An Introduction to Graphs

- Graph Problems
- Graph Terminology and Examples
- Representing Graphs
- Topological Sorting
- Graph Searching
Graph Questions

Here are a few of the kinds of questions that can be solved by graph algorithms:

1. Is there a way to take the T from Porter Square to Coolidge Corner?

2. What places can I get to by Bus, without spending more than $.60, from Harvard Station?

3. If the 77 bus stops running, will that change the answer to the previous question?

4. Are there any bus lines whose disruption would change the answer to the second question?

5. What is the route from Harvard Station to Coolidge Corner that requires the fewest scheduled stops?
What is a Graph?

• From a mathematical perspective, a graph is a symmetric, irreflexive relationship defined on elements of a finite, non-empty set.

  This is the essential definition, but it is too abstract for most purposes.

• In more concrete terms, a graph G can be defined as an ordered pair of sets <V,E> where

  • V is a non-empty set of vertices.
    
    V defines the set that the relationship is defined on.

  • E is a set of edges. The set of edges can be empty. An edge is an unordered pair of elements from V.
    
    E defines the relationship defined by the graph.
Graph Vocabulary

What do all those words in the definition mean?

**Symmetric**

A relationship is *symmetric* if for every pair x, y of elements in the set, if x has the relationship with y, then y has the relationship with x. In more concrete terms, if (x,y) is a member of E, the so is (y,x). (In fact, these two edges are identical and ordinarily only one is present.)
An Example of Symmetry - "Friendship"

In the (somewhat idealized) relationship shown in this graph, friendship is a symmetric relationship.

The friendships among a small set of people is illustrated by the following graph:

\[ V = \{\text{Ann, Jill, Bob, Ted}\} \]

\[ E = \{(\text{Ann, Bob}), (\text{Bob, Jill}), (\text{Bob, Ted}), (\text{Ted, Jill})\} \]

- There is no “direction” to these friendships; the fact that Ann is Bob’s friend implies that Bob is Ann’s friend.
Directed Graphs

It is often useful to have graph-like structures that represent relationships that are not symmetric. Such structures are called **directed graphs** or **digraphs**. These are quite common in practice.

- The edges of directed graphs are called *arcs* and are drawn as arrows.
- *In-arcs* come into the vertex, *out-arcs* go out.

In this example, the set $V$ is the same, but now the relationship represented is the "is taller than" relationship.

$V = \{\text{Ann, Jill, Bob, Ted}\}$

$E = \{(\text{Ann, Bob}), (\text{Bob, Jill}), (\text{Ted, Bob}), (\text{Ted, Jill}), (\text{Ted, Ann}), (\text{Ann, Jill})\}$

- Ann is taller than Bob and Jill. Bob is taller than Jill. Ted is taller than Ann, Bob and Jill. Jill is not taller than anyone.
Graph Vocabulary (continued)

**Irreflexive**

A relationship is irreflexive if it is *never* true between any vertex and itself - there exist no edges of the form \((x, x)\).

Once again, it is often useful to have relationships where this property does not hold. Such relationships are called *loop graphs*. Loop graphs can be directed as well, resulting in *loop digraphs*.

**Loop Digraph Example - "Likes car of"**

Everyone likes Ann’s car, including Ann. Bob also likes his own car. Ted and Jill only like Ann’s car.

![Loop Digraph Example](image)

- **Note:** So far, every relationship we have seen has "involved" every member of the set \(V\). This not mandatory! For example we could add a new person to this graph who didn’t like anyone’s car, and whose car nobody likes.
Graph Vocabulary (continued)

**Relationship**

A relationship is, in this context, a binary operation that evaluates to **TRUE** or **FALSE**.

In the context of graphs, a relationship is **TRUE** for \( x, y \) if \((x,y)\) is a member of \( E \).

Once again, this structure is too restrictive for many practical uses. Therefore, the relationship is often extended to include a *real-valued function* (defined only when the relationship is **TRUE**).

