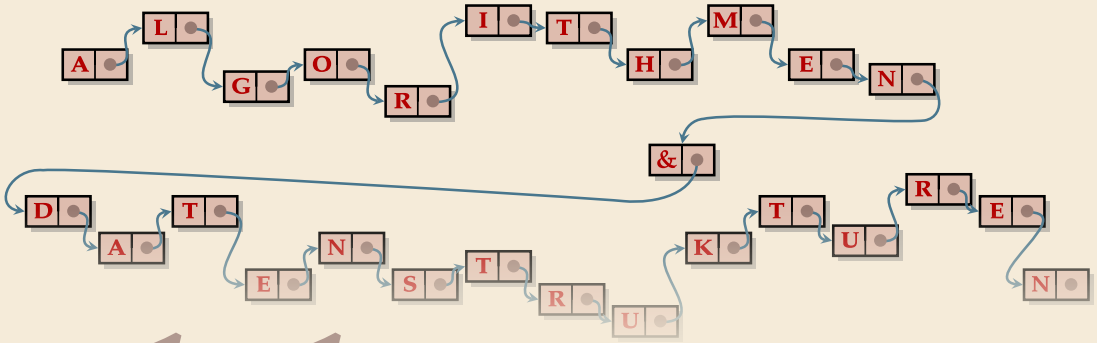


11 Graphen

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

CS210 (Sommersemester 2026)
Philipps-Universität Marburg
version `2026-07-02 01:08`

Outline

11 Graphen

11.1 Wo(zu er)finden wir Graphen?

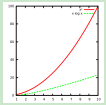
Clicker Question

Geben Sie alle zusammenpassenden Paare an:

(A) Graph

(B) Graf

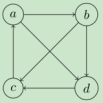
(C) Grave

(1) 

(2) 

(3) 

(4) 

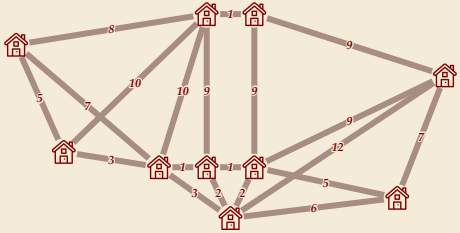
(5) 

(6) à

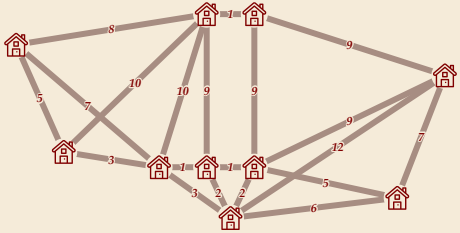
→ sli.do/cs210

Graphen

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

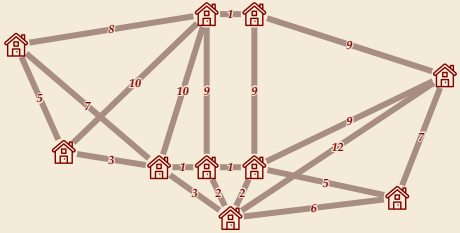


Graphen



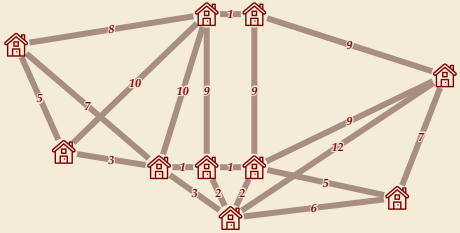
- Ein ***Graph*** ist eine Abstraktion für Daten, bei denen es (primär) um ***Entitäten*** und ihre paarweisen ***Beziehungen*** geht
- finden sich in vielfältiger Form
- In Anwendungen auch **Netzwerke** genannt (für uns synonym zu „Graph“)

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

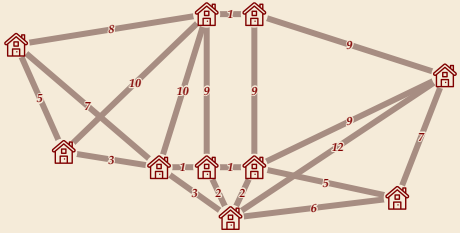


- Ein ***Graph*** ist eine Abstraktion für Daten, bei denen es (primär) um ***Entitäten*** und ihre paarweisen ***Beziehungen*** geht
- finden sich in vielfältiger Form
- In Anwendungen auch **Netzwerke** genannt (für uns synonym zu „Graph“)
- **Beispiele** für Graphen/Netwerke
  - **soziale Netzwerke:**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

