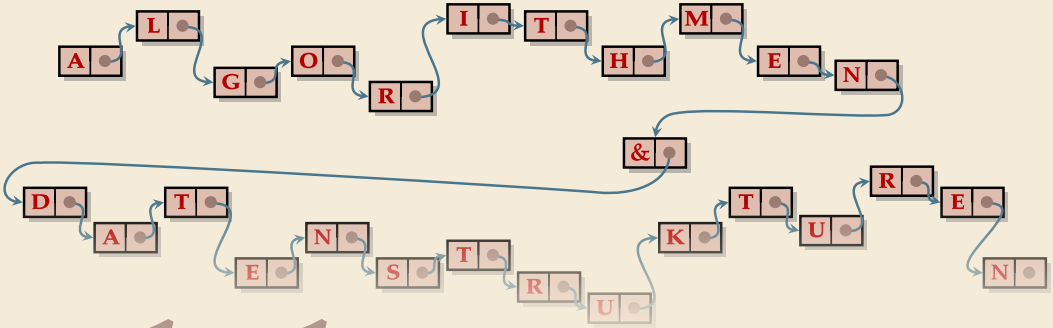


11 Graphen

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

CS210 (Sommersemester 2026)
Philipps-Universität Marburg
version `2026-07-01 08:34`

Outline

11 Graphen

11.1 Wo(zu er)finden wir Graphen?

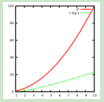
Clicker Question

Geben Sie alle zusammenpassenden Paare an:

(A) Graph

(B) Graf

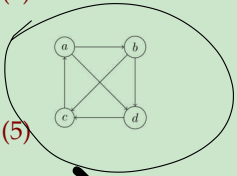
(C) Grave

(1) 

(2) 

(3) 

(4) 

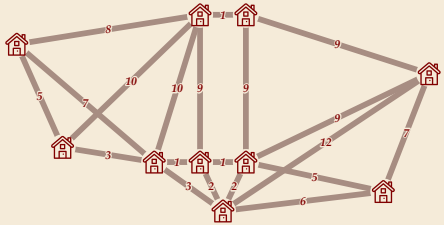
(5) 

(6) à

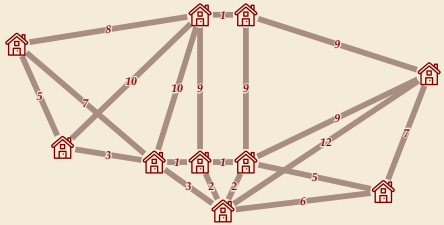
→ *sli.do/cs210*

Graphen

- Ein ***Graph*** ist eine Abstraktion für Daten, bei denen es (primär) um ***Entitäten*** und ihre paarweisen ***Beziehungen*** geht

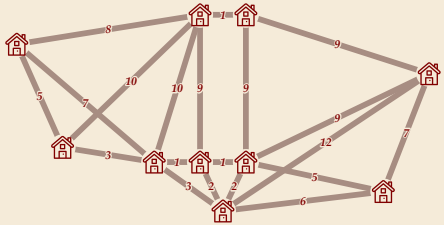


Graphen



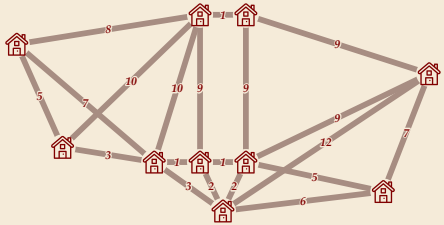
- Ein ***Graph*** ist eine Abstraktion für Daten, bei denen es (primär) um ***Entitäten*** und ihre paarweisen ***Beziehungen*** geht
- finden sich in vielfältiger Form
- In Anwendungen auch **Netzwerke** genannt (für uns synonym zu „Graph“)

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/Netwerke
  - **soziale Netzwerke:** z. B. Personen und Freundschaftsbeziehungen, . . .
  - **physische Netzwerke:** Straßen, Stromleitungen, das Internet (Computernetzwerk), . . .
  - **Informationsnetzwerke:** das World Wide Web, Ontologien, Protein-Interaktionen, . . .

Graphen

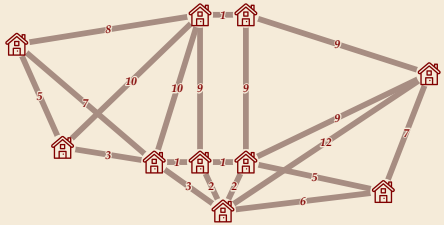


- Ein ***Graph*** ist eine Abstraktion für Daten, bei denen es (primär) um ***Entitäten*** und ihre paarweisen ***Beziehungen*** geht
- finden sich in vielfältiger Form
- In Anwendungen auch **Netzwerke** genannt (für uns synonym zu „Graph“)
- **Beispiele** für Graphen/Netwerke
  - **soziale Netzwerke:** 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?)

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/Netwerke
  - **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?

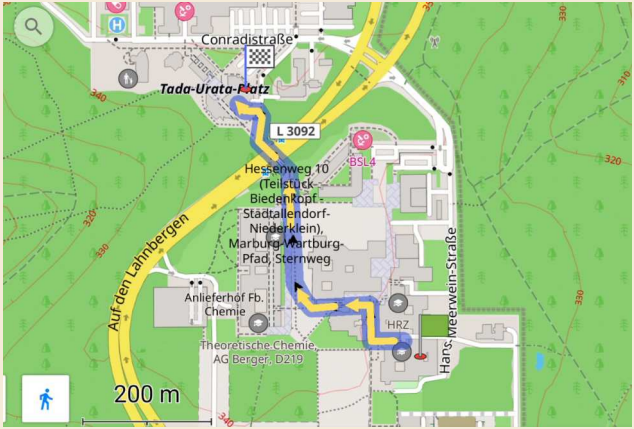
(Ergibt es Sinn, die alle über einen Kamm zu scheren?)