- This kind of graph-like structure is called a **network** or **weighted graph**.

- Networks can be directed, reflexive, etc.
Network Example - "Taller Than"

In this example, the “taller-than” graph from a previous example has been extended to include information about *how many inches taller* the taller person is.

- Ted is 1.2" taller than Jill, 1.0" taller than Bob, and 0.5" taller than Ann.

- Ann is 0.7" taller than Jill, and 0.5" taller than Bob.

- Bob is 0.2" taller than Jill.

- Notice that the "is x inches taller" property is transitive in this example (just as we would expect, based on our ordinary intuitions about height). Since Ted is 1.0" taller than Bob, and Bob is 0.2" taller than Jill, Ted is 1.2" taller than Jill. However, it important to note that transitivity is not always obeyed in networks.
For the sake of simplicity...

• Usually when we refer to a graph, we will not distinguish between loop-graphs and ordinary graphs, or networks and ordinary graphs (sticking to Weiss’ terms).

• We will *always* distinguish between undirected and directed graphs, however, because the difference is absolutely essential.

There are an incredible number of variations on the basic graph, and an incredible number of uses. Some interesting books devoted largely to this topic:

*Introductory Graph Theory* - Gary Chartrand

*Applied Combinatorics* - Alan Tucker
A Few More Definitions...

- The **order** of a graph is the number of vertices in the graph. This is often denoted $|V|$.

- The **size** of a graph is the number of edges (or arcs) in the graph. This is often denoted $|E|$.

We will usually talk about the complexity of a graph algorithm in terms of the order or size (or both) of the graph.

- Two vertices are **adjacent** if there is an edge connecting them.

- A **path** is a sequence of vertices such that each consecutive pair of vertices in the sequence are adjacent.
  - The sequence may be empty - Weiss includes a zero-length path from each vertex to itself.

- A **simple path** is a path where each vertex appears at most once, with the exception of the first and last which may be identical.
More definitions...

• A **cycle** in a **directed** graph is a path containing at least one edge such that the path begins and ends at the same vertex. A **simple cycle** is a cycle that contains each vertex except the first and last at most once (and the first and last must be the same).

• A **cycle** in an **undirected** graph is much the same, but contains each edge at most once.

• A graph is **acyclic** if it contains no cycles whatsoever.

• An acyclic directed graph is often called a **DAG** (Directed Acyclic Graph).

• The **length** of a path is the number of edges in the path.
More definitions...

- A graph is **connected** if there exists a path from every vertex to every other vertex.

- A directed graph is **weakly connected** if it would be connected were it not for the direction of its arcs.

---

**Undirected**

- **Connected:**
  - there is a path from A to B and from B to A

**Directed**

- **Weakly Connected:**
  - there is a path from A to B, and if we disregard the direction of the arc, there is a path from B to A

**Not Connected**

- A  
- B
What can you say about these graphs?

1

A B E
C D

2

V W
X Y Z
Some of the properties (i.e. directedness) are easy to spot. Others, like whether a graph is acyclic or connected, require some work to figure out.

Luckily there are algorithms to solve these problems.
Representing Graphs

• How can we actually implement a graph? What data structures make sense?

  There are many answers, depending on what the graph will be used for (what operations will be performed on the graph, etc.).

• We will focus on one representation that works well for a variety of uses - the adjacency matrix (a matrix of what is adjacent to what)).

  1. Vertices are mapped to integers 0..N-1

  2. An N-by-N matrix A is defined such that

     \[ A[i][j] = 1 \text{ if } (v_i, v_j) \text{ is an edge in } E \]

     = 0 otherwise
Example

1. Assign each vertex a number.
   - A = 0
   - B = 1
   - C = 2
   - D = 3

2. Create the adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>0 (A)</th>
<th>1 (B)</th>
<th>2 (C)</th>
<th>3 (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (A)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 (B)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 (C)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3 (D)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Notes

- It can be important to keep the vertex name-to-number mapping in a data structure that allows quick lookups. We'll see in a few lectures that hash tables can be very appropriate for this.

