Graph Search

Lecture 15

March 13, 2025

Part I

Graphs Review

Graph Definition

Definition

An undirected graph $G = (V, E)$ is a 2-tuple:

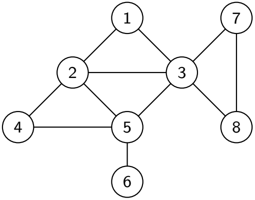
1. V is a set of vertices (also referred to as nodes/points)
2. E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.

Graph Definition

Definition

An undirected graph $G = (V, E)$ is a 2-tuple:

1. V is a set of vertices (also referred to as nodes/points)
2. E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.



Common Shorthands

Writing out the full formal notation can sometimes be cumbersome, so we will often use the following shorthands:

Common Shorthands

Writing out the full formal notation can sometimes be cumbersome, so we will often use the following shorthands:

- uv for an edge $\{u, v\}$

Common Shorthands

Writing out the full formal notation can sometimes be cumbersome, so we will often use the following shorthands:

- uv for an edge $\{u, v\}$
- n or V for the number of vertices $|V|$

Common Shorthands

Writing out the full formal notation can sometimes be cumbersome, so we will often use the following shorthands:

- uv for an edge $\{u, v\}$
- n or V for the number of vertices $|V|$
- m or E for the number of edges $|E|$

Flavors of Graphs: Directed vs Undirected

In a *directed* graph, edges go "from u to v"—order matters!

Flavors of Graphs: Directed vs Undirected

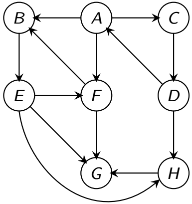
In a *directed* graph, edges go "from u to v"—order matters!

Formal definition: each $e \in E$ is represented by an *ordered* tuple (u, v) with $u, v \in V$ and $u \neq v$.

Flavors of Graphs: Directed vs Undirected

In a *directed* graph, edges go "from u to v"—order matters!

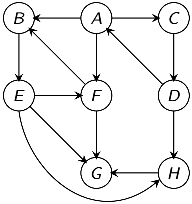
Formal definition: each $e \in E$ is represented by an *ordered* tuple (u, v) with $u, v \in V$ and $u \neq v$.



Flavors of Graphs: Directed vs Undirected

In a *directed* graph, edges go "from u to v"—order matters!

Formal definition: each $e \in E$ is represented by an *ordered* tuple (u, v) with $u, v \in V$ and $u \neq v$.



We will use both directed and undirected graphs in 374, defaulting to undirected if left unspecified.

Flavors of Graphs: Weighted vs Unweighted

In a *weighted* graph, edges have “weights”.

Flavors of Graphs: Weighted vs Unweighted

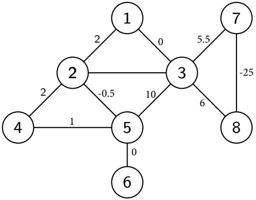
In a *weighted* graph, edges have “weights”.

Formal definition: each edge $e \in E$ has a number $w(e)$ associated with it.

Flavors of Graphs: Weighted vs Unweighted

In a *weighted* graph, edges have "weights".

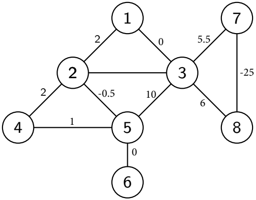
Formal definition: each edge $e \in E$ has a number $w(e)$ associated with it.



Flavors of Graphs: Weighted vs Unweighted

In a *weighted* graph, edges have "weights".

Formal definition: each edge $e \in E$ has a number $w(e)$ associated with it.



We will use both weighted and unweighted graphs in 374, defaulting to unweighted if left unspecified.

Flavors of Graphs: Multi vs Simple

In a *multi*graph, there may be multiple edges between the same pair of vertices, as well as edges between a vertex and itself.

Flavors of Graphs: Multi vs Simple

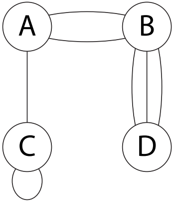
In a *multi*graph, there may be multiple edges between the same pair of vertices, as well as edges between a vertex and itself.