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. Fahrtzeit



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 . . .

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

Anwendung 2: Page Rank

1990s Suchmaschinen: erstes Suchergebnis ≈ 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 ⇝ viele Links **zu** dieser Seite

⇝ *Verwende Links zwischen Seiten selbst als Hauptkriterium für Relevanz!*

Anwendung 2: Page Rank

1990s Suchmaschinen: erstes Suchergebnis ≈ 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 ⇝ viele Links **zu** dieser Seite

⇝ *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

Anwendung 2: Page Rank

1990s Suchmaschinen: erstes Suchergebnis ≈ 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 ⇝ viele Links **zu** dieser Seite

⇝ *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
  - ⇝ cleverer Algorithmus berechnet das als *steady state* einer *Markovkette*
  - ⇝ clevere Implementierung schafft das verteilt im Cluster für Millarden von Seiten

Anwendung 2: Page Rank

1990s Suchmaschinen: erstes Suchergebnis ≈ 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 ⇝ viele Links **zu** dieser Seite

⇝ *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
  - ⇝ cleverer Algorithmus berechnet das als *steady state* einer *Markovkette*
  - ⇝ clevere Implementierung schafft das verteilt im Cluster für Millarden von Seiten

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

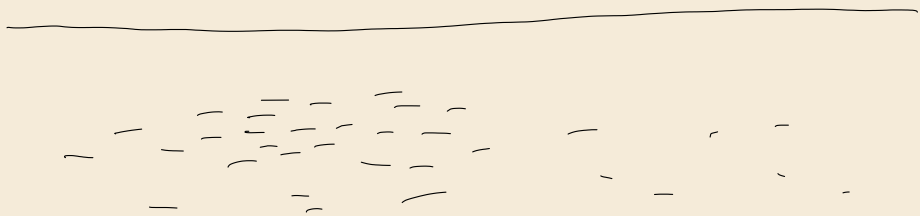
- Entität = Website; Beziehung = Hyperlinks
- Beziehungen haben feste Richtung (asymmetrisch), aber keine Gewichtung

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

- Das **Genom** (eines Individuums) zu **sequenzieren** ist mittlerweile kommerziell möglich

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



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 **rekonstruierten** (*genome assembly*)

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

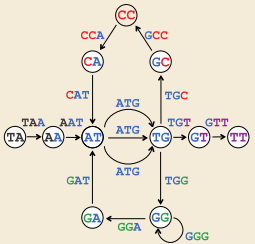
- Das **Genom** (eines Individuums) zu **sequenzieren** ist mittlerweile kommerziell möglich
- Labor-Techniken liefern aber nur kurze „Schnipsel“ des Genoms
  - 100 – 5000 Basenpaare (Zeichen) pro Read (je nach Technologie)
  - Menschliches Genom hat etwa 3 000 000 000 Basenpaare! (für einen Chromosomensatz!)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

⇝ benötigen Algorithmen, die daraus das Genom **rekonstruierten** (*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

DeBruijn$_3$(**TAATGCCATGGGATGTT**)



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig. 4.1

https://cogniterra.org/lesson/29910/step/2?unit=22007

11.2 Terminologie

Kernbegriffe

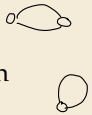
- **Knoten** (***vertex***) = Entitäten im Graph
  - Synonyme: Ecke, Punkt; *node*, *point*
  - Formelsprache: Knoten $v \in V$ (Menge aller Knoten eines Graphen)
- **Kante** (***edge***) = Beziehung zwischen zwei Knoten
  - Synonyme: Pfeil, Linie, Relation; *arc*
  - Formelsprache: Kante $e \in E$ (Menge 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	*gerichteter* Graph (*directed graph*, *Digraph*)	*ungerichteter* Graph
≤ 1 Kante zwischen u und v	*einfacher* Graph	*Multigraph* G. mit *parallelen* Kanten
Kanten von v zu v	mit *Schleifen* (*loops*)	(schleifenfrei)
Gewichtung auf Kanten	*(Kanten-) gewichteter* G.	*ungewichteter* Graph



🙃 jegliche Kombination kann Sinn ergeben . . .

Graphen in allen „Geschmacksrichtungen“

- Graphen modellieren sehr diverse Entitäten und Beziehungen

⇝ mehrere verschiedene (aber wiederkehrende!) Varianten

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

Ungerichtete Graphen – Formale Definition

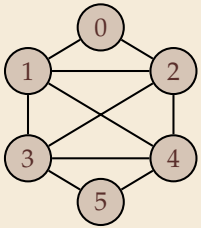
- Standard-Annahme: Graph ist ungewichtet, ungerichtet, schleifenfrei & einfach
- *Graph* $G = (V, E)$ mit
  - V endliche Menge von *Knoten*
  - $E \subseteq [V]^2$ eine Menge von *Kanten*, mit $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

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 . . .