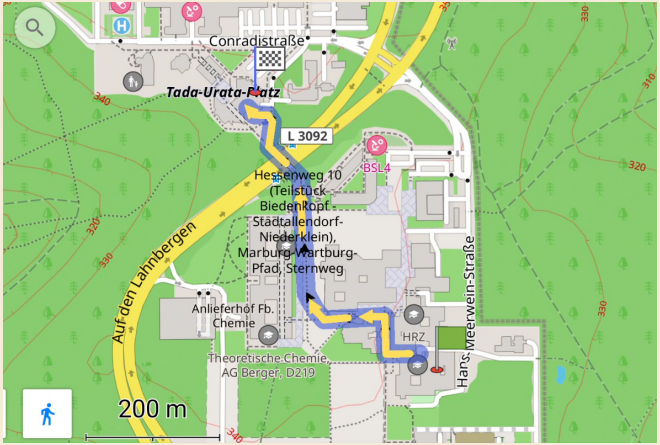
(Ergibt es Sinn, die alle über einen Kamm zu scheren?)

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 ≈ 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

$\rightsquigarrow$ benötigen Algorithmen, die daraus das Genom **rekonstruierten** (*genome assembly*)

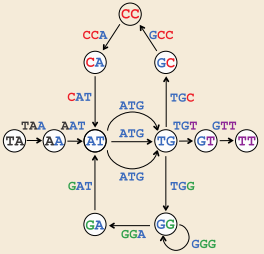
Anwendung 3: de-Bruijn-Graphs, Overlap-Graphs

- 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$_3$(**TA**ATGCCATGGGATGTT)



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 = Knote 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

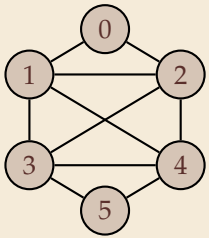
- Standard-Annahme: Graph ist ungewichtet, ungerichtet, schleifenfrei & einfach
- *Graph* $G = (V, E)$ mit
  - V endliche Menge von *Knoten*
  - $E \subseteq [V]^2$ eine Menge von *Kanten*, mit $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

2-elementige Teilmengen

Beispiel

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

Graphische Darstellung



so . . .

Ungerichtete Graphen – Formale Definition

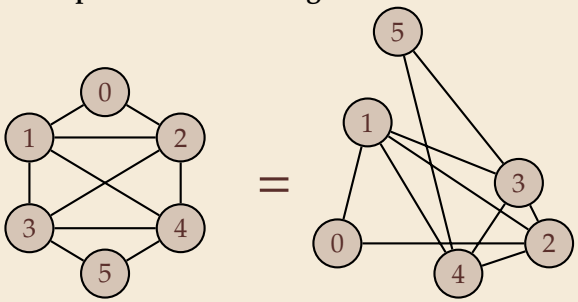
- Standard-Annahme: Graph ist ungewichtet, ungerichtet, schleifenfrei & einfach
- *Graph* $G = (V, E)$ mit
  - V endliche Menge von *Knoten*
  - $E \subseteq [V]^2$ eine Menge von *Kanten*, mit $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

2-elementige Teilmengen

Beispiel

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

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

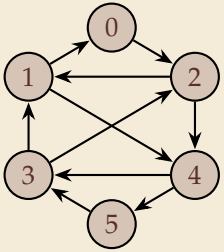
Gerichtete Graphen – Formale Definition

- 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

$$
\begin{aligned}
V &= \{0, 1, 2, 3, 4, 5\} \\
E &= \{(0, 2), (1, 0), (1, 4), (2, 1), (2, 4), \\
&\quad (3, 1), (3, 2), (4, 3), (4, 5), (5, 3)\}
\end{aligned}
$$