Formal definition: E is a multi-set, and edges $e = \{u, v\}$ may have $u = v$.

Flavors of Graphs: Multi vs Simple

In a *multi*graph, there may be multiple edges between the same pair of vertices, as well as edges between a vertex and itself.

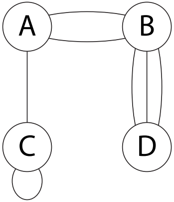
Formal definition: E is a multi-set, and edges $e = \{u, v\}$ may have $u = v$.



Flavors of Graphs: Multi vs Simple

In a *multi*graph, there may be multiple edges between the same pair of vertices, as well as edges between a vertex and itself.

Formal definition: E is a multi-set, and edges $e = \{u, v\}$ may have $u = v$.



We will (almost) exclusively consider simple graphs in 374.

Why Graphs?

Graphs are an extremely useful and widespread model to understand and solve problems in CS, Math, and beyond.

Why Graphs?

Graphs are an extremely useful and widespread model to understand and solve problems in CS, Math, and beyond.

- *Simple* enough to admit nice problem statements, proofs, and algorithms.

Why Graphs?

Graphs are an extremely useful and widespread model to understand and solve problems in CS, Math, and beyond.

- *Simple* enough to admit nice problem statements, proofs, and algorithms.
- *Abstract* enough to allow us to phrase problems we care about in terms of a graph problem, filtering out unneeded details.

Graph Example: Bridges of Köningsberg

In the map below, can we find a route that crosses each bridge *exactly* once?

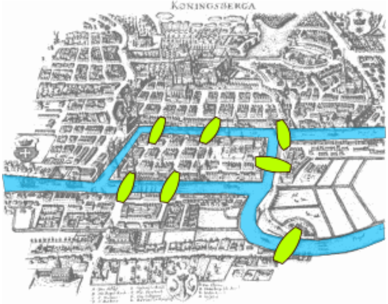


Image source: Wikipedia

Graph Example: Walking to Class

What's the fastest route to walk from Siebel to Lincoln Hall?

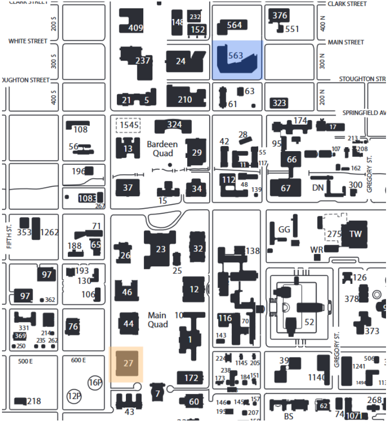


Image source: UIUC

Graphs in 374

For this class, we will be interested in graphs through the lens of computation.

Graphs in 374

For this class, we will be interested in graphs through the lens of computation.

- Lecture: See common graph problems and efficient algorithms to solve them.

Graphs in 374

For this class, we will be interested in graphs through the lens of computation.

- Lecture: See common graph problems and efficient algorithms to solve them.
- Lab / Homework / Exams: Given a problem (not necessarily phrased in terms of graphs), reduce it to a problem on graphs and apply a known algorithm to solve it.

Graphs in 374

For this class, we will be interested in graphs through the lens of computation.

- Lecture: See common graph problems and efficient algorithms to solve them.
- Lab / Homework / Exams: Given a problem (not necessarily phrased in terms of graphs), reduce it to a problem on graphs and apply a known algorithm to solve it.
- "In the wild": Want to use someone else's (optimized, bug-free) implementation of a graph algorithm to solve your particular problem of interest.

Part II

Graph Data Structures

Adjacency List

For each vertex, store a list of all adjacent vertices.

Adjacency List

For each vertex, store a list of all adjacent vertices.

Efficiency for graph with n vertices and m edges?

- Space
- Check adjacency
- Iterate over edges

Adjacency Matrix

Store an $n \times n$ matrix of bits, where $A[i, j] = 1$ iff $\{i, j\} \in E$.

Adjacency Matrix

Store an $n \times n$ matrix of bits, where $A[i, j] = 1$ iff $\{i, j\} \in E$.

Efficiency for graph with n vertices and m edges?