Ungerichtete Graphen – Formale Definition

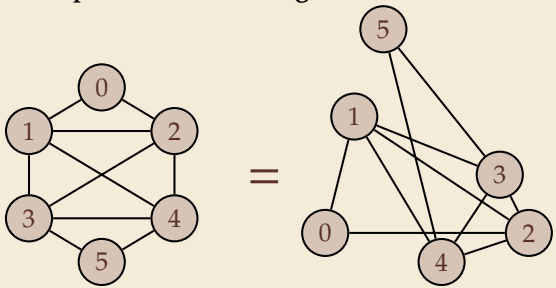
- Standard-Annahme: Graph ist ungewichtet, ungerichtet, schleifenfrei & einfach
- *Graph* $G = (V, E)$ mit
  - V endliche Menge von *Knoten*
  - $E \subseteq [V]^2$ eine Menge von *Kanten*, mit $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

2-elementige Teilmengen

Beispiel

$$
\begin{aligned}
V &= \{0,1,2,3,4,5\} \\
E &= \big\{\{0,1\},\{1,2\},\{1,4\},\{1,3\},\{0,2\}, \\
&\qquad \{2,4\},\{2,3\},\{3,4\},\{3,5\},\{4,5\}\big\}.
\end{aligned}
$$

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*, ohne Schleifen
    $V^2 = V \times V = \{(x, y) : x \in V \wedge y \in V\}$ 2-Tuple / (geordnete) Paare aus V

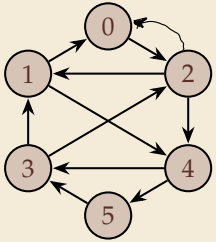
Gerichtete Graphen – Formale Definition

- 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

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

Graphische Darstellung



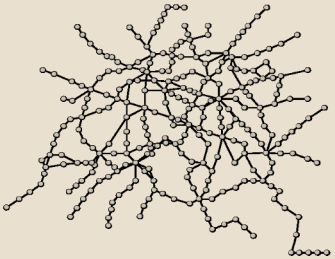
11.3 Ungerichtete Graphen

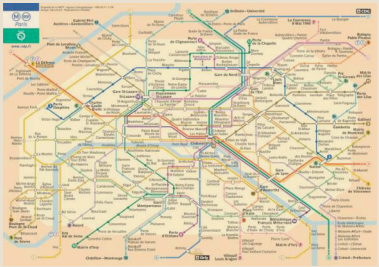
Undirected graphs

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.



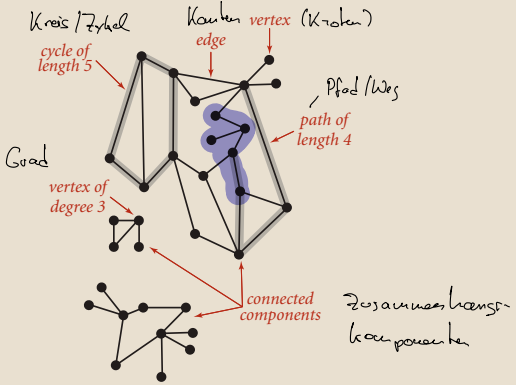


Graph terminology

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

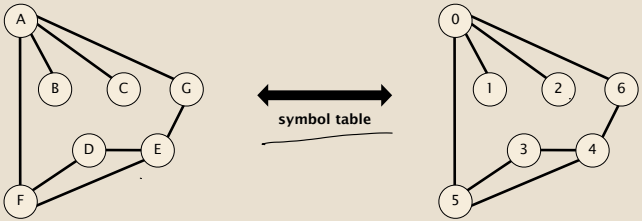
problem	description
s–t path	*Is there a path between s and t ?*
shortest s–t path	*What is the shortest path between s and t ?*
cycle	*Is there a cycle in the graph ?*
Euler cycle	*Is there a cycle that uses each edge exactly once ?*
Hamilton cycle	*Is there a cycle that uses each vertex exactly once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

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

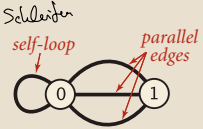
Graph representation

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*

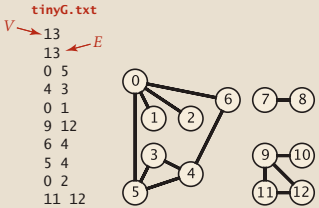
```
In in = new In(args[0]);
Graph G = new Graph(in);

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.