Graphische Darstellung



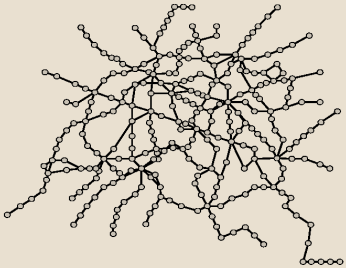
11.3 Ungerichtete Graphen

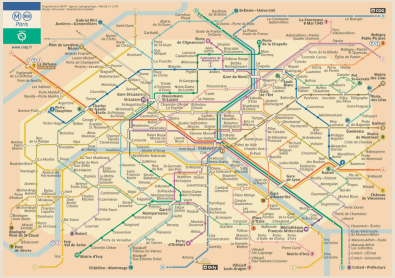
Undirected graphs

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.



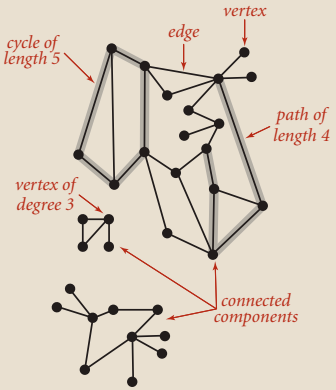


Graph terminology

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

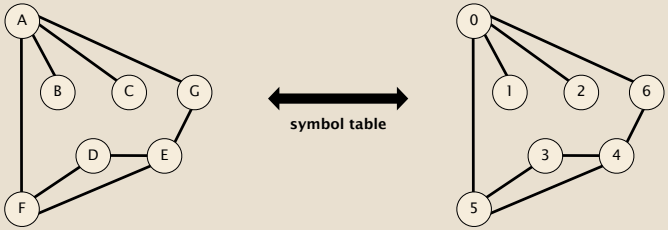
problem	description
s-t path	*Is there a path between s and t ?*
shortest s-t path	*What is the shortest path between s and t ?*
cycle	*Is there a cycle in the graph ?*
Euler cycle	*Is there a cycle that uses each edge exactly once ?*
Hamilton cycle	*Is there a cycle that uses each vertex exactly once ?*
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?

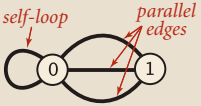
Graph representation

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*

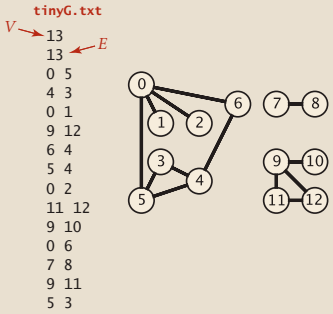
```
In in = new In(args[0]);
Graph G = new Graph(in);

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

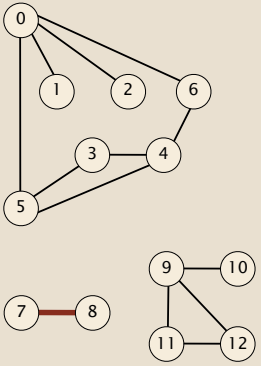
public class Graph		
	Graph(int V)	*create an empty graph with V vertices*
	Graph(In in)	*create a graph from input stream*
void	addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer>	adj(int v)	*vertices adjacent to v*
int	V()	*number of vertices*
int	E()	*number of edges*

```
// 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).

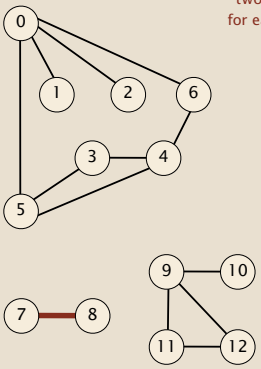


0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

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

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`.