- Space
- Check adjacency
- Iterate over edges

Which Do I Use?

Whether an adjacency list or matrix is better will depend on the problem, as well as how you solve it.

Which Do I Use?

Whether an adjacency list or matrix is better will depend on the problem, as well as how you solve it.

By default, we'll assume that input graphs are represented by adjacency lists, without either of the optimizations.

Which Do I Use?

Whether an adjacency list or matrix is better will depend on the problem, as well as how you solve it.

By default, we'll assume that input graphs are represented by adjacency lists, without either of the optimizations.

- If you want to change to a different representation, make sure your run time analysis accounts for that!

Part III

Connectivity in Undirected Graphs

Definitions

For a graph $G = (V, E)$:

- A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that for all $1 \leq i \leq k - 1$, $\{v_i, v_{i+1}\} \in E$.

Definitions

For a graph $G = (V, E)$:

- A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that for all $1 \leq i \leq k-1$, $\{v_i, v_{i+1}\} \in E$.
  - The length of a path is $k-1$ (the number of edges).
  - A single vertex is a path of length 0.

Definitions

For a graph $G = (V, E)$:

- A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that for all $1 \leq i \leq k-1$, $\{v_i, v_{i+1}\} \in E$.
  - The length of a path is $k-1$ (the number of edges).
  - A single vertex is a path of length 0.
- A cycle is a path with $k \geq 3$ where we additionally have that $\{v_k, v_1\} \in E$.

Definitions

For a graph $G = (V, E)$:

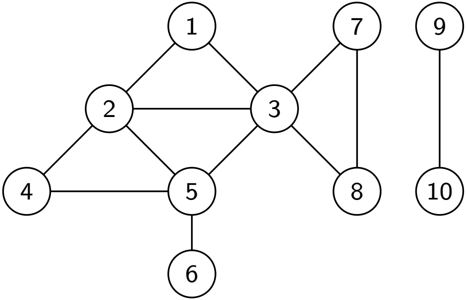
- A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that for all $1 \leq i \leq k - 1$, $\{v_i, v_{i+1}\} \in E$.
  - The length of a path is $k - 1$ (the number of edges).
  - A single vertex is a path of length 0.
- A cycle is a path with $k \geq 3$ where we additionally have that $\{v_k, v_1\} \in E$.
  - The length of a cycle is k (the number of edges).
  - The requirement that $k \geq 3$ ensures that a single edge does not count as a cycle.

Definitions

For a graph $G = (V, E)$:

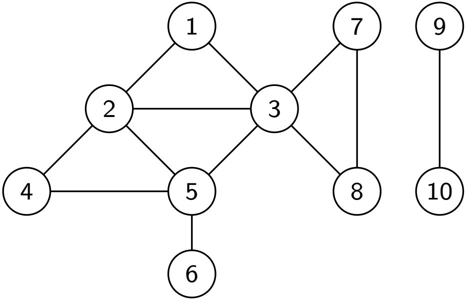
- A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that for all $1 \leq i \leq k - 1$, $\{v_i, v_{i+1}\} \in E$.
  - The length of a path is $k - 1$ (the number of edges).
  - A single vertex is a path of length 0.
- A cycle is a path with $k \geq 3$ where we additionally have that $\{v_k, v_1\} \in E$.
  - The length of a cycle is k (the number of edges).
  - The requirement that $k \geq 3$ ensures that a single edge does not count as a cycle.
- A vertex u is connected to v if there is a path with $v_1 = u$ and $v_k = v$.

Definitions Examples



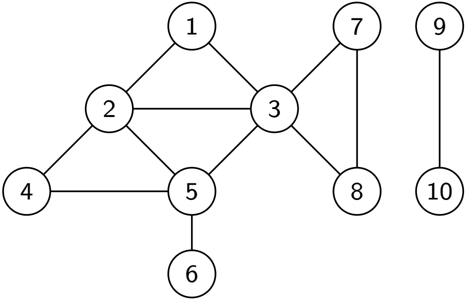
1, 2, 4, 5, 6 is a path.