```
% 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

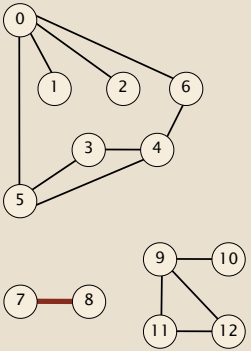
public class Graph		
	Graph(int V)	*create an empty graph with V vertices*
	Graph(In in)	*create a graph from input stream*
void	addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer>	adj(int v)	*vertices adjacent to v*
int	V()	*number of vertices*
int	E()	*number of edges*

```
// 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).



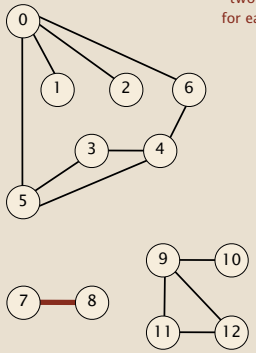
0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

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

Adjazenzmatrix

Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;
for each edge v–w in graph: `adj[v][w] = adj[w][v] = true`.



two entries for each edge

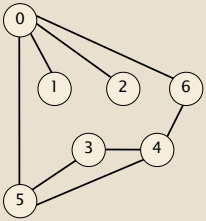
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0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

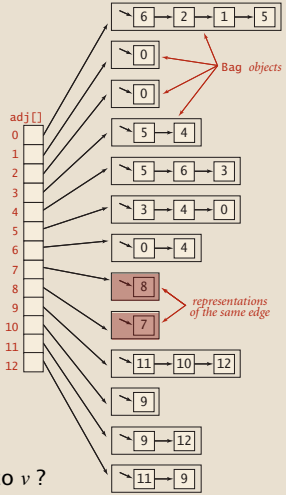
Q. How long to iterate over vertices adjacent to v ?

Adjazenzlisten

Adjacency-list graph representation

Maintain vertex-indexed array of lists.





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

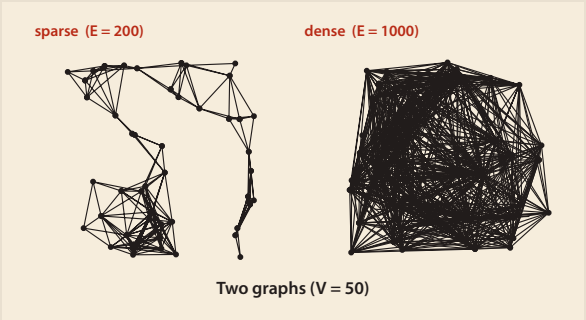
Graph representations

In practice. Use adjacency-lists representation.

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

dicht oder dünn besetzt

huge number of vertices,
small average vertex degree



Two graphs (V = 50)

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

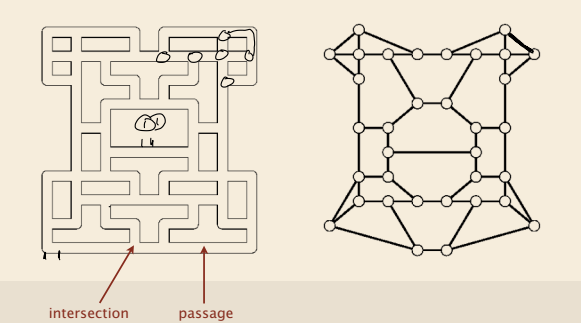
11.5 Tiefensuche

Depth-First Search

Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.

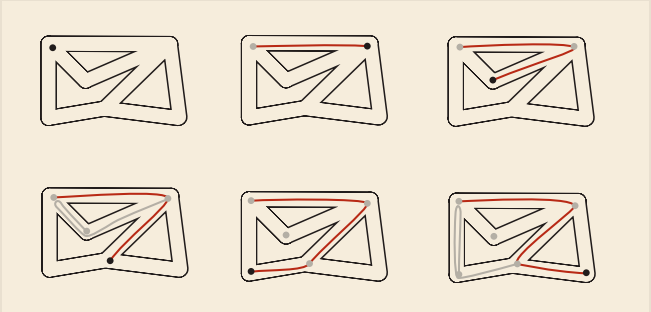


Goal. Explore every intersection in the maze.

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.



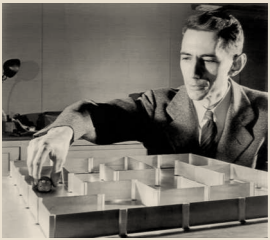
Trémaux maze exploration

Algorithm.

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

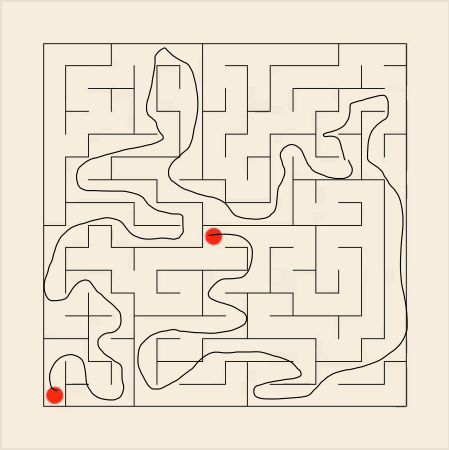
First use? Theseus entered Labyrinth to kill the monstrous Minotaur;
Ariadne instructed Theseus to use a ball of string to find his way back out.

The Labyrinth (with Minotaur)

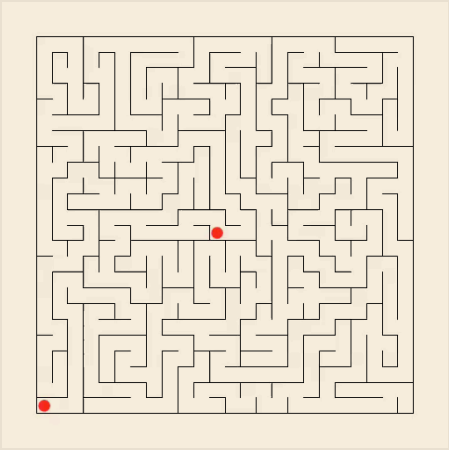


Claude Shannon (with Theseus mouse)

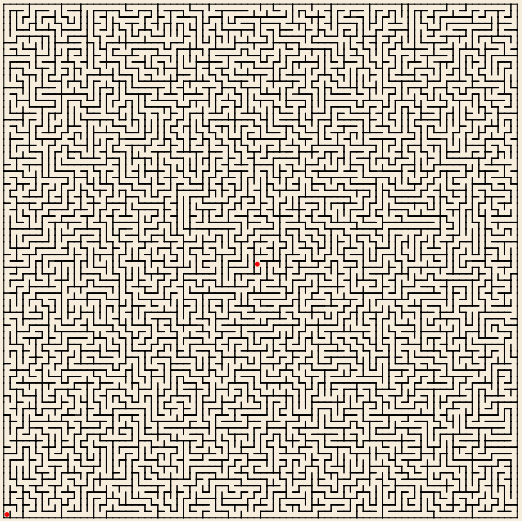
Maze exploration: easy



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;

   public DepthFirstPaths(Graph G, int s)
   {
      ...
      dfs(G, 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;
         }
   }
}
```

marked[v] = true if v connected to s

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

initialize data structures

find vertices connected to s

recursive DFS does the work

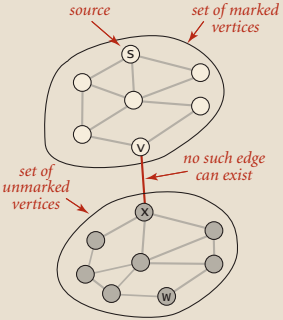
Depth-first search: properties

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

Dfs linear in n+m O(n+m)

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]

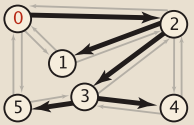
Each vertex connected to s is visited once.

