

11

Graphen

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

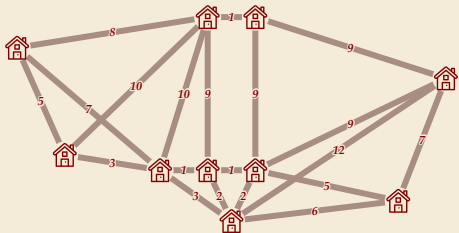
Outline

11 Graphen

- 11.1 Wo(zu er)finden wir Graphen?
- 11.2 Terminologie
- 11.3 Ungerichtete Graphen
- 11.4 Graph Repräsentationen
- 11.5 Tiefensuche
- 11.6 Breitensuche
- 11.7 Gerichtete Graphen
- 11.8 Topologische Sortierung
- 11.9 Starke Zusammenhangskomponenten
- 11.10 Minimale Spannbäume
- 11.11 Kürzeste Wege

11.1 Wo(zu er)finden wir Graphen?

Graphen



- ▶ Ein **Graph** ist eine Abstraktion für Daten, bei denen es (primär) um **Entitäten** und ihre paarweisen **Beziehungen** geht
- ▶ finden sich in vielfältiger Form
- ▶ In Anwendungen auch **Netzwerke** genannt (für uns synonym zu „Graph“)
- ▶ **Beispiele** für Graphen/Netzwerke
 - ▶ **soziale Netzwerke:** z. B. Personen und Freundschaftsbeziehungen, ...
 - ▶ **physische Netzwerke:** Straßen, Stromleitungen, das Internet (Computernetzwerk), ...
 - ▶ **Informationsnetzwerke:** das World Wide Web, Ontologien, Protein-Interaktionen, ...

OK, alles Entitäten und Verbindungen ... aber das sind so unterschiedlichen Bereiche, haben die denn überhaupt irgendwas gemeinsam?

(Ergibt es Sinn, die alle über einen Kamm zu scheren?)

Ja! Abstraktion offenbart wiederkehrenden Fragestellungen aus verschiedenen Bereichen

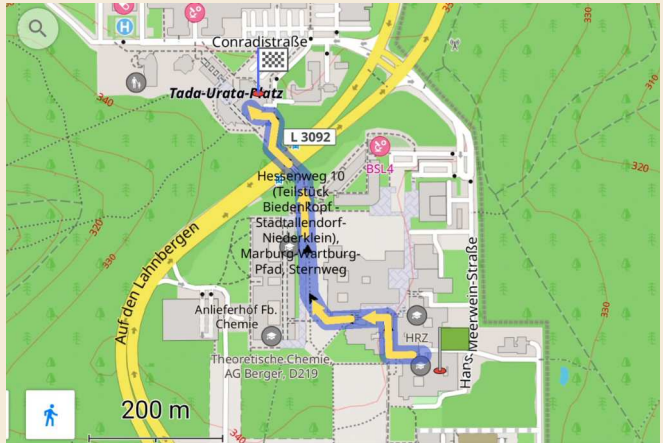
↪ **Graph-Algorithmen liefern wiederverwendbare Lösungen!**

Anwendung 1: Routenplanung

Ziel Finde kürzesten Weg von Start zu Ziel


Gegeben: Straßenkarte, Startpunkt, Zielpunkt

- ▶ Entität = Straßenkreuzung
- ▶ Beziehung = Straße
- ▶ i. d. R. bidirektional
(außer Einbahnstraßen!)
- ▶ Beziehung ist gewichtet
z. B. Fahrzeit



Anwendung 2: Page Rank

1990s Suchmaschinen: erstes Suchergebnis \approx Seite, die Suchbegriff am häufigsten erwähnt

 *Da fällt mir doch eine „Optimierung“ für meine (100% legitime) Website ein . . .*

Google's erste Generation von Ranking: *Page Rank*

▶ Idee wichtige / relevante / vertrauenswürdige Seite \rightsquigarrow viele Links **zu** dieser Seite

\rightsquigarrow *Verwende Links zwischen Seiten selbst als Hauptkriterium für Relevanz!*

▶ ***Random-Surfer-Metapher***

▶ Random Surfer klickt stets auf einen zufällig gewählten Link der aktuellen Seite (oder startet mit kleiner Wahrscheinlichkeit neu auf zufälliger Seite)

▶ Relevanz = Anteil der Zeit auf dieser Seite

\rightsquigarrow cleverer Algorithmus berechnet das als *steady state* einer *Markovkette*

\rightsquigarrow clevere Implementierung schafft das verteilt im Cluster für Milliarden von Seiten

\rightsquigarrow *Google's initialer Erfolg beruht auf der Abstraktion des WWW zu einem Graphen!*

