Graphs

What is a graph?

A graph is a set of vertices $V$ and a set of edges $(u,v) \in E$ where $u,v \in V$
Graphs

How do graphs differ? What are graph characteristics we might care about?

Different types of graphs

Undirected – edges do not have a direction

Different types of graphs

Directed – edges do have a direction

Different types of graphs

Weighted – edges have an associated weight
Different types of graphs

Weighted – edges have an associated weight

Terminology

Path – A path is a list of vertices $p_1, p_2, \ldots, p_n$ where there exists an edge $(p_i, p_{i+1}) \in E$
Path — A path is a list of vertices $p_1, p_2, ..., p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$.

A simple path contains no repeated vertices (often this is implied).

Cycle — A subset of the edges that form a path such that the first and last node are the same.

Edges: $(A,B), (B,D), (D,A)$
Path: B, A, D, B
Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

Does this graph have a cycle?

Cycle – A subset of the edges that form a path such that the first and last node are the same

not a cycle
Cycle – A path $p_1, p_2, \ldots, p_k$ where $p_1 = p_k$

Connected – every pair of vertices is connected by a path

Is this graph connected?
**Terminology**

*Connected* – every pair of vertices is connected by a path

- not connected

---

**Terminology**

*Strongly connected (directed graphs)* – Every two vertices are reachable by a path

- Is this graph strongly connected?

---

**Terminology**

*Strongly connected (directed graphs)* – Every two vertices are reachable by a path

- not strongly connected

---

**Terminology**

*Strongly connected (directed graphs)* – Every two vertices are reachable by a path

- Is this graph strongly connected?
Terminology

Strongly connected (directed graphs) —
Every two vertices are reachable by a path

Is this graph strongly connected?

Different types of graphs

What is a tree (in our terminology)?
Different types of graphs

Tree – connected, undirected graph without any cycles

DAG – directed, acyclic graph

need to specify root
Different types of graphs

Complete graph – an edge exists between every node

A
B
C
D
E
F

Bipartite graph – a graph where every vertex can be partitioned into two sets X and Y such that all edges connect a vertex \( u \in X \) and a vertex \( v \in Y \)

A
B
C
D
E
F
G

When do we see graphs in real life problems?

- Transportation networks (flights, roads, etc.)
- Communication networks
- Web
- Social networks
- Circuit design
- Bayesian networks

Representing graphs
Representing graphs

Adjacency list – Each vertex \( u \in V \) contains an adjacency list of the set of vertices \( v \) such that there exists an edge \( (u,v) \in E \)

\[
\begin{align*}
A: & \quad B \quad D \\
B: & \quad A \quad D \\
C: & \quad D \\
D: & \quad A \quad B \quad C \quad E \\
E: & \quad D
\end{align*}
\]

Representing graphs

Adjacency matrix – A \( |V| \times |V| \) matrix \( A \) such that:

\[
a_{ij} = \begin{cases} 
1 & \text{if } (i,j) \in E \\
0 & \text{otherwise}
\end{cases}
\]

\[
\begin{array}{ccccc}
A & B & C & D & E \\
\hline
A & 0 & 1 & 0 & 1 & 0 \\
B & 1 & 0 & 0 & 1 & 0 \\
C & 0 & 0 & 0 & 1 & 0 \\
D & 1 & 1 & 1 & 0 & 1 \\
E & 0 & 0 & 0 & 1 & 0
\end{array}
\]
Representing graphs

Adjacency matrix – A $|V| \times |V|$ matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

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

Is it always symmetric?
Adjacency list vs. adjacency matrix

<table>
<thead>
<tr>
<th>Adjacency list</th>
<th>Adjacency matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pros and cons?</td>
<td></td>
</tr>
</tbody>
</table>

50

Sparse adjacency matrix

Rather than using an adjacency list, use an adjacency hashtable

52

Can we get the best of both worlds?

Dense graphs
Constant time lookup to discover if an edge exists
Simple to implement
For non-weighted graphs, only requires boolean matrix

51

Sparse adjacency matrix

Constant time lookup
Space efficient
Not good for dense graphs, why?

53
Weighted graphs

**Adjacency list**
- Store the weight as an additional field in the list

![Adjacency list diagram](image)

**Adjacency matrix**
- \( a_{ij} = \begin{cases} \text{weight} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases} \)

\[
\begin{array}{ccccc}
\text{A} & \text{B} & \text{C} & \text{D} & \text{E} \\
0 & 8 & 0 & 3 & 0 \\
8 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 13 & 0 \\
3 & 2 & 10 & 0 & 13 \\
0 & 0 & 0 & 13 & 0 \\
\end{array}
\]

Graph algorithms/questions

- **Graph traversal (BFS, DFS)**
- Shortest path from a to b
  - Unweighted
  - Weighted positive weights
  - Negative/positive weights
- Minimum spanning trees
- Are all nodes in the graph connected?
- Is the graph bipartite?