Depth-first search: properties

Proposition. After DFS, can check if vertex v is connected to s in constant time and can find v–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;
}
```



edgeTo[]

0	
1	2
2	0
3	2
4	3
5	3

5 - 3 - 2 - 0

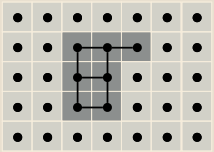
Depth-first search application: flood fill

Challenge. Flood fill (Photoshop magic wand).

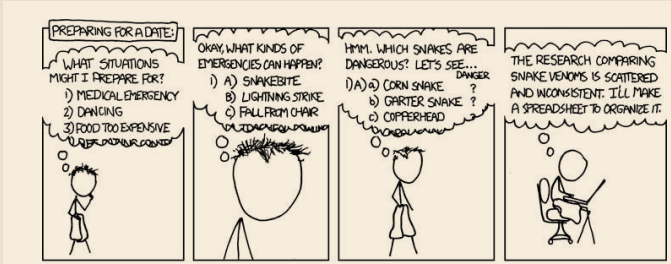
Assumptions. Picture has millions to billions of pixels.

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



11.6 Breitensuche

Breadth-First Search

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.

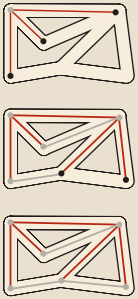
BFS (from source vertex s)

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

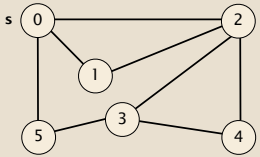
Breadth-first search properties

Q. In which order does BFS examine vertices?

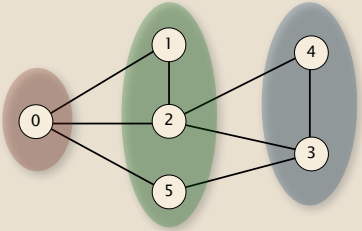
A. Increasing distance (number of edges) from s.

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 $E + V$.



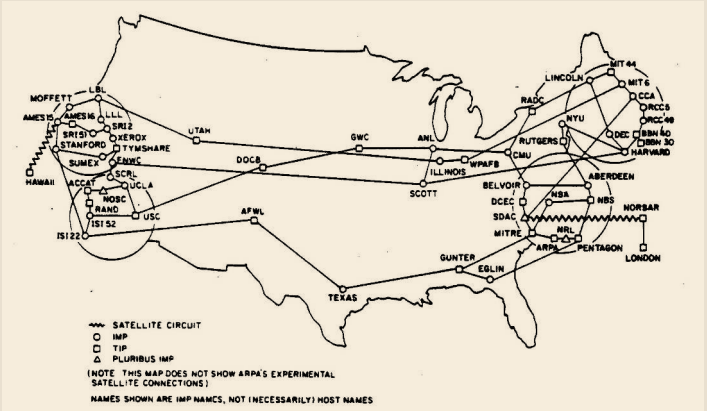
graph G



dist = 0 dist = 1 dist = 2

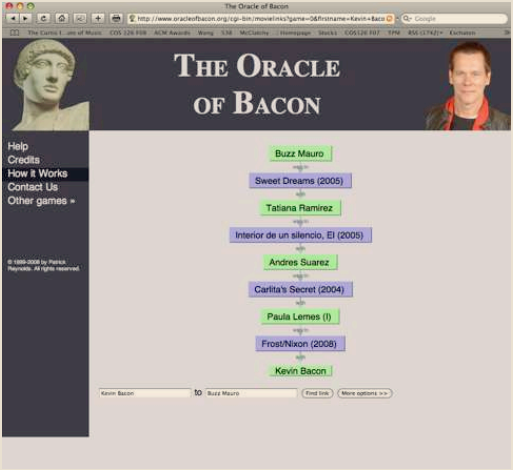
Breadth-first search application: routing

Fewest number of hops in a communication network.



ARPANET, July 1977

Breadth-first search application: Kevin Bacon numbers



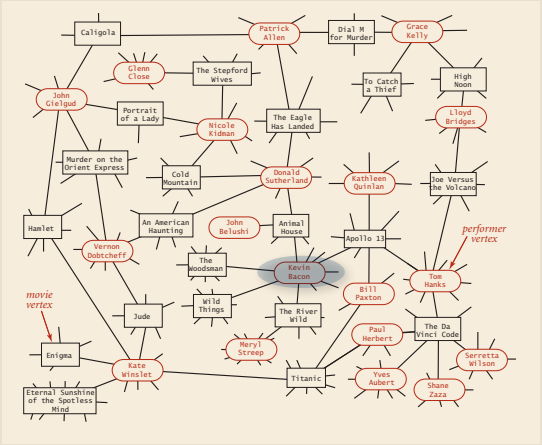
http://oracleofbacon.org

Endless Games board game

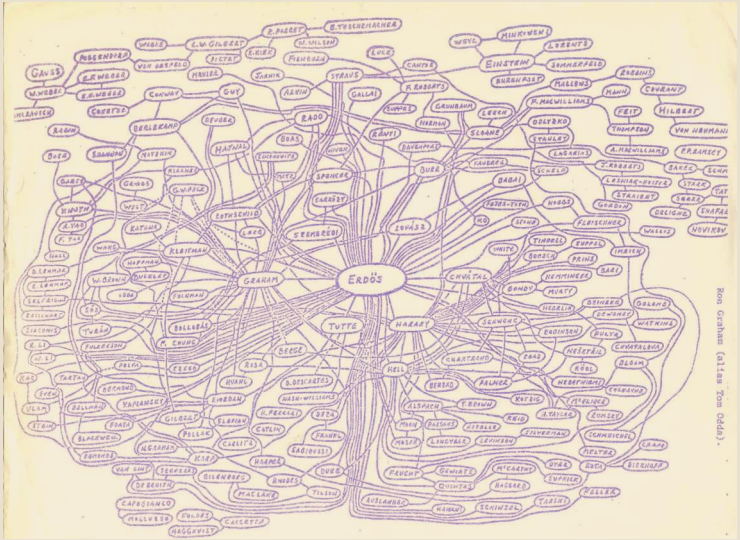
SixDegrees iPhone App

Kevin Bacon graph

- Include one vertex for each performer and one for each movie.
- Connect a movie to all performers that appear in that movie.
- Compute shortest path from s = Kevin Bacon.



Breadth-first search application: Erdös numbers



hand-drawing of part of the Erdös graph by Ron Graham

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.

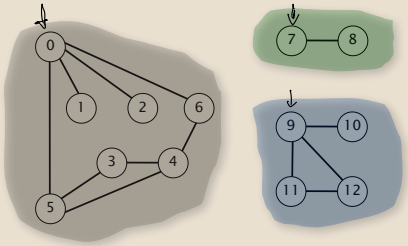
Depth-first search. Yes. [next few slides]

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.



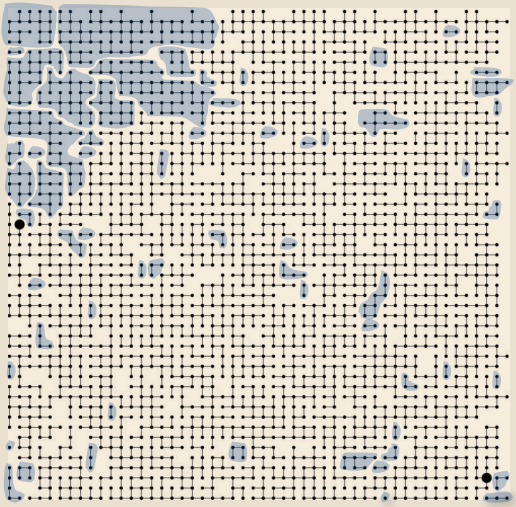
3 connected components

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

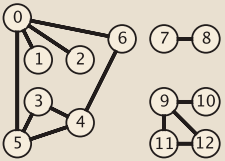
Connected components

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