- Sometimes you may want to do a vertex number-to-name lookup- an array of strings works fine.
Representation of the Adjacency Matrix

• Storing the adjacency matrix as an ordinary array is generally very wasteful.
  
  • The size of the array will always be $N^2$, where $N$ is the number of vertices.
  
  • The maximum number of edges possible is $N^2$, and this only occurs in a loop digraph where every vertex is adjacent to itself and every other vertex. This is usually a very uninteresting graph.

• Weiss uses an adjacency list representation. We’ll explore a slightly different approach, because we have already seen **sparse arrays** (Ellard, Chapter 4).

  Our implementation of sparse arrays can be used to combine the convenience of arrays with the efficiency of adjacency lists.

  This requires some shortcuts into the sparse array ADT, but we will do so in the name of simplicity.
Example C code

typedef struct {
    sa_t *adjmat;
    int  edge_count;
    int  vertex_count;
} graph_t;

• Internally, an sa_t represents the array via linked lists. To get a list of all vertices adjacent to vertex x, use:

    adjmat->rows[x]

This gives back a linked list of vertices adjacent to x in O(1) time.

• The structure shown above can represent a loop graph and/or a directed graph. However, it does not provide a natural representation for networks.

• See Ellard, Chapter 4 for details of sparse arrays.
Yet Another Look at the Tennis Problem

• Once again, imagine that we have been given the task of ranking tennis players. Unfortunately, this time you have no control over who plays in each match (or even whether any matches take place at all). All you can do is carefully note the result of matches that have already taken place.

• Can you still rank the tennis players in some way that makes sense?

• Well, sort of - we can model the relationship between players as a graph of who has beaten whom, and topologically sort the vertices.
Topological Sorting

• From Weiss: A **topological sort** is *any* ordering of vertices of a DAG such that if there is a path from x to y then x precedes y in the ordering. Every vertex v in V must appear exactly once in the ordering.

• **The graph must be directed.** Otherwise, for each path there is a "mirror" path in the opposite direction, which violates the ordering.

• **The graph must be acyclic.** If the graph contains a cycle, then every vertex in the cycle precedes every other vertex in the cycle - clearly violating the definition.

• **Note:** loop graphs can be handled as a special exception (by ignoring the loop edges). We will not pursue this; it clutters up the algorithms...

• **Note:** our algorithm for performing a topological sort will detect an error if the directed graph cannot be topologically sorted because it is acyclic, and therefore is useful in determining whether a directed graph is acyclic.
Topological Sort Examples

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

Any permutation of A, B, C is valid, because none violate any constraints!

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

A, C, B or C, A, B

B must follow A and C, but the relative order of A and C is not specified.

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

The ONLY valid order is A, C, B.

B must follow A and C (as before), but now C must follow A.

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

Cyclic

Cannot be topologically sorted
Tennis Results

- Imagine that we construct a directed graph of the relationship "has been beaten by" over a set of female tennis players, and then topologically sort it. Imagine that the result of the topological sort is the following list:

  "has been beaten by"

  Liz, Kathy, Beth, Sue, Ann

What does this imply?

- It does not imply that Liz is the worst, or that Ann is the best. It simply means that Liz has not beaten anyone, and Ann has not lost to anyone. But perhaps Liz and Ann have not played any games yet!

- Relative order in the topological sort implies that the negation of the property (i.e. "has not been beaten by") does hold, but that is all that it implies. (A topological order might actually represent very little useful information.)

Of course, for ranking tennis players, this is not very satisfactory (it might make much more sense to omit players who hadn’t played at least one match from the ordering) but topological sorting does give an ordering consistent with the relation.
Topological Sorting Algorithm

• To topologically sort graph $G = <V,E>$:

1. Find a vertex $v$ in $V$ that has no in-arcs.
   
   If there are none, and the graph is not empty, the graph is not acyclic. In order for the graph to be acyclic, all paths must “begin” somewhere, and therefore there must be at least one vertex that has no in-arcs.