▶ Entität = Website; Beziehung = Hyperlinks

▶ Beziehungen haben feste Richtung (asymmetrisch), aber keine Gewichtung

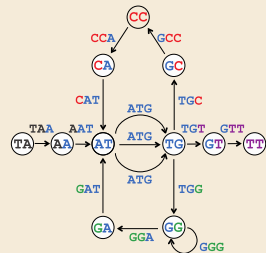
Anwendung 3: de-Bruijn-Graphs, Overlap-Graphs

- ▶ Das **Genom** (eines Individuums) zu **sequenzieren** ist mittlerweile kommerziell möglich
- ▶ Labor-Techniken liefern aber nur kurze „Schnipsel“ des Genoms
 - ▶ 100 – 5000 Basenpaare (Zeichen) pro Read (je nach Technologie)
 - ▶ Menschliches Genom hat etwa 3 000 000 000 Basenpaare! (für einen Chromosomensatz!)
 - ▶ bekommen Tausende Kopien, die an zufälligen Stellen zerbrochen wurden

↪ benötigen Algorithmen, die daraus das Genom **rekonstruieren** (*genome assembly*)

- ▶ Kern-Algorithmen modellieren das Genome-Assembly-Problem als Graph-Problem
 - ▶ Entitäten = Teilstrings
(für jeden Read einmal ohne erstes Zeichen, einmal ohne letztes Zeichen)
 - ▶ Beziehung = Read der Teilstrings verbindet
 - ▶ Beziehung hat Richtung, aber keine Gewichtung
 - ▶ Ziel: eindeutiger Weg durch Graph mit allen Reads

DEBRUIJN₃(**TAATGCCATGGATGTT**)



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig. 4.1
<https://cogniterra.org/lesson/29910/step/2?unit=22007>

11.2 Terminologie

Kernbegriffe

- ▶ **Knoten** (*vertex*) = Entitäten im Graph
 - ▶ Synonyme: Ecke, Punkt; *node*, *point*
 - ▶ Formelsprache: Knoten $v \in V$ (Menge aller Knoten eines Graphen)
- ▶ **Kante** (*edge*) = Beziehung zwischen zwei Knoten
 - ▶ Synonyme: Pfeil, Linie, Relation; *arc*
 - ▶ Formelsprache: Kante $e \in E$ (Menge aller Kanten eines Graphen)
- ▶ Graph = Knoten und Kanten
 - ▶ Synonym: Netzwerk
 - ▶ Formelsprache: $G = (V, E)$

Graphen in allen „Geschmacksrichtungen“

- ▶ Graphen modellieren sehr diverse Entitäten und Beziehungen

↪ mehrere verschiedene (aber wiederkehrende!) Varianten

Eigenschaft	erfüllt	nicht erfüllt
Kanten sind Einbahnstraßen	<i>gerichteter</i> Graph (<i>directed graph, Digraph</i>)	<i>ungerichteter</i> Graph
≤ 1 Kante zwischen u und v	<i>einfacher</i> Graph	<i>Multigraph</i> G. mit <i>parallelen</i> Kanten
Kanten von v zu v	mit <i>Schleifen</i> (<i>loops</i>)	(schleifenfrei)
Gewichtung auf Kanten	(<i>Kanten-</i>) <i>gewichteter</i> G.	<i>ungewichteter</i> Graph

☺ jegliche Kombination kann Sinn ergeben ...

↪ Man muss angeben, welche Art von Graph gemeint ist!

- ▶ Oft sind aber die Algorithmen ähnlich
- ▶ kann verwirren, aber macht Methoden wiederverwertbar

Ungerichtete Graphen – Formale Definition

▶ Standard-Annahme: Graph ist ungewichtet, ungerichtet, schleifenfrei & einfach

▶ *Graph* $G = (V, E)$ mit

▶ V endliche Menge von *Knoten*

▶ $E \subseteq [V]^2$ eine Menge von *Kanten*, mit $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

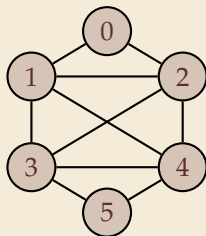
2-elementige Teilmengen

Beispiel

$$V = \{0, 1, 2, 3, 4, 5\}$$

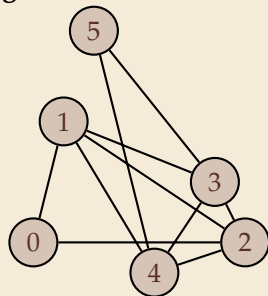
$$E = \{\{0, 1\}, \{1, 2\}, \{1, 4\}, \{1, 3\}, \{0, 2\}, \\ \{2, 4\}, \{2, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}.$$

Graphische Darstellung



so ...

=



... oder auch so

(gleicher Graph!)