Definitions Examples



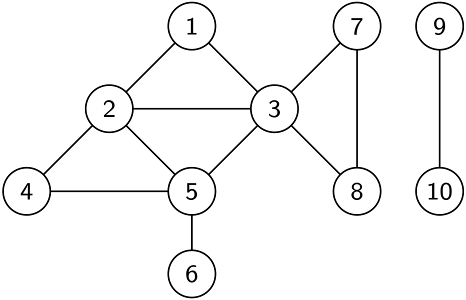
$1, 2, 4, 5, 6$ is a path.
$1, 2, 4, 5, 3$ is a cycle.

Definitions Examples



1, 2, 4, 5, 6 is a path.
1, 2, 4, 5, 3 is a cycle.
1 is connected to **8**.

Definitions Examples



1, 2, 4, 5, 6 is a path.
1, 2, 4, 5, 3 is a cycle.
1 is connected to **8**.
1 is *not* connected to **10**.

Connected Components

For an *undirected* graph, u is connected to v if and only if v is connected to u.

Connected Components

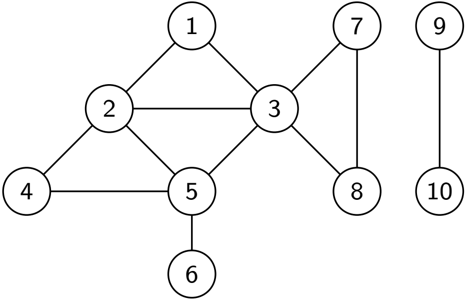
For an *undirected* graph, u is connected to v if and only if v is connected to u.

We define a connected component of a graph $G = (V, E)$ as a maximal set of vertices that are all connected to each other.

Connected Components

For an *undirected* graph, u is connected to v if and only if v is connected to u.

We define a connected component of a graph $G = (V, E)$ as a maximal set of vertices that are all connected to each other.

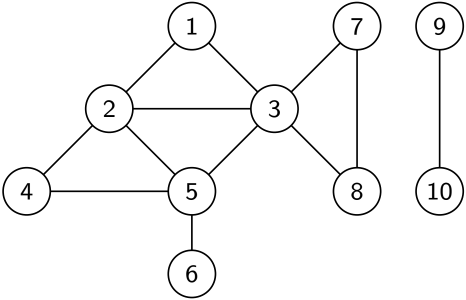


Algorithmic Connectivity

Let $G = (V, E)$ be an undirected graph. We want to solve the following problems:

1. Given nodes u and v, is u connected to v?
2. Given a node u, find the connected component containing u.
3. Find all connected components in G.

Intuitive Example



How would you find the connected component containing 6?

Naïve Search

```
Naïve(G, u):
    S ← {u}
    while (there is edge uv with u ∈ S, v ∉ S) do
        S ← S ∪ {v}
    Output S
```

Naïve Search

```
Naïve(G,u):
    S ← {u}
    while (there is edge uv with u ∈ S, v ∉ S) do
        S ← S ∪ {v}
    Output S
```

Correctness?

Naïve Search

```
Naïve(G,u):
    S ← {u}
    while (there is edge uv with u ∈ S, v ∉ S) do
        S ← S ∪ {v}
    Output S
```

Correctness?

- Never adds a vertex to S that isn't connected to u (induction).

Naïve Search

```
Naïve(G, u):
    S ← {u}
    while (there is edge uv with u ∈ S, v ∉ S) do
        S ← S ∪ {v}
    Output S
```

Correctness?

- Never adds a vertex to S that isn't connected to u (induction).
- Only terminates when no edges between S and the remaining graph, so no path from u can ever exit S.

Naïve Search

```
Naïve(G, u):
    S ← {u}
    while (there is edge uv with u ∈ S, v ∉ S) do
        S ← S ∪ {v}
    Output S
```

Correctness?

- Never adds a vertex to S that isn't connected to u (induction).
- Only terminates when no edges between S and the remaining graph, so no path from u can ever exit S.

Run time?

Naïve Search

```
Naïve(G, u):
    S ← {u}
    while (there is edge uv with u ∈ S, v ∉ S) do
        S ← S ∪ {v}
    Output S
```

Correctness?

- Never adds a vertex to S that isn't connected to u (induction).
- Only terminates when no edges between S and the remaining graph, so no path from u can ever exit S.