2. Place vertex $v$ at the end of the sorted array.

3. Delete $v$ from $V$ and delete $v$’s out-arcs from $E$.

4. Repeat until $V$ is empty.

An important optimization (explored in Weiss):

Steps #1 and #3 are potentially quite costly (since finding the vertex could require scanning the entire vertex set, and removing the edges could require scanning the edge set). However, we only have to do the vertex scan once because the only vertices with zero in-arcs are vertices that either started with zero in-arcs, or had their last in-arc removed as the result of step #3. Therefore, we can easily keep track of them and update the list of vertices with zero in-arcs as we go.
Topological Sort Example

1. Vertex A has no in-arcs. (We could have chosen D just as easily.)

2. Add A to the end of the "sorted" list.

3. Remove A, and its out-arcs from the graph.

4. V is not empty, so iterate...

1. Vertex D has no in-arcs. (We could choose B just as easily.)

2. Add D to the end of the "sorted" list.

3. Remove D, and its out-arcs from the graph.

4. V is not empty, so iterate...
Topological Sort Example (continued)

1. Vertex B has no in-arcs.
2. Add B to the end of the "sorted" list.
3. Remove B and its out-arcs from the graph.
4. V is not empty, so iterate...(Actually) at this point we can just toss the last vertex into the list.)

- So we have found one possible topological sort - but others were possible.
Graph Questions Redux

1. Is there a way to take the T from Harvard Station to Coolidge Corner?
   
   • Let $G$ be the graph $\langle V, E \rangle$ where $V$ is the set of T stops and $E$ is the set of pairs of T stops one stop distant from each other. Are there any T stops near my home connected to any T stops near Harvard Square? Does there exist a path between $V_{\text{Harvard}}$ and $V_{\text{Coolidge}}$?

2. What places can I get to, via Bus, without spending more than $.60, from Harvard Station?
   
   • What is the set of vertices connected to a given vertex? If I follow every valid simple path starting at Harvard Station, what vertices (T stops) will I visit?

   • Questions #1 and #2 are very similar, and we can solve them by the same basic method - GRAPH SEARCH.

   • In question #1, we are searching the graph in a manner similar to tree search (but without an inorder property to make things easy). In question #2, we search the graph in a manner similar to tree traversal (but without the nice structure of a tree).
Trees and Graphs

It’s worth noting that binary trees (as we have defined them) are actually a special kind of connected, directed, acyclic graph. So within the core of our binary tree traversal algorithms are some ideas we can use to do graph traversal. Unfortunately, we have to abandon our assumptions about the structure of a binary tree:

**binary trees**
- nodes have 0, 1, or 2 children
- each node has exactly one parent, except the root, which has no parent

**graphs and digraphs**
- vertices can have any number of adjacent vertices (out-arcs for digraphs)
- vertices of digraphs can also have any number of in-arcs
- no distinguished "root"
Basic Connected Graph Traversal

The following algorithm will visit all of the vertices connected to the starting vertex.

1. Let \( s \) be some vertex. This is the \textit{start} vertex - perhaps chosen arbitrarily.

2. Let \( S \) be a set of "active" vertices, initially containing only \( s \).

3. Loop until \( S \) is empty, or the "goal" of the traversal has been reached.

   a. Select a vertex \( v \) from \( S \). Remove it from \( S \).

   b. Mark \( v \) as \textit{visited}.

   c. Process \( v \) in whatever way appropriate.

   d. Add all the unvisited vertices adjacent to \( v \) to \( S \).

How \( v \) is selected (step 3.1), and/or how the vertices are added to \( S \) (step 3.4) has a \textit{profound} effect on the order that vertices are visited.
Some Types of Traversal

• If $S$ is a QUEUE, so that vertex selection is done by a DEQUEUE and adding unvisited vertices is done by a sequence of ENQUEUEs, then the traversal is called **Breadth-First**.