Eine „hilfreiche“ Anordnung der Knoten findet man schönsten dynamisch

https://csacademy.com/app/graph_editor

Gerichtete Graphen – Formale Definition

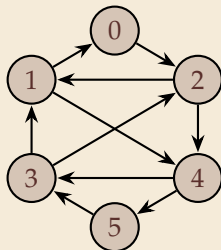
- ▶ Standard-Annahme: Digraph ist ungewichtet, schleifenfrei & einfach
- ▶ *Digraph / Gerichteter Graph* $G = (V, E)$ mit
 - ▶ V endliche Menge von *Knoten*
 - ▶ $E \subseteq V^2 \setminus \{(v, v) : v \in V\}$ Menge (*gerichteter*) *Kanten*,
 $V^2 = V \times V = \{(x, y) : x \in V \wedge y \in V\}$ 2-Tuple / (geordnete) Paare aus V

Beispiel

$$V = \{0, 1, 2, 3, 4, 5\}$$

$$E = \{(0, 2), (1, 0), (1, 4), (2, 1), (2, 4), \\ (3, 1), (3, 2), (4, 3), (4, 5), (5, 3)\}$$

Graphische Darstellung



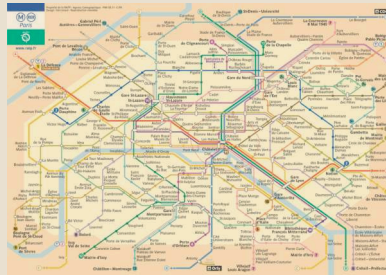
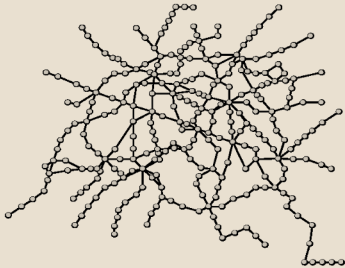
11.3 Ungerichtete Graphen

Undirected graphs

Graph. Set of **vertices** connected pairwise by **edges**.

Why study graph algorithms?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.

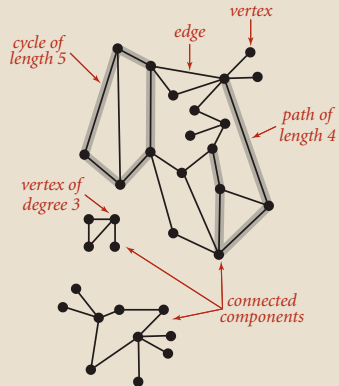


Graph terminology

Path. Sequence of vertices connected by edges.

Cycle. Path whose first and last vertices are the same.

Two vertices are **connected** if there is a path between them.



Some graph-processing problems

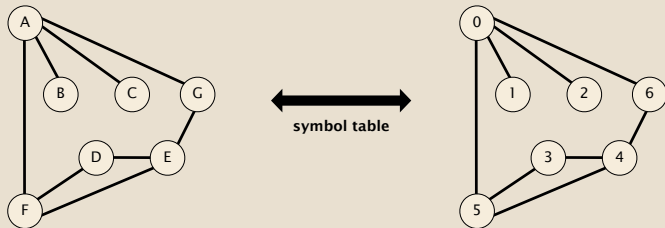
problem	description
s-t path	<i>Is there a path between s and t ?</i>
shortest s-t path	<i>What is the shortest path between s and t ?</i>
cycle	<i>Is there a cycle in the graph ?</i>
Euler cycle	<i>Is there a cycle that uses each edge exactly once ?</i>
Hamilton cycle	<i>Is there a cycle that uses each vertex exactly once ?</i>
connectivity	<i>Is there a way to connect all of the vertices ?</i>
biconnectivity	<i>Is there a vertex whose removal disconnects the graph ?</i>
planarity	<i>Can the graph be drawn in the plane with no crossing edges ?</i>
graph isomorphism	<i>Do two adjacency lists represent the same graph ?</i>

Challenge. Which graph problems are easy? difficult? intractable?

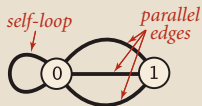
Graph representation

Vertex representation.

- This lecture: use integers between 0 and $V-1$.
- Applications: convert between names and integers with symbol table.



Anomalies.



Graph API

```
public class Graph
```

```
    Graph(int V)
```

create an empty graph with V vertices

```
    Graph(In in)
```

create a graph from input stream

```
    void addEdge(int v, int w)
```

add an edge v-w

```
    Iterable<Integer> adj(int v)
```

vertices adjacent to v

```
    int V()
```

number of vertices

```
    int E()
```

number of edges

```
In in = new In(args[0]);  
Graph G = new Graph(in);  
  
for (int v = 0; v < G.V(); v++)  
    for (int w : G.adj(v))  
        StdOut.println(v + "-" + w);
```

← read graph from
input stream

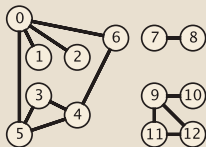
← print out each
edge (twice)

Graph API: sample client

Graph input format.

tinyG.txt

```
V → 13
13 ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```



```
% java Test tinyG.txt
0-6
0-2
0-1
0-5
1-0
2-0
3-5
3-4
:
12-11
12-9
```

```
In in = new In(args[0]);
Graph G = new Graph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);
```

← read graph from
input stream

← print out each
edge (twice)

Typical graph-processing code

