram of graph with labels and edge costs]

*Figure: shortest path to vertices B, C from A*

• **Step 2**
  
  - Computation from vertex C
  - Adj[C] = {B, D};
  - sDist[B] > sDist[C]+ EdgeCost[C,B]
    - 4 > 1+2 (True)
    - Therefore, sDist[B]=3;
    - sDist[D]=2;

![Diagram showing updated sDist values]

*Figure: Shortest path from B, D using C as intermediate vertex*
Adj[B]={E};
sDist[E]=sDist[B]+EdgeCost[B,E]
=3+3=6;

Adj[D]={E};
sDist[E]=sDist[D]+EdgeCost[D,E]
=3+3=6
This is same as the initial value that was computed so sDist[E] value is not changed.

• Step 4
Adj[E]=0; means there is no outgoing edges from E
And no more vertices, algorithm terminated. Hence the path which follows the algorithm is

Figure: Shortest path to E using B as intermediate vertex

Figure: the path obtained using Dijkstra's Algorithm