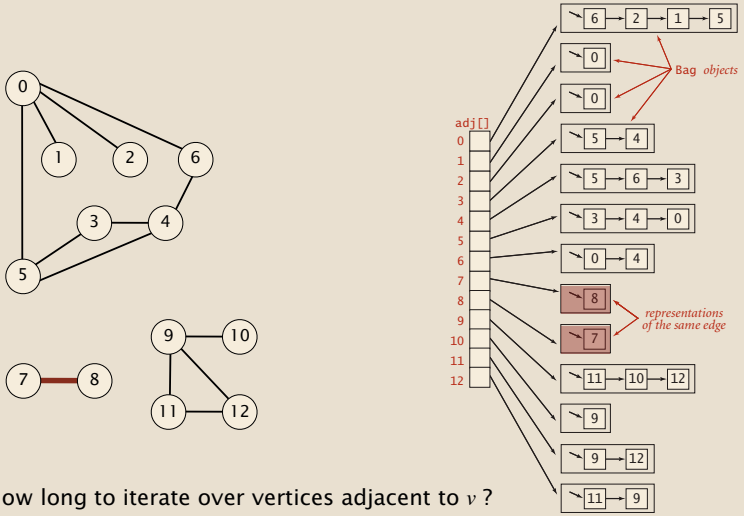
two entries
for each edge

	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 ?

Adjacency-list graph representation

Maintain vertex-indexed array of lists.



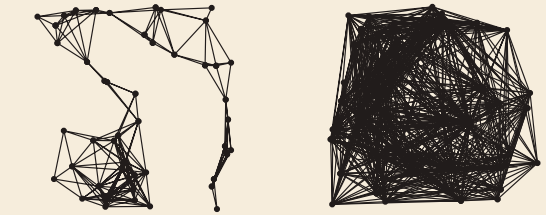
Q. How long to iterate over vertices adjacent to v ?

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



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)

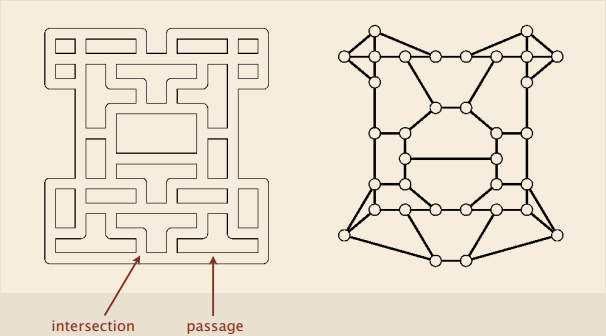
iterator for vertices adjacent to v

11.5 Tiefensuche

Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.

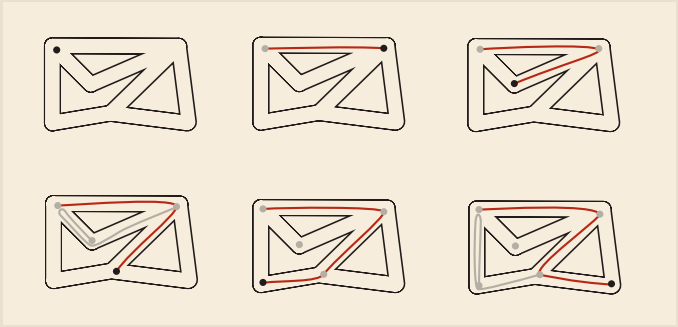


Goal. Explore every intersection in the maze.

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.



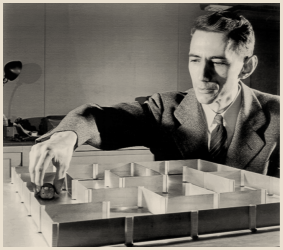
Trémaux maze exploration

Algorithm.

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

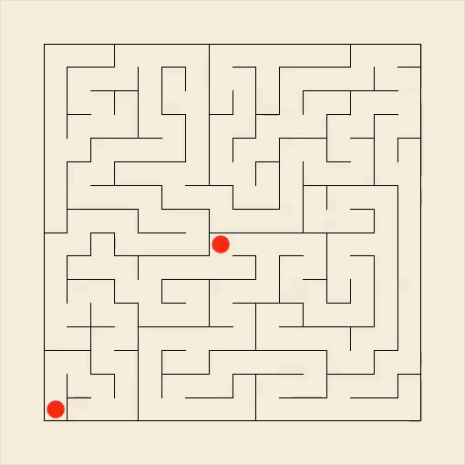
First use? Theseus entered Labyrinth to kill the monstrous Minotaur;
Ariadne instructed Theseus to use a ball of string to find his way back out.

