Lecture 29

Graphs I

- Announcements
  - Early Exam Seating
  - Darwin Competition
- Introduction to Graphs
  - Types of Graphs
  - Definitions
- Data Structures for Graphs
  - Adjacency Matrix
  - Adjacency List
Announcements
Hi-

We hope you all enjoyed your Thanksgiving Break and that travel was not too traumatic!

The Fall 2021 Darwin Competition will be held on Tuesday, November 30, 7:30-8:30pm, in Physics 203 (the big lecture hall). We hope you can attend! As is tradition, prizes are very inexpensive replicas of creatures of the animal kingdom.

The competition has the following format:

1. Competitors will be divided into 16 slates of 3-4 creatures apiece.
2. Creatures will play a "match" in pairs: 10 creatures of each species, for 1000 turns. At the end, the majority species wins that match.
3. Each species will compete against every other species in its slate 5 times. The species with the best win-loss-tie record will continue into the next round.

Losers will receive awesome prizes.
4. The winners will move onto the sweet-16 round, of 4 slates. Again, round-robin. 5 matches per pair. Losers receive awesome prizes.
5. The winners of the Sweet 16 round will compete in the final round. Losers will receive a choice of superb or awesome prizes.
6. If there is a tie for first place, participants in the tie will play single matches until a creature wins. The winner will receive a choice of best, superb or simply awesome prizes.
7. Then: Winter.

If you wish to update your creature submission, please make sure to push those changes to your Lab 8 submission by 11:59pm, Monday!

Good luck!
Duane & Aaron
Introduction to Graphs
Types of Graphs
Graphs

Computer science often considers entities and relationships (connections) between pairs of entities:

- Computers and ethernet cables between some pairs of computers.
- Airports and direct flights between some pairs of airports.
- People and friendships between some pairs of people.
- States of a computation and transitions between certain pairs of states.

We model these as *graphs* with *vertices* and *edges* between certain vertex pairs.

Graphs are also known as *networks*.
Computers and ethernet cables between some pairs of computers.

- Vertices: Computers.
- Edges: Pairs of computers that are physically connected by a single cable (not by wifi).
Remember that a graph has edges, which are connections between **pairs** of entities.

- The image on the left is not a graph. For example, is there an edge between ⓐ and ⓑ? It seems that they are connected, but it is not clear which individual edges exist.
- The right image illustrates a variety of graphs, which are known as *network topologies*. The options lead to different costs and performance metrics. The Bus image is not a graph.
Airports and direct flights between some pairs of airports.

- Vertices: Airports.
- Edges: Direct flights from one airport to another airport with distance labels.
People and friendships between some pairs of people.

- Vertices: People in the network.
- Edges: Pairs of people who are friends.
States of a computation and transitions between certain pairs of states.
- Vertices: States of a computation (e.g. solving a puzzle).
- Edges: Moving from one state to another in one “step”.
Discussion: Measuring Graph in Various Ways

What types of concepts or structures would be relevant in each of the previous examples?

- Computer network.
- Flight network.
- Friend network.
- Rubik’s cube graph.

For example, there is a problem in a computer network if two of its computers can’t communicate.

Try to express your answers in terms of the graph itself.

For example, what does a broken network mean in terms of the graph itself.

Most of these concepts and structures will require some type of algorithm to uncover.

Before designing algorithms, we need to clarify our terminology and consider graph data structures.
Definitions
Basic Definitions

In this class we define a graph $G$ as a pair $(V, E)$ where

- $V = V(G)$ is a set of vertices. Each vertex is distinct. Typically, $n$ denotes the number of vertices as in $V = \{v_1, v_2, \ldots, v_n\}$. Vertices are also called nodes.

- $E = E(G)$ is a set of edges. Each edge is a pair of vertices. Typically, $m$ denotes the number of edges as in $E = \{e_1, e_2, \ldots, e_m\}$ where each edge $e \in E$ is written as $(v_i, v_j)$ where $v_i, v_j \in V$.

We draw vertices as dots and edges as lines (or curves) connecting two dots.
Undirected and Directed Graphs

The precise definition of an edge depends on the type of graph:

- In an **undirected graph** the edge \((u,v)\) is the same as the edge \((v,u)\).
  If \((u,v) \in E\), then \((v,u) \in E\) and both representations are equivalent.

- In a **directed graph** the edge \((u,v)\) is not the same as the edge \((v,u)\).
  Thus, \((u,v) \in E\) is independent of \((v,u) \in E\).

In a **simple graph** each \((u,v)\) is in \(E\) at most once and furthermore \(u \neq v\).

**Question**: What is the largest number of edges in a simple graph with \(n\) vertices?

- \(m = n \cdot (n-1) / 2\) in an undirected graph, and \(m = n \cdot (n-1)\) in a directed graph.
Multiple Edges and Loops

A graph is not simple if it has one or more of the following:

- A *loop* is an edge of the form \((v, v)\).
  In other words it is an edge connecting a vertex to itself.
- Multiple edges are duplicate copies of an individual edge \((u,v)\) in \(E\).
  Thus, \(e_i = (u,v)\) and \(e_j = (u,v)\) for \(i \neq j\) and \(e_i, e_j \in E\).