• If $S$ is a STACK, so that vertex selection is done by a POP and adding unvisited vertices is done by a sequence of PUSHES, then the traversal is called **Depth-First**.

There are other select/add strategies. We will talk about some of them later - but there really is no end to this. Each algorithm might choose which vertex to select next for reasons known only to itself (i.e. random).
Depth-First Traversal Example

Let s = 2.

2. PUSH

3.a. POP() gives 2.

3.b. Mark 2 as visited.


3.d. PUSH(1) and (3), the vertices adjacent to 2 that have not been visited.

3.a. POP() gives 3.

3.b. Mark 3 as visited.


3.d. PUSH(4), the only vertex adjacent to 3 that has not been visited.

3.a. POP() gives 4.

3.b. Mark 4 as visited.


3.d. Nothing to PUSH.

3.a. POP() gives 1.
3.b. Mark 1 as visited.


3.d. \texttt{PUSH}(0), the only vertex adjacent to 1 that has not been visited.

3.a. \texttt{POP()} gives 0.

3.b. Mark 0 as visited.

3.c. Process 0.

3.d. Nothing to \texttt{PUSH()}. And there’s nothing left in the stack, so we’re done.

So we visit 2, 3, 4, 1, 0

We "dive" as deep as we can in each direction, hence the name \texttt{DEPTH-FIRST}.

Analogous to tree pre-order traversal.
Breadth-First Traversal Example

2. ENQUEUE (2).
3.a. DEQUEUE() gives 2.
3.b. Mark 2 as visited.
3.d. ENQUEUE(1) and ENQUEUE (3), the nodes adjacent to 2 that haven’t been visited.
3.a. DEQUEUE() gives 1.
3.b. Mark 1 as visited.
3.d. ENQUEUE(0), the only node adjacent to 1 that hasn’t been visited.
3.a. DEQUEUE() gives 3.
3.b. Mark 3 as visited.
3.d. ENQUEUE(4).
3.a. DEQUEUE() gives 0.
3.b. Mark 0 as visited.
3.c. Process 0.
3.d. Nothing to ENQUEUE().
3.a. DEQUEUE() gives 4.
3.b. Mark 4 as visited.
3.d. Nothing to ENQUEUE(). And there’s nothing left in the queue, so we’re done.

So we visit 2, 1, 3, 0, 4

We gradually move out "broader and broader" in each direction - hence BREADTH-FIRST.

Analogous to tree level-order traversal.
Analysis of Graph Traversal

Cost of choosing s.

Adding s to S.

Checking whether S is empty, or if some other condition has been met.

Select a vertex v, remove from S.

Mark v as visited.

Process v.

Add all the unvisited vertices adjacent to v to S.

Each of these steps has a cost that depends on some combination of how S is represented, how v is selected, how visited vertices are "marked", as well as how many vertices and edges there are. We could do one analysis for every case, but it would be very messy.
Simple Analysis of Traversal

Let’s try a simple case:

s is known in advance.

S is a QUEUE.

No special ending conditions

Mark is O(1), processing is also O(1).

So steps #1 and #2 are O(1).

Step #3 repeats |V| times, and each iteration costs:

O(1) to see if S is empty

O(1) to Dequeue v, and mark and process it.

O(k) where k is the # of edges from v, to examine each edge and see if it leads to an unvisited vertex.

This assumes that we can find and traverse the list of adjacent vertices in time O(k) - checking for "visited" is O(1) from our earlier assumption.
Analysis (continued)

So what is O(k)? At worst it is O(|E|), but it cannot always be this bad!

The average number of edges emanating from each vertex is:

<table>
<thead>
<tr>
<th></th>
<th>undirected</th>
<th>directed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>E</td>
</tr>
</tbody>
</table>

Since we are doing this for every vertex, the total is going to be:

<table>
<thead>
<tr>
<th></th>
<th>undirected</th>
<th>directed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>E</td>
</tr>
</tbody>
</table>

= O(|E|)

So, the total cost of the graph traversal is O(|V| + |E|), which is O(max(|V|, |E|))