```
public class Graph
```

```
    Graph(int V)
```

create an empty graph with V vertices

```
    Graph(In in)
```

create a graph from input stream

```
    void addEdge(int v, int w)
```

add an edge v-w

```
    Iterable<Integer> adj(int v)
```

vertices adjacent to v

```
    int V()
```

number of vertices

```
    int E()
```

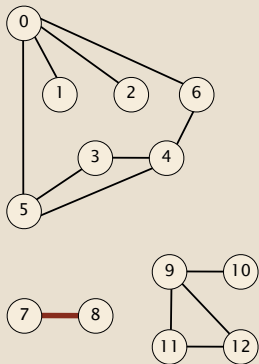
number of edges

```
// degree of vertex v in graph G
public static int degree(Graph G, int v)
{
    int degree = 0;
    for (int w : G.adj(v))
        degree++;
    return degree;
}
```

11.4 Graph Repräsentationen

Set-of-edges graph representation

Maintain a list of the edges (linked list or array).



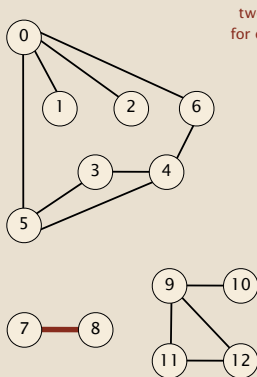
0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Q. How long to iterate over vertices adjacent to v ?

Adjacency-matrix graph representation

Maintain a two-dimensional V -by- V boolean array;

for each edge $v-w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.



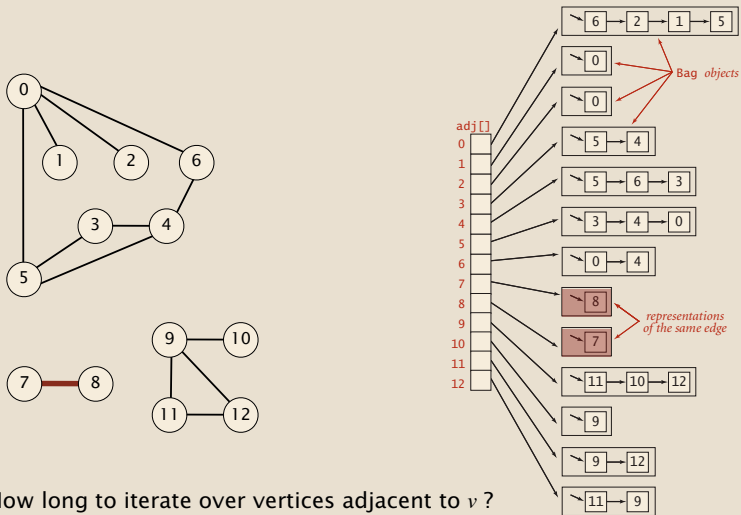
two entries
for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Q. How long to iterate over vertices adjacent to v ?

Adjacency-list graph representation

Maintain vertex-indexed array of lists.



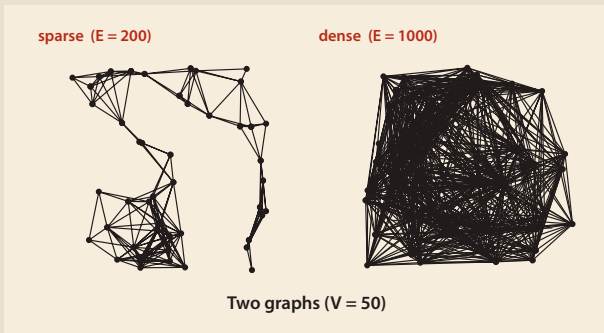
Q. How long to iterate over vertices adjacent to v ?

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v .
- Real-world graphs tend to be **sparse**.

huge number of vertices,
small average vertex degree



Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v .
- Real-world graphs tend to be **sparse**.

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w ?	iterate over vertices adjacent to v ?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	$E + V$	1	$degree(v)$	$degree(v)$

* disallows parallel edges

Adjacency-list graph representation: Java implementation

```
public class Graph
{
    private final int V;
    private Bag<Integer>[] adj;