```
13
13
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;  }

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

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


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);
}
```

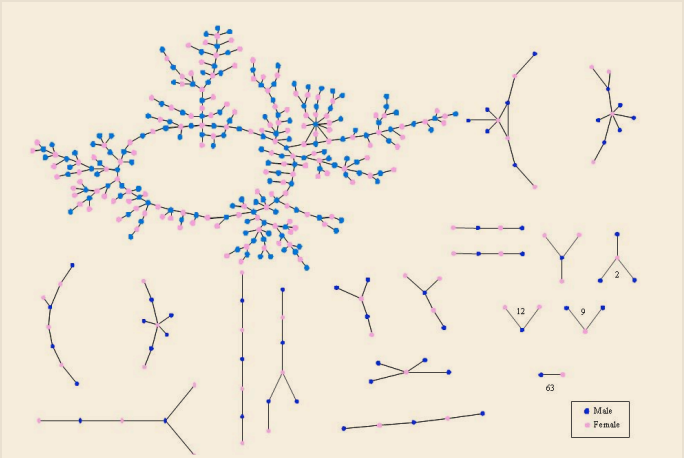
number of components

id of component containing v

v and w connected iff same id

all vertices discovered in same call of dfs have same id

Connected components application: study spread of STDs



Relationship graph at "Jefferson High"

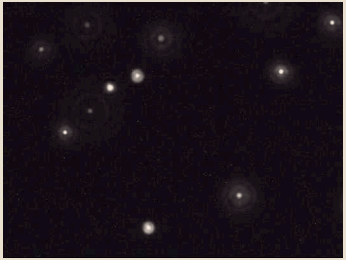
Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. American Journal of Sociology, 110(1): 44-99, 2004.

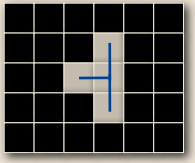
Connected components application: particle detection

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



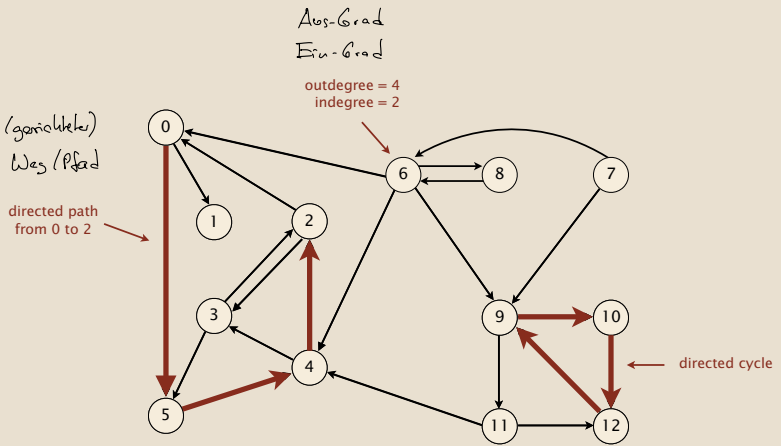


Particle tracking. Track moving particles over time.

11.7 Gerichtete Graphen

Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.



Digraph applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

problem	description
s→t path	*Is there a path from s to t ?*
shortest s→t path	*What is the shortest path from s to t ?*
directed cycle	*Is there a directed cycle in the graph ?*
topological sort	*Can the digraph be drawn so that all edges point upwards?*
strong connectivity	*Is there a directed path between all pairs of vertices ?*
transitive closure	*For which vertices v and w is there a directed path from v to w ?*
PageRank	*What is the importance of a web page ?*

topologisches Sortieren

starke Zusammenhangskomponenten

Digraph API

Almost identical to `Graph` API.

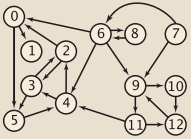
public class Digraph		
	`Digraph(int V)`	*create an empty digraph with V vertices*
	`Digraph(In in)`	*create a digraph from input stream*
`void`	`addEdge(int v, int w)`	*add a directed edge v→w*
`Iterable<Integer>`	`adj(int v)`	*vertices pointing from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`Digraph`	`reverse()`	*reverse of this digraph*
`String`	`toString()`	*string representation*

Digraph API

tinyDG.txt