The Labyrinth (with Minotaur)

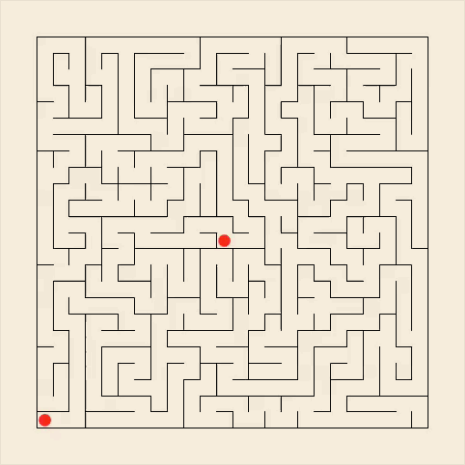


Claude Shannon (with Theseus mouse)

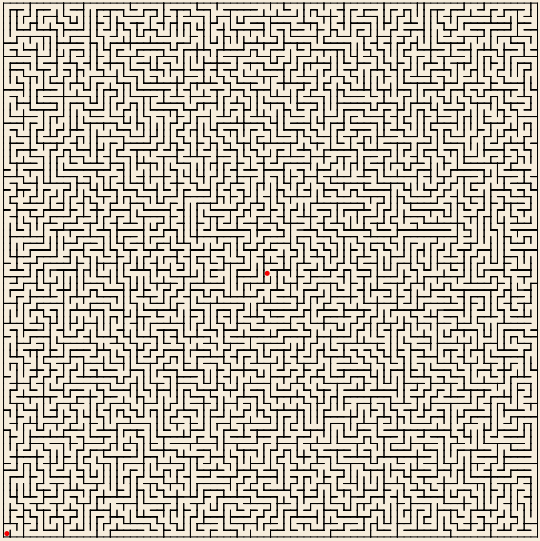
Maze exploration: easy



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

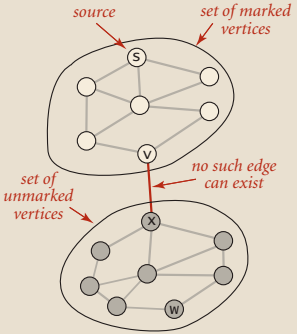
recursive DFS does the work

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).

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]

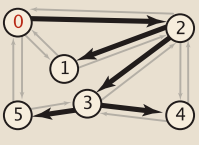
Each vertex connected to s is visited once.

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

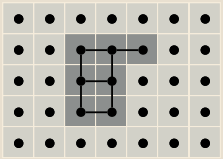
Depth-first search application: flood fill

Challenge. Flood fill (Photoshop magic wand).

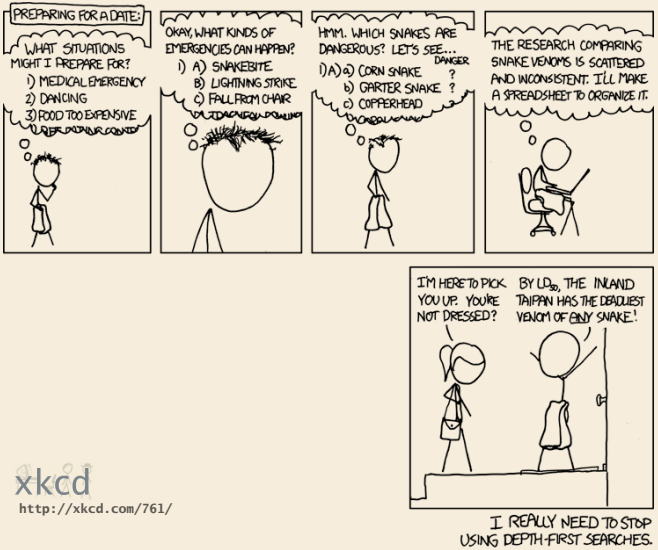
Assumptions. Picture has millions to billions of pixels.

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

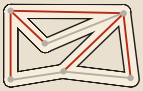
Breadth-first search

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

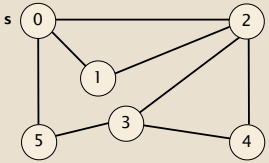
Breadth-first search properties

Q. In which order does BFS examine vertices?

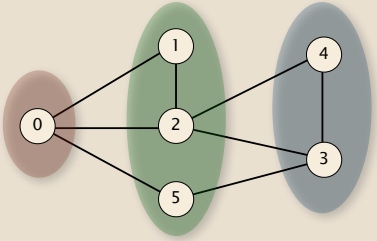
A. Increasing distance (number of edges) from s.