DFS and BFS

- How are they implemented?
- What would be the result starting at A?
  - If you ask for the children of a node, they’re given in alphabetical order.
- Run-time (in terms of V and E):
  - Adjacency list
  - Adjacency matrix
Search implemented

TreeDFS(v)
  visit(v)
  if not leaf(v) for all c in children(v)
    TreeDFS(v)

BFS

TreeBFS(T)
  Enqueue(Q, Root(T))
  while Enqueue(Q)  
    while Dequeue(Q)  
      v := Dequeue(Q)
      visit(v)
    for all c in children(v)
      Enqueue(Q, c)

DFS

TreeDFS(T)
  Enqueue(Q, Root(T))
  while Enqueue(Q)  
    while Dequeue(Q)  
      v := Dequeue(Q)
      visit(v)
    for all c in children(v)
      Enqueue(Q, c)
TreeDFS(v)
    visit(v)
    if not leaf(v)
        for all c in children(v)
            TreeDFS(v)

Running time of BFS/DFS

Adjacency list
- How many times does it visit each vertex?
- How many times is each edge traversed?
- $\Theta(|V| + |E|)$ – for trees, i.e., assuming a connected graph

Adjacency matrix
- For each vertex visited, how much work is done?
- $\Theta(|V|^2)$ – for trees, i.e., assuming a connected graph
DFS/BFS

Do they visit all of the nodes?

If the graph is connected or strongly connected

DFS/BFS for graphs

What needs to change for graphs?

Need to make sure we don't visit a node multiple times

BFS for graphs

What order will BFS visit starting at A (again, assume children are enumerated alphabetically)?

A B D E C F G

BFS for graphs

What order will BFS visit starting at A (again, assume children are enumerated alphabetically)?

A B D E C F G
DFS on graphs

DFS(G)
1 for all v ∈ V
2 visited[v] ← false
3 for all v ∈ V
4 if visited[v]
5 DFS-Visit(v)

DFS-Visit(v)
1 visited[v] ← true
2 PreVisit(v)
3 for all edges (u, v) ∈ E
4 if visited[v]
5 DFS-Visit(v)
6 PostVisit(v)
DFS on graphs

DFS(G)
1 for all $v \in V$
2 visited[v] = false
3 for all $v \in V$
4 if !visited[v]
5 DFS-Visit(v)

DFS-Visit(u)
1 visited[u] = true
2 PreVisit(u)
3 for all edges $(u, v) \in E$
4 if !visited[v]
5 DFS-Visit(v)
6 PostVisit(u)

DFS for graphs

What order will DFS visit starting at A (again, assume children are enumerated alphabetically)?

DFS(G)
1 for all $v \in V$
2 visited[v] = false
3 for all $v \in V$
4 if !visited[v]
5 DFS-Visit(v)

DFS-Visit(u)
1 visited[u] = true
2 PreVisit(u)
3 for all edges $(u, v) \in E$
4 if !visited[v]
5 DFS-Visit(v)
6 PostVisit(u)

What does DFS do?

- Finds connected components
- Each call to DFS-Visit from DFS starts exploring a new set of connected components
- Helps us understand the structure/connectedness of a graph
Running time of graph BFS/DFS

Nothing changes!

Adjacency list
- $O(|V| + |E|)$

Adjacency matrix
- $O(|V|^2)$

DAGs

Can represent dependency graphs

Topological sort

A linear ordering of all the vertices such that for all edges $(u,v)$
- $E$, $u$ appears before $v$ in the ordering

An ordering of the nodes that “obeys” the dependencies, i.e., an activity can’t happen until its dependent activities have happened

Topological sort

**Topological-Sort1(G)**
1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. **Topological-Sort1(G)**
Topological sort

**Topological-Sort(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort(G)**

![Diagram](image)

82

Topological sort

**Topological-Sort(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort(G)**

![Diagram](image)

83

Topological sort

**Topological-Sort(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort(G)**

![Diagram](image)

84

Topological sort

**Topological-Sort(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort(G)**

![Diagram](image)

85
Topological sort

1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. Topological-Sort1($G$)

Topological sort

1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. Topological-Sort1($G$)
### Running time?

**Topological-Sort**

1. Find a node $v$ with no incoming edges.
2. Delete $v$ from $G$.
3. Add $v$ to linked list.
4. **Topological-Sort**($G$).

### Running time?

**Topological-Sort**($G$).

1. Find a node $v$ with no incoming edges.
2. Delete $v$ from $G$.
3. Add $v$ to linked list.
4. **Topological-Sort**($G$).