```
V
13
22
 4  2
 2  3
 3  2
 6  0
 0  1
 2  0
11 12
12  9
 9 10
 9 11
 7  9
10 12
11  4
 4  3
 3  5
 6  8
 8  6
 ⋮
```

E




```
% java Digraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
⋮
11->4
11->12
12->9
```

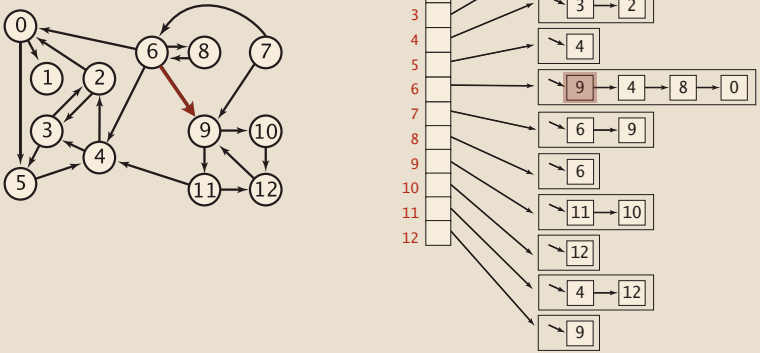
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

read digraph from input stream

print out each edge (once)

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.



Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v.
- Real-world digraphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices pointing from v?
list of edges	E	1	E	E
adjacency matrix	V^2	1†	1	V
adjacency lists	$E + V$	1	$outdegree(v)$	$outdegree(v)$

† disallows parallel edges

Adjacency-lists graph representation (review): Java implementation

```
public class Graph
{
   private final int V;
   private final 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

create empty graph with V vertices

add edge v–w

iterator for vertices adjacent to v

Adjacency-lists digraph representation: Java implementation

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

   public Digraph(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);

   }

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

adjacency lists

create empty digraph with V vertices

add edge v→w

iterator for vertices pointing from v

Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a digraph algorithm.

DFS (to visit a vertex v)

Mark v as visited.
Recursively visit all unmarked vertices w pointing from v.

Depth-first search (in directed graphs)

Code for directed graphs identical to undirected one.

[substitute `Digraph` for `Graph`]

```
public class DirectedDFS
{
   private boolean[] marked;

   public DirectedDFS(Digraph G, int s)
   {
      marked = new boolean[G.V()];
      dfs(G, s);
   }

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

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

true if path from s

constructor marks vertices reachable from s

recursive DFS does the work

client can ask whether any vertex is reachable from s

Reachability application: program control-flow analysis

Every program is a digraph.

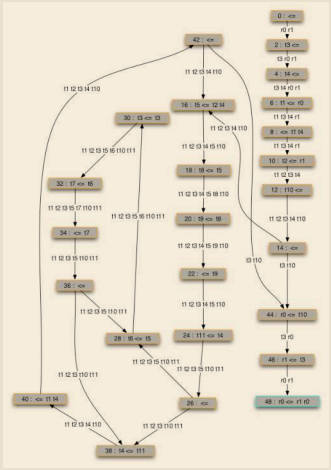
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



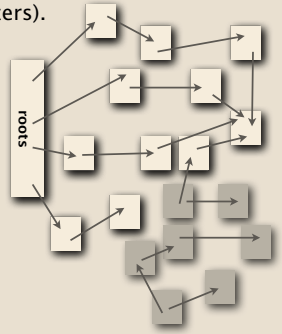
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).

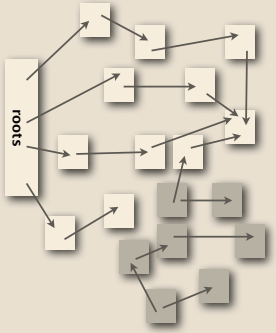


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



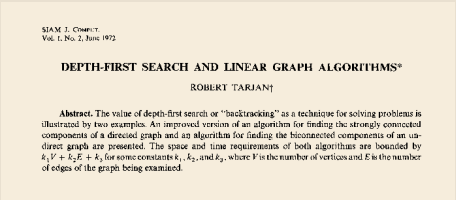
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.

- ✓ Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.



SIAM J. COMPUT.
Vol. 1, No. 2, June 1972

DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*

ROBERT TARJAN†

Abstract. The value of depth-first search or "backtracking" as a technique for solving problems is illustrated by two examples. An improved version of an algorithm for finding the strongly connected components of a directed graph and an algorithm for finding the biconnected components of an undirect graph are presented. The space and time requirements of both algorithms are bounded by $k_1V + k_2E + k_3$ for some constants k_1, k_2, and k_3, where V is the number of vertices and E is the number of edges of the graph being examined.

Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a digraph algorithm.

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**
- **for each unmarked vertex pointing from v:**
  add to queue and mark as visited.

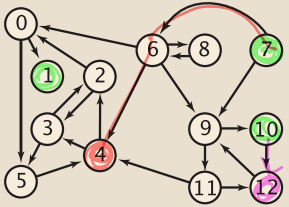
Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a set of source vertices, find shortest path from any vertex in the set to each other vertex.

Ex. $S = \{ 1, 7, 10 \}$.

- Shortest path to 4 is 7→6→4.
- Shortest path to 5 is 7→6→0→5.
- Shortest path to 12 is 10→12.
- ...



Q. How to implement multi-source shortest paths algorithm?

A. Use BFS, but initialize by enqueuing all source vertices.

11.8 Topologische Sortierung

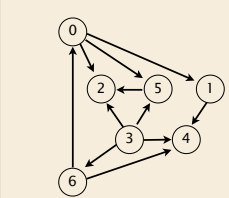
Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

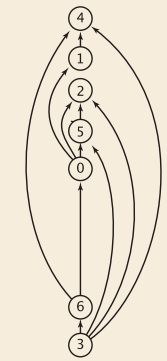
Digraph model. vertex = task; edge = precedence constraint.

0. Algorithms
1. Complexity Theory
2. Artificial Intelligence
3. Intro to CS
4. Cryptography
5. Scientific Computing
6. Advanced Programming

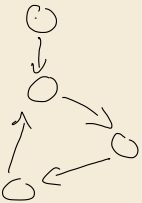
tasks



precedence constraint graph



feasible schedule



gerichteter Kreis

immer problematisch

DAG

directed acyclic graph

gerichteter azyklischer Graph

Zeit

Depth-first search order