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 $E + V$.



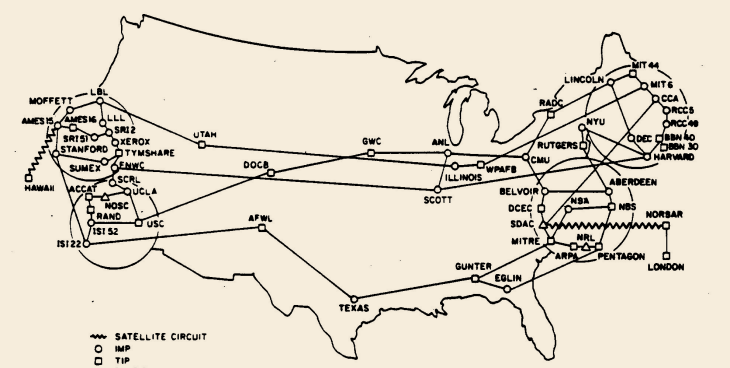
graph G



dist = 0 dist = 1 dist = 2

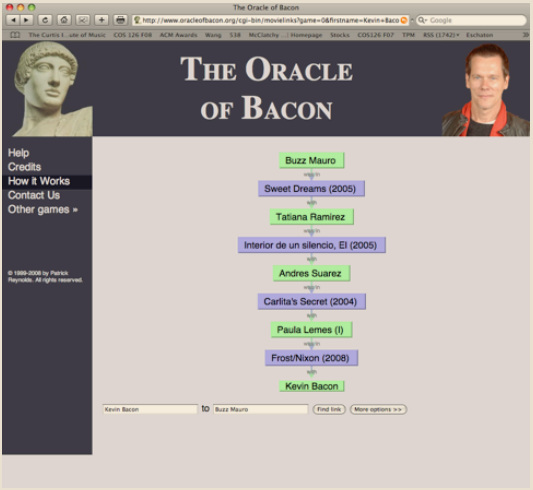
Breadth-first search application: routing

Fewest number of hops in a communication network.



ARPANET, July 1977

Breadth-first search application: Kevin Bacon numbers



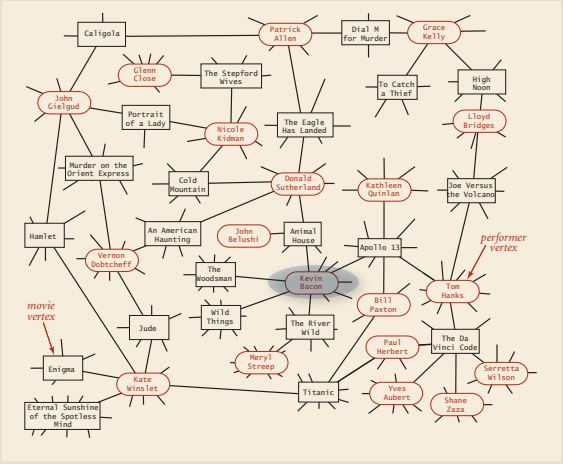
http://oracleofbacon.org

Endless Games board game

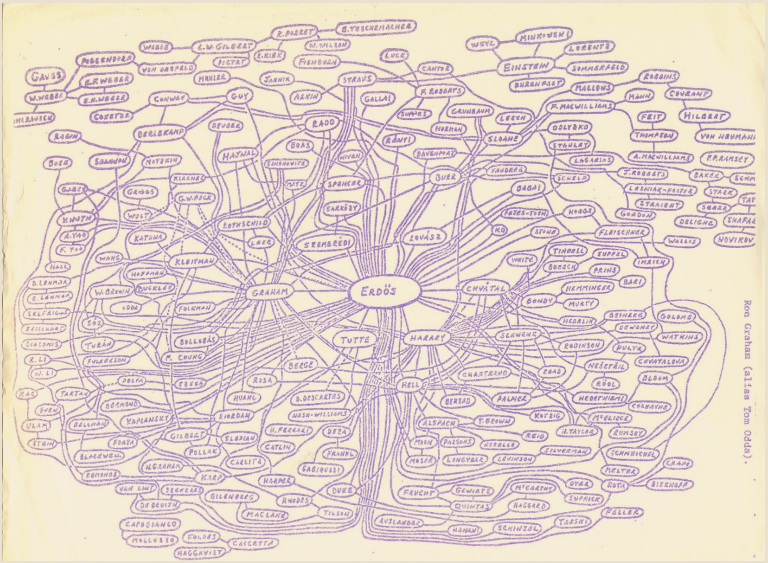
SixDegrees iPhone App

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.