### Running time?

**Topological-Sort**($G$).

1. Find a node $v$ with no incoming edges.
2. Delete $v$ from $G$.
3. Add $v$ to linked list.
4. **Topological-Sort**($G$).

### Running time?

**Topological-Sort**($G$).

1. Find a node $v$ with no incoming edges.
2. Delete $v$ from $G$.
3. Add $v$ to linked list.
4. **Topological-Sort**($G$).

**O(|V| + |E|)**

### Running time?

**Topological-Sort**($G$).

1. Find a node $v$ with no incoming edges.
2. Delete $v$ from $G$.
3. Add $v$ to linked list.
4. **Topological-Sort**($G$).

**O(|V| + |E|)**

How many calls? $|V|$
Running time?

**Topological-Sort1(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort1(G)**

**Overall running time?**

\[ O(|V|^2 + |V| |E|) \]

---

Can we do better?

**Topological-Sort1(G)**
1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. **Topological-Sort1(G)**

---

Topological sort 2

**Topological-Sort2(G)**
1. For all edges \( (u, v) \in E \)
2. \( active[v] = active[v] + 1 \)
3. For all \( v \in V \)
4. If \( active[v] = 0 \)
5. Enqueue(S, \( v \))
6. While !empty(S)
7. \( u \leftarrow dequeue(S) \)
8. Add \( u \) to linked list
9. For each edge \( (u, v) \in E \)
10. \( active[v] = active[v] - 1 \)
11. If \( active[v] = 0 \)
12. Enqueue(S, \( v \))

---

Topological sort 2

**Topological-Sort2(G)**
1. For all edges \( (u, v) \in E \)
2. \( active[v] = active[v] + 1 \)
3. For all \( v \in V \)
4. If \( active[v] = 0 \)
5. Enqueue(S, \( v \))
6. While !empty(S)
7. \( u \leftarrow dequeue(S) \)
8. Add \( u \) to linked list
9. For each edge \( (u, v) \in E \)
10. \( active[v] = active[v] - 1 \)
11. If \( active[v] = 0 \)
12. Enqueue(S, \( v \))
Topological sort 2

**TopologicalSort2(G)**

1. for all edges \((u, v) \in E\)
2. \(active[u] \rightarrow active[u] + 1\)
3. for all \(v \in V\)
4. if \(active[v] = 0\)
5. ENQUEUE(S, v)
6. while !EMPTY(S)
7. \(u \leftarrow DEQUEUE(S)\)
8. add \(u\) to linked list
9. for each edge \((u, v) \in E\)
10. \(active[v] \leftarrow active[v] - 1\)
11. if \(active[v] = 0\)
12. ENQUEUE(S, v)

Running time?

How many times do we process each node?
How many times do we process each edge?

\(O(|V| + |E|)\)

Detecting cycles

Undirected graph
- BFS or DFS. If we reach a node we’ve seen already, then we’ve found a cycle

Directed graph
- Call TopologicalSort
- If the length of the list returned \(\neq |V|\) then a cycle exists
**Connectedness**

Given an undirected graph, for every node $u \in V$, can we reach all other nodes in the graph?

**Algorithm + running time**

Run BFS or DFS—Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false.

Running time: $O(|V| + |E|)$

---

**Strongly connected**

Given a directed graph, can we reach any node $v$ from any other node $u$?

Can we do the same thing?

---

**Transpose of a graph**

Given a graph $G$, we can calculate the transpose of a graph $G^T$ by reversing the direction of all the edges.

Running time to calculate $G^T$: $\Theta(|V| + |E|)$

---

**Strongly connected**

Strongly-Connected($G$)

- Run DFS-Visit or BFS from some node $u$
- If not all nodes are visited: return false
- Create graph $G^T$
- Run DFS-Visit or BFS on $G^T$ from node $u$
- If not all nodes are visited: return false
- return true
Is it correct?

What do we know after the first pass?
- Starting at $u$, we can reach every node.

What do we know after the second pass?
- All nodes can reach $u$. Why?
  - We can get from $u$ to every node in $G$, therefore, if we reverse the edges (i.e. $G^r$), then we have a path from every node to $u$.

Which means that any node can reach any other node. Given any two nodes $s$ and $t$ we can create a path through $u$.

![Diagram of a path from s to t through u](image)

Runtime?

Strongly-Connected($G$)
- Run DFS-Visit or BFS from some node $u$
  - If not all nodes are visited: return false
- Create graph $G^r$
- Run DFS-Visit or BFS on $G^r$ from node $u$
  - If not all nodes are visited: return false
  - return true

$O(|V| + |E|)$

Shortest path algorithms

- Dijkstra’s
- Bellman-Ford
- Floyd-Warshall
- Johnson’s