Run time? $O(n \cdot m)$
Can we do better?

Whatever First Search

Issue with naïve algorithm: we spend too much time looking for the next vertex to add.

Whatever First Search

Issue with naïve algorithm: we spend too much time looking for the next vertex to add.

Fix: only look at neighbors of added vertices, processed one at a time.

Whatever First Search

Issue with naïve algorithm: we spend too much time looking for the next vertex to add.

Fix: only look at neighbors of added vertices, processed one at a time.

```
WFS(G,u):
    Initialize S = ∅
    Initialize ToExplore = {u}
    while ToExplore is non-empty
        Remove a node x from ToExplore
        if x is not in S
            Add x to S
            for each edge xy in Adj(x)
                If y is not in S, add y to ToExplore
    Output S
```

Efficiency of WFS

```
WFS(G,u):
    Initialize S = ∅
    Initialize ToExplore = {u}
    while ToExplore is non-empty
        Remove a node x from ToExplore
        if x is not in S
            Add x to S
            for each edge xy in Adj(x)
                If y is not in S, add y to ToExplore
    Output S
```

Breadth and Depth First Search

```
WFS(G,u):
    Initialize S = ∅
    Initialize ToExplore = {u}
    while ToExplore is non-empty
        Remove a node x from ToExplore
        if x is not in S
            Add x to S
            for each edge xy in Adj(x)
                If y is not in S, add y to ToExplore
    Output S
```

Not fully specified: what order do we process ToExplore in?
(Equivalently, what data structure is ToExplore?)

Breadth and Depth First Search

```
WFS(G,u):
    Initialize S = ∅
    Initialize ToExplore = {u}
    while ToExplore is non-empty
        Remove a node x from ToExplore
        if x is not in S
            Add x to S
            for each edge xy in Adj(x)
                If y is not in S, add y to ToExplore
    Output S
```

Not fully specified: what order do we process ToExplore in?
(Equivalently, what data structure is ToExplore?)

- Stack (LIFO): Depth-first search

Breadth and Depth First Search

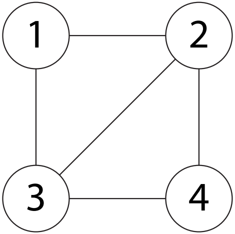
```
WFS(G,u):
    Initialize S = ∅
    Initialize ToExplore = {u}
    while ToExplore is non-empty
        Remove a node x from ToExplore
        if x is not in S
            Add x to S
            for each edge xy in Adj(x)
                If y is not in S, add y to ToExplore
    Output S
```

Not fully specified: what order do we process ToExplore in?
(Equivalently, what data structure is ToExplore?)

- Stack (LIFO): Depth-first search
- Queue (FIFO): Breadth-first search

BFS and DFS Examples

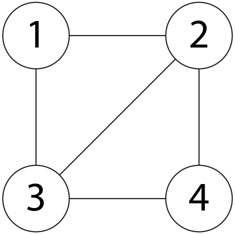
DFS



ToExplore =

{ }

BFS



ToExplore =

{ }

WFS Trees

Each execution of WFS gives us a spanning tree of the connected component containing u: when we add a vertex v to S, include the edge that added v to `ToExplore`.

WFS Trees

Each execution of WFS gives us a spanning tree of the connected component containing u: when we add a vertex v to S, include the edge that added v to `ToExplore`.

(Exercise: modify the pseudocode to also output this tree.)

WFS Trees

Each execution of WFS gives us a spanning tree of the connected component containing u: when we add a vertex v to S, include the edge that added v to `ToExplore`.

(Exercise: modify the pseudocode to also output this tree.)

For **BFS only**, we are guaranteed that the (unique) path in this tree from u to any other vertex uses as few edges as possible.

WFS Trees

Each execution of WFS gives us a spanning tree of the connected component containing u: when we add a vertex v to S, include the edge that added v to `ToExplore`.

(Exercise: modify the pseudocode to also output this tree.)

For **BFS only**, we are guaranteed that the (unique) path in this tree from u to any other vertex uses as few edges as possible.

The tree for **DFS** has some more subtle properties that we'll explore in the next lecture.