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w)
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v)
    { return adj[v]; }
}
```

← adjacency lists
(using Bag data type)

← create empty graph
with V vertices

← add edge v-w
(parallel edges and
self-loops allowed)

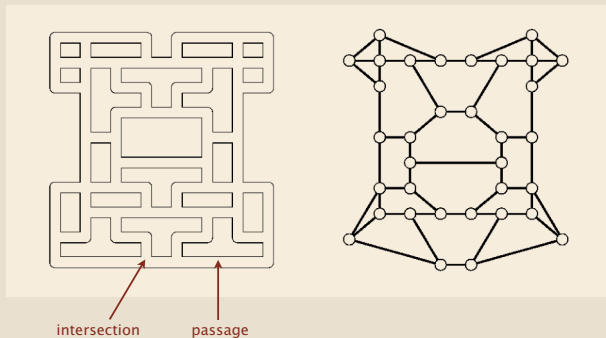
← iterator for vertices adjacent to v

11.5 Tiefensuche

Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.

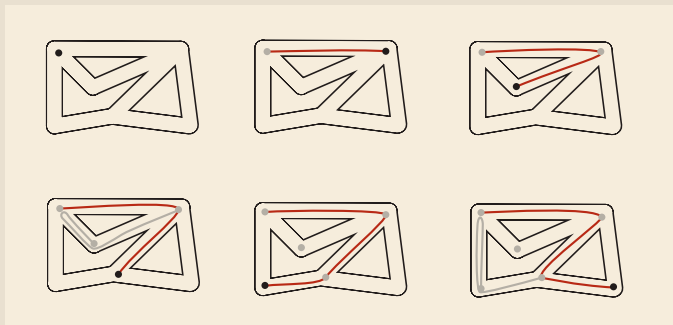


Goal. Explore every intersection in the maze.

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.



Trémaux maze exploration

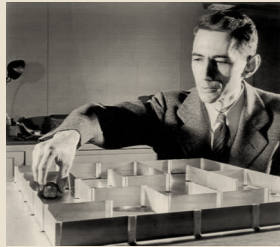
Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.

First use? Theseus entered Labyrinth to kill the monstrous Minotaur; Ariadne instructed Theseus to use a ball of string to find his way back out.

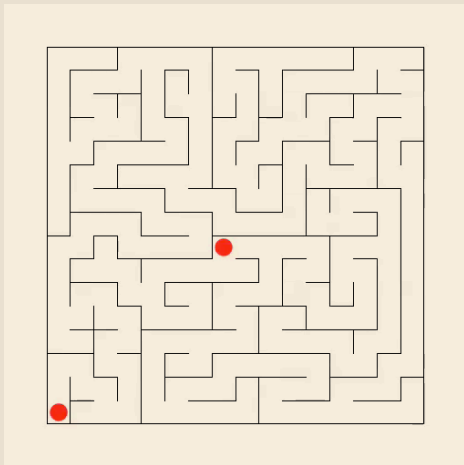


The Labyrinth (with Minotaur)

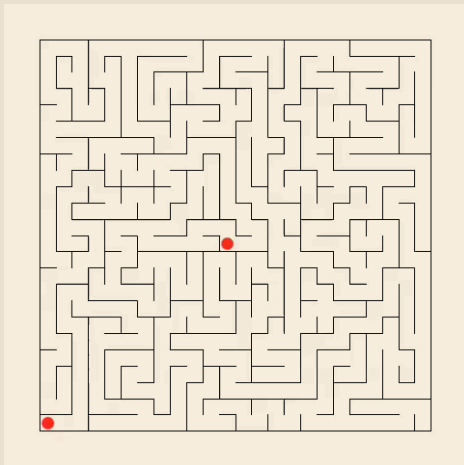


Claude Shannon (with Theseus mouse)

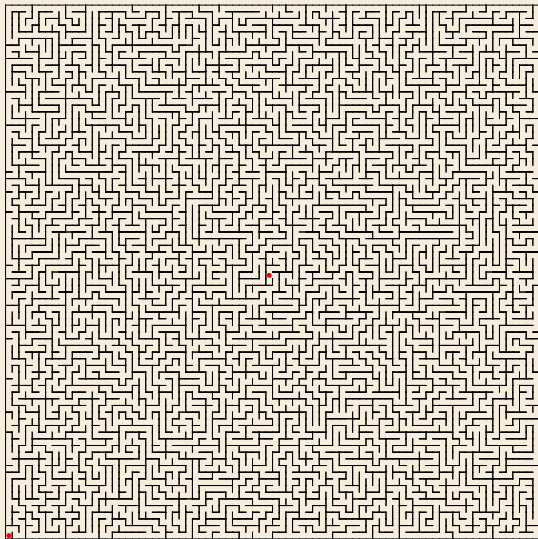
Maze exploration: easy



Maze exploration: medium



Maze exploration: challenge for the bored



Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

- Create a Graph object.
- Pass the Graph to a graph-processing routine.
- Query the graph-processing routine for information.