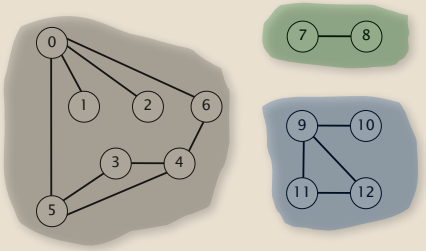
Depth-first search. Yes. [next few slides]

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.



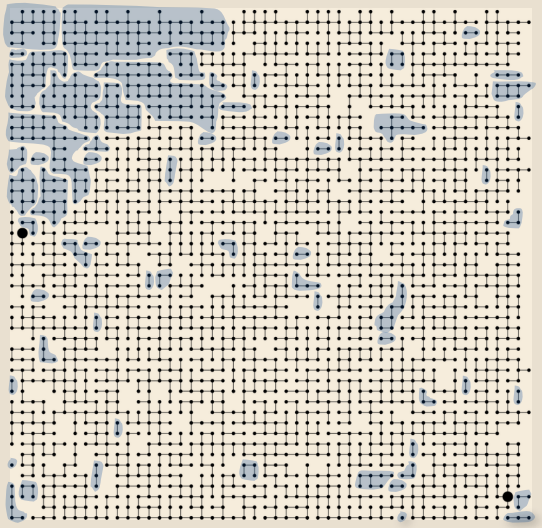
3 connected components

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

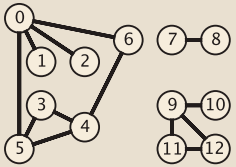
Connected components

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

V → 13
E → 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);
}
```

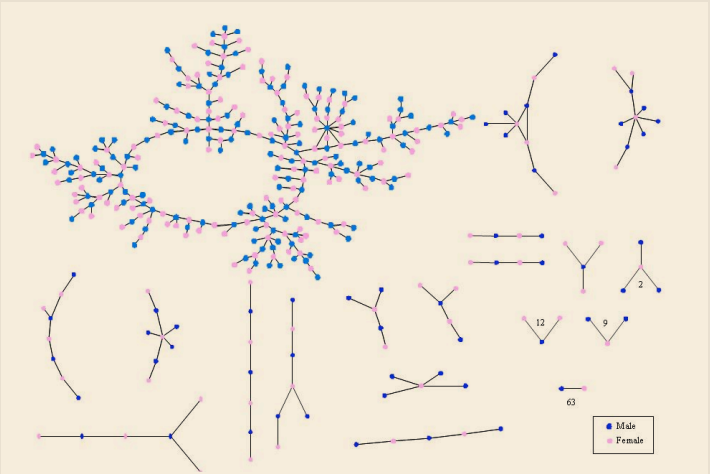
number of components

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"

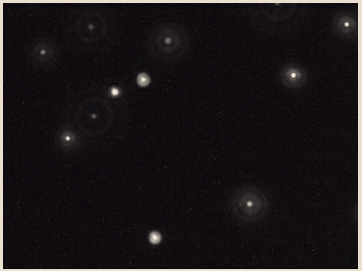
Peter Bearman, James Moody, and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. American Journal of Sociology, 110(1): 44–99, 2004.

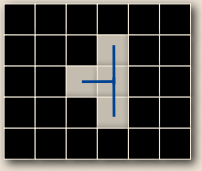
Connected components application: particle detection

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



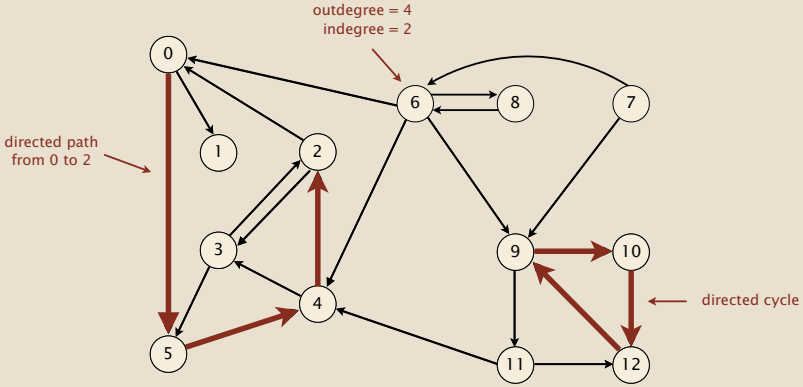


Particle tracking. Track moving particles over time.

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 ?*

Digraph API

Almost identical to `Graph` API.

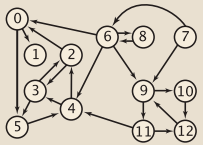
public class Digraph		
	`Digraph(int V)`	*create an empty digraph with V vertices*
	`Digraph(In in)`	*create a digraph from input stream*
`void`	`addEdge(int v, int w)`	*add a directed edge v→w*
`Iterable<Integer>`	`adj(int v)`	*vertices pointing from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number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

Digraph API

tinyDG.txt

V → 13
E → 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
 ⋮
```




```
% 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
```