```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   public DepthFirstOrder(Digraph G)
   {
      reversePostorder = new Stack<Integer>();
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         if (!marked[v]) dfs(G, v);
   }

   private void dfs(Digraph G, int v)
   {
      marked[v] = true;
      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w);
      reversePostorder.push(v);
   }

   public Iterable<Integer> reversePostorder()
   {  return reversePostorder;  }
}
```

Berechnet Reihenfolge der Knoten

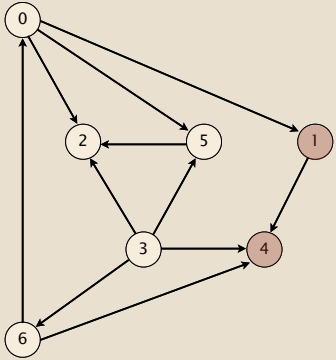
Thm: Entweder ist das topologische Sortieren oder es gibt Kreis.

returns all vertices in "reverse DFS postorder"

Topological sort in a DAG: intuition

Why does topological sort algorithm work?

- First vertex in postorder has outdegree 0.
- Second-to-last vertex in postorder can only point to last vertex.
- ...



postorder

4 1 2 5 0 6 3

topological order

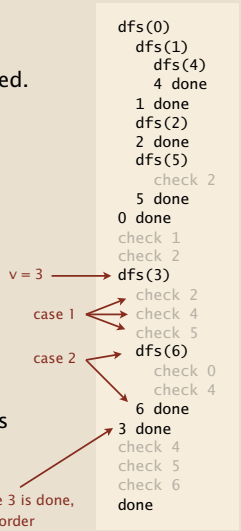
3 6 0 5 2 1 4

Topological sort in a DAG: correctness proof

Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge v→w. When `dfs(v)` is called:

- Case 1: `dfs(w)` has already been called and returned.
  Thus, `w` was done before `v`.
- Case 2: `dfs(w)` has not yet been called.
  `dfs(w)` will get called directly or indirectly
  by `dfs(v)` and will finish before `dfs(v)`.
  Thus, `w` will be done before `v`.
- Case 3: `dfs(w)` has already been called,
  but has not yet returned.
  Can't happen in a DAG: function call stack contains
  path from `w` to `v`, so v→w would complete a cycle.

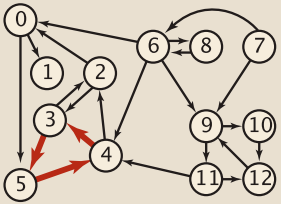


Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.



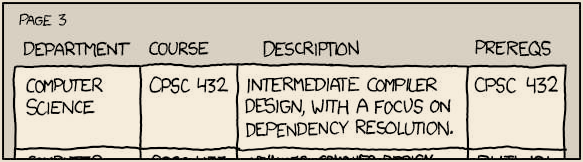
a digraph with a directed cycle

Goal. Given a digraph, find a directed cycle.

Solution. DFS. What else? See textbook.

Directed cycle detection application: precedence scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?



http://xkcd.com/754

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

```
public class A extends B
{
   ...
}
```

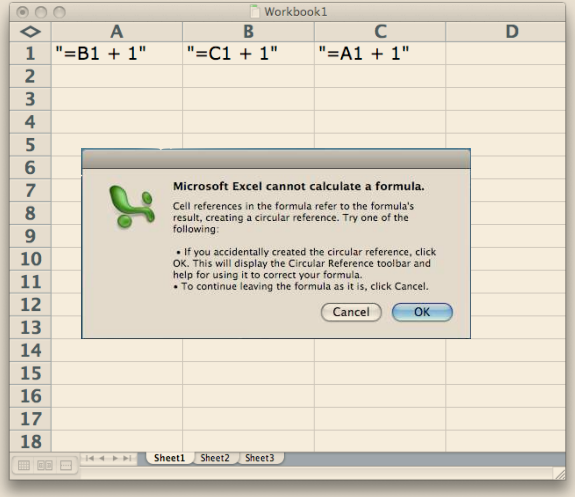
```
public class B extends C
{
   ...
}
```

```
public class C extends A
{
   ...
}
```

```
% javac A.java
A.java:1: cyclic inheritance
involving A
public class A extends B { }
       ^
1 error
```

Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)



Depth-first search orders

Observation. DFS visits each vertex exactly once. The order in which it does so can be important.

Orderings.

- Preorder: order in which `dfs()` is called.
- Postorder: order in which `dfs()` returns.
- Reverse postorder: reverse order in which `dfs()` returns.

```
private void dfs(Graph G, int v)
{
   marked[v] = true;
   preorder.enqueue(v);
   for (int w : G.adj(v))
      if (!marked[w]) dfs(G, w);
   postorder.enqueue(v);
   reversePostorder.push(v);
}
```