```
public class Paths
```

```
    Paths(Graph G, int s)           find paths in G from source s
```

```
    boolean hasPathTo(int v)       is there a path from s to v?
```

```
    Iterable<Integer> pathTo(int v) path from s to v; null if no such path
```

```
Paths paths = new Paths(G, s);  
for (int v = 0; v < G.V(); v++)  
    if (paths.hasPathTo(v))  
        StdOut.println(v);
```

← print all vertices
connected to s

Depth-first search: data structures

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

Data structures.

- Boolean array `marked[]` to mark visited vertices.
- Integer array `edgeTo[]` to keep track of paths.
(`edgeTo[w] == v`) means that edge $v-w$ taken to visit w for first time
- Function-call stack for recursion.

Depth-first search: Java implementation

```
public class DepthFirstPaths
```

```
{
```

```
    private boolean[] marked;
```

```
    private int[] edgeTo;
```

```
    private int s;
```

← marked[v] = true
if v connected to s

← edgeTo[v] = previous
vertex on path from s to v

```
    public DepthFirstPaths(Graph G, int s)
```

```
    {
```

```
        ...
```

```
        dfs(G, s);
```

```
    }
```

← initialize data structures

← find vertices connected to s

```
    private void dfs(Graph G, int v)
```

```
    {
```

```
        marked[v] = true;
```

```
        for (int w : G.adj(v))
```

```
            if (!marked[w])
```

```
            {
```

```
                dfs(G, w);
```

```
                edgeTo[w] = v;
```

```
            }
```

```
    }
```

← recursive DFS does the work

```
}
```

Depth-first search: properties

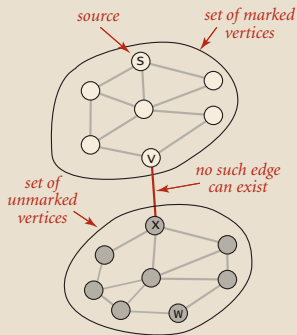
Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees (plus time to initialize the `marked[]` array).

Pf. [correctness]

- If w marked, then w connected to s (why?)
- If w connected to s , then w marked.
(if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one).

Pf. [running time]

Each vertex connected to s is visited once.



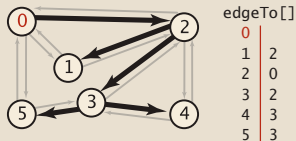
Depth-first search: properties

Proposition. After DFS, can check if vertex v is connected to s in constant time and can find $v \rightarrow s$ path (if one exists) in time proportional to its length.

Pf. `edgeTo[]` is parent-link representation of a tree rooted at vertex s .

```
public boolean hasPathTo(int v)
{ return marked[v]; }

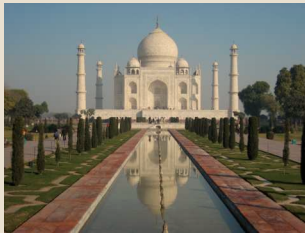
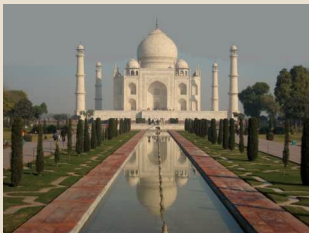
public Iterable<Integer> pathTo(int v)
{
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}
```



Depth-first search application: flood fill

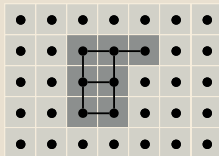
Challenge. Flood fill (Photoshop magic wand).

Assumptions. Picture has millions to billions of pixels.

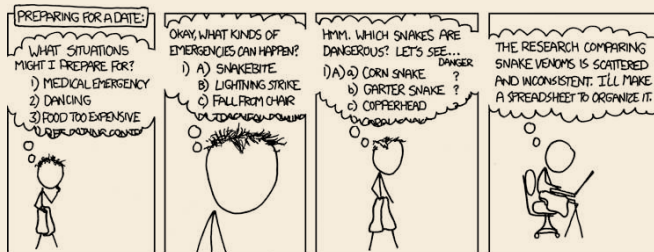


Solution. Build a **grid graph** (implicitly).

- Vertex: pixel.
- Edge: between two adjacent gray pixels.
- Blob: all pixels connected to given pixel.



Depth-first search application: preparing for a date



I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.

xkcd
<http://xkcd.com/761/>

11.6 Breitensuche

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v
 - add each of v 's unvisited neighbors to the queue, and mark them as visited.
-



Breadth-first search: Java implementation

```
public class BreadthFirstPaths
{
    private boolean[] marked;
    private int[] edgeTo;
    private int[] distTo;

    ...
```

```
private void bfs(Graph G, int s) {
    Queue<Integer> q = new Queue<Integer>();
    q.enqueue(s);
    marked[s] = true;
    distTo[s] = 0;