A graph with multiple edges is often called a *multigraph*.
Labeled vs Unlabeled Graphs

A *labeled graph* is a graph where each vertex is given a unique identifier. Two labeled graphs must have the exactly the same set of edges to be the same.

![Labeled Graphs](image)

A labeled graph.
It includes edge (a,c).

A different labeled graph.
It does not included edge (a,c).
Edge-Labeled Graphs (also known as Weighted vs Unweighted Graphs)

Often we label the edges to represent distances, costs, strengths, or weights. These graphs are often called *edge-labeled graphs* or *weighted graph*.

In some cases negative weights make sense.

The label, distance, cost, strength, or weight between u and v is denoted $w_{u,v}$.

- In the example $w_{A,C} = 30$ and $W_{C,A} = 30$ since the graph is undirected. Also, $w_{A,H}$ is undefined.
- Values are not associated with the vertices in this model.
Explicit vs Implicit Graphs

In a computer program a graph is *explicit* if we have stored a full representation of the graph. Otherwise, we may be considering an *implicit graph*, which is conceptual, and is not fully stored.

The Rubik’s Cube has 43 quintillion states.

It can be explored without explicitly creating the full graph.
Degrees

The degree of a vertex is the number of edges that are incident with it. In a directed graph, in-degree / out-degree counts the outgoing / incoming edges.

The degree of vertex c is 4.

The in-degree of vertex c is 3.
The out-degree of vertex c is 1.

The total degree summed over all vertices is $2 \cdot m$.
The total in-degree equals $m$ and the total out-degree equals $m$.

We often let $d$ be the max degree of any vertex in the graph, with $d_{\text{in}}$ and $d_{\text{out}}$ for directed graphs.

For example, the undirected graph above has $d = 4$, while the directed graph has $d_{\text{in}} = 3$ and $d_{\text{out}} = 2$. 
Data Structures for Graphs
Discussion: Data Structures for Graphs

How can we store a graph as a data structure?

- How is a vertex stored?
- How is an individual edge stored?

Think about this for 30 seconds.
Then discuss it with your neighbor for 2 minutes.

Try to think of at least two different approaches.

- Will there be any time and/or space tradeoffs between these approaches?
Adjacency Matrix vs Adjacency Lists

Graphs are typically stored using *adjacency matrices* or *adjacency lists*.

- An adjacency matrix is a matrix (i.e., 2-dimensional array). It stores 0s and 1s for simple graphs, \[ M \]
- An adjacency list is an array of linked lists.

The two types of representations have their own pros and cons.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

An adjacency matrix. \[ M[a][b] \] is 1 if \( ab \) is an edge.

<table>
<thead>
<tr>
<th></th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b, c</td>
</tr>
<tr>
<td>b</td>
<td>a, c</td>
</tr>
<tr>
<td>c</td>
<td>a, b, d</td>
</tr>
<tr>
<td>d</td>
<td>c, e, f</td>
</tr>
<tr>
<td>e</td>
<td>d, f</td>
</tr>
<tr>
<td>f</td>
<td>d, e</td>
</tr>
</tbody>
</table>

An adjacency list. \[ L[a] \] is the linked list of neighbors for vertex \( a \).

In this example, each edge is noted twice (e.g. \( M[a][b] = M[b][a] = 1 \) as well as \( a: b, \ldots \) and \( b: a, \ldots \)).

- If we store each edge once in an adjacent matrix, then we can limit it to be upper triangular \( \nabla \).
- If we store each edge once in a linked list, then we need to scan both lists.
Adjacency Matrices

Adjacency matrices can be used for undirected or directed graphs that are simple or non-simple.

- In undirected graphs each edge is usually stored twice.
- In directed graphs each edge is stored once.
- Non-simple graphs use *adjacency counts* instead of 0/1 in the adjacency matrix.

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Directed non-simple graph.

Adjacency matrix.
Adjacency Lists

Adjacency lists can be used for undirected or directed graphs that are simple or non-simple.

- In undirected graphs each edge is usually stored twice (i.e. in the adjacency list for both vertices).
- Non-simple graphs list neighbors more than once.

In directed graphs only the outgoing lists are stored, or both outgoing and incoming lists are stored.

Each adjacency list can be a singly-linked or doubly-linked list.
Example Operation: Delete Vertex

Consider the deletion of vertex v from a graph.

- Does it matter if the graph is undirected or directed?
- What about out-lists vs out-lists and in-lists?

Analysis

- Adjacency matrix is $O(n)$-time. (Travel along the row and column associated with v.)
- Adjacency list is $O(n+m)$-time or $O(d^2)$-time. (Need to traverse v's list and the lists of other vertices.)
Efficiency of Operations
The method of storing a graph affects the efficiency of operations and algorithms running on it.

Use these types of tables as guidelines. We can often argue that the operations are more efficient.

- The graph may have special structure (e.g., in a 3-regular graph all vertices have degree $d = 3$ which is $O(1)$).
- We may be storing additional information that improves the efficiency (e.g. degree[v] or v.degree).
- In applications that only query the graph (i.e. no add / delete), we may use both types of representations.

We’ll also see that the structure package has a small quirk with its implementations.