    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                q.enqueue(w);
                marked[w] = true;
                edgeTo[w] = v;
                distTo[w] = distTo[v] + 1;
            }
        }
    }
}
```

initialize FIFO queue of
vertices to explore

found new vertex w
via edge v-w

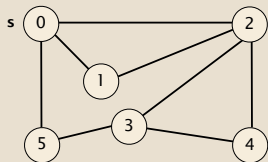
Breadth-first search properties

Q. In which order does BFS examine vertices?

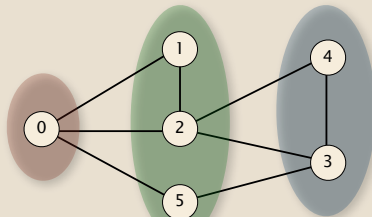
A. Increasing distance (number of edges) from s .

queue always consists of ≥ 0 vertices of distance k from s ,
followed by ≥ 0 vertices of distance $k+1$

Proposition. In any connected graph G , BFS computes shortest paths from s to all other vertices in time proportional to $E + V$.



graph G



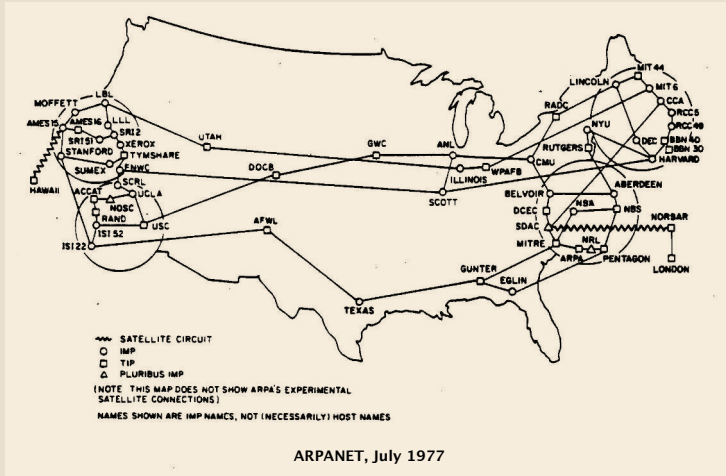
dist = 0

dist = 1

dist = 2

Breadth-first search application: routing

Fewest number of hops in a communication network.



Breadth-first search application: Kevin Bacon numbers

The Oracle of Bacon website interface. At the top, it says "THE ORACLE OF BACON" with a classical bust on the left and a photo of Kevin Bacon on the right. Below this, a search path is displayed:

- Buzz Mauro
- Sweet Dreams (2005)
- Tatiana Ramirez
- Interior de un silencio, El (2005)
- Andres Suarez
- Carlita's Secret (2004)
- Paula Lemes (I)
- Frost/Nixon (2008)
- Kevin Bacon

At the bottom, there is a search bar with "Kevin Bacon" entered, and another search bar with "Buzz Mauro" entered, along with "Find link" and "More options >>>" buttons.

<http://oracleofbacon.org>



Endless Games board game

A screenshot of the SixDegrees iPhone app. The screen shows a search path for "Uma Thurman" with a "2 Degrees" indicator. The path is:

- Uma Thurman
- acted in
- Be Cool (2005) 1°
- with
- Scott Adsit
- who acted in
- The Informant! (2009) 2°
- with
- Matt Damon

At the bottom, there is a navigation bar with icons for "Lookup", "Tiles", "Guess Degrees", and "Scoreboard".

SixDegrees iPhone App

Connectivity queries

Def. Vertices v and w are **connected** if there is a path between them.

Goal. Preprocess graph to answer queries of the form *is v connected to w ?* in **constant** time.

```
public class CC
```

```
    CC(Graph G)
```

find connected components in G

```
    boolean connected(int v, int w)
```

are v and w connected?

```
    int count()
```

number of connected components

```
    int id(int v)
```

*component identifier for v
(between 0 and count() - 1)*

Union-Find? Not quite.

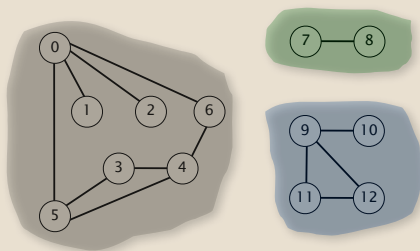
Depth-first search. Yes. [next few slides]

Connected components

The relation "is connected to" is an **equivalence relation**:

- Reflexive: v is connected to v .
- Symmetric: if v is connected to w , then w is connected to v .
- Transitive: if v connected to w and w connected to x , then v connected to x .

Def. A **connected component** is a maximal set of connected vertices.



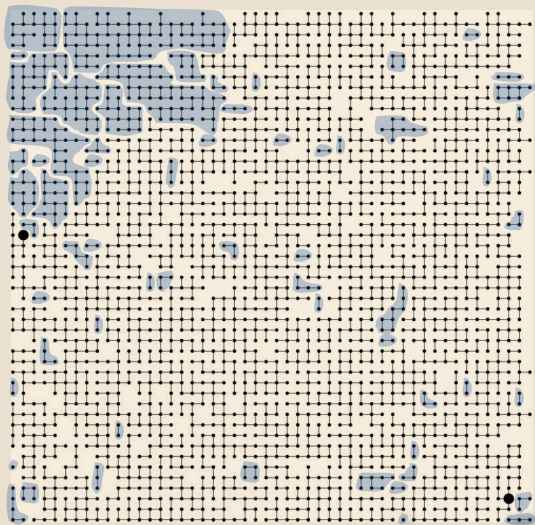
3 connected components

v	id[]
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	2
10	2
11	2
12	2

Remark. Given connected components, can answer queries in constant time.

Connected components

Def. A **connected component** is a maximal set of connected vertices.



63 connected components

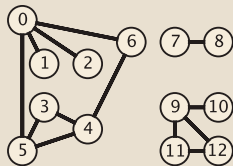
Connected components

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v , run DFS to identify all vertices discovered as part of the same component.



`tinyG.txt`

```
V → 13  
13 ← E  
0 5  
4 3  
0 1  
9 12  
6 4  
5 4  
0 2  
11 12  
9 10  
0 6  
7 8  
9 11  
5 3
```

Finding connected components with DFS

```
public class CC
{
    private boolean[] marked;
    private int[] id;
    private int count;

    public CC(Graph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    public int count()
    public int id(int v)
    public boolean connected(int v, int w)
    private void dfs(Graph G, int v)
}
```

← id[v] = id of component containing v
← number of components

← run DFS from one vertex in each component

← see next slide

Finding connected components with DFS (continued)

```
public int count()  
{ return count; }
```

← number of components

```
public int id(int v)  
{ return id[v]; }
```

← id of component containing v

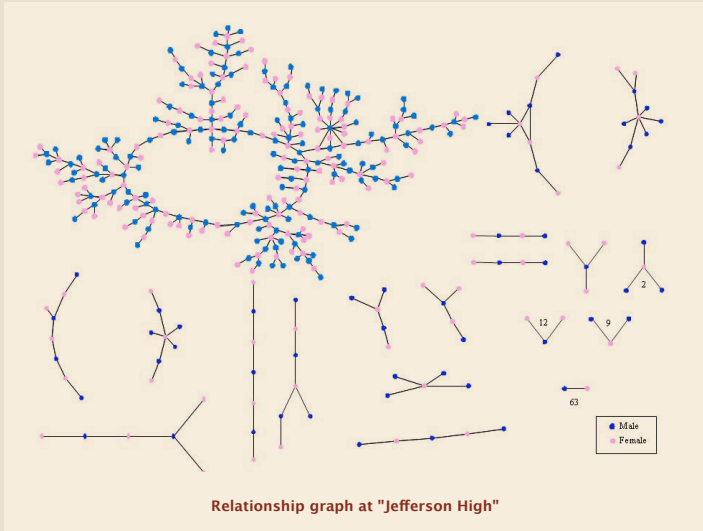
```
public boolean connected(int v, int w)  
{ return id[v] == id[w]; }
```

← v and w connected iff same id

```
private void dfs(Graph G, int v)  
{  
    marked[v] = true;  
    id[v] = count;  
    for (int w : G.adj(v))  
        if (!marked[w])  
            dfs(G, w);  
}
```

← all vertices discovered in
same call of dfs have same id

Connected components application: study spread of STDs

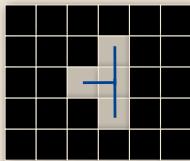


Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. *American Journal of Sociology*, 110(1): 44-99, 2004.

Connected components application: particle detection

Particle detection. Given grayscale image of particles, identify "blobs."

- Vertex: pixel.
- Edge: between two adjacent pixels with grayscale value ≥ 70 .
- Blob: connected component of 20-30 pixels.



black = 0
white = 255

Particle tracking. Track moving particles over time.

11.7 Gerichtete Graphen

11.8 Topologische Sortierung

11.9 Starke Zusammenhangskomponenten

11.10 Minimale Spannbäume

11.11 Kürzeste Wege

Übersicht – Graph Terminology

Undirected Graphs

- ▶ $V(G)$ set of vertices, $E(G)$ set of edges
- ▶ write uv (or vu) for edge $\{u, v\}$
- ▶ edges *incident* at vertex v : $E(v)$
- ▶ u and v are *adjacent* iff $\{u, v\} \in E$,
- ▶ *neighborhood* $N(v) = \{w \in V : w \text{ adjacent to } v\}$
- ▶ *degree* $d(v) = |E(v)|$

- ▶ *walk* („Weg“) $w[0..n]$ of length n : sequence of vertices with $\forall i \in [0..n) : w[i]w[i+1] \in E$
- ▶ *path* („Pfad“) p is a (vertex-) simple walk: no duplicate vertices except possibly its endpoints
- ▶ *edge-simple* walk: no edge used twice
- ▶ *cycle* c is a closed path, i. e., $c[0] = c[n]$

- ▶ G is *connected*
iff for all $u \neq v \in V$ there is a path from u to v
- ▶ G is *acyclic* iff \nexists cycle (of length $n \geq 1$) in G

Directed Graphs (where different)

- ▶ uv for (u, v)

- ▶ iff $(u, v) \in E \vee (v, u) \in E$
- ▶ in-/out-neighbors $N_{\text{in}}(v), N_{\text{out}}(v)$
- ▶ in-/out-degree $d_{\text{in}}(v), d_{\text{out}}(v)$

- ▶ *strongly connected* for digraphs
(*weakly connected* = connected ignoring directions)