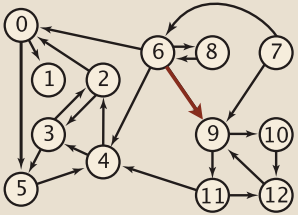
```
In in = new In(args[0]);
Digraph G = new Digraph(in);

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

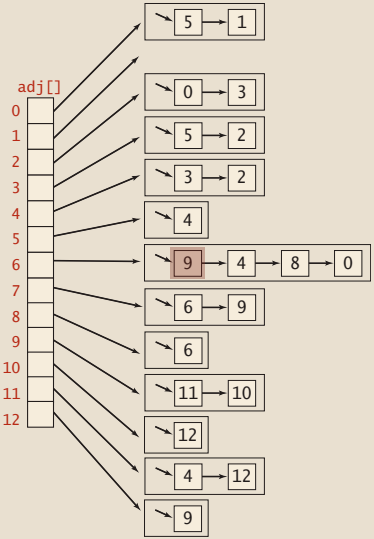
read digraph from input stream

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.

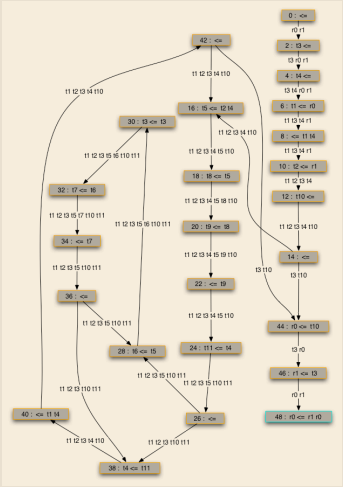
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



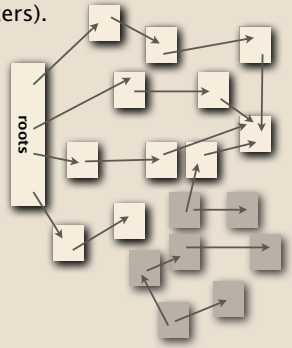
Reachability application: mark-sweep garbage collector

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).

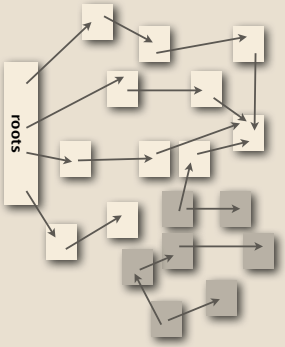


Reachability application: mark-sweep garbage collector

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).



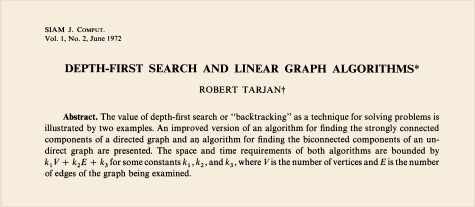
Depth-first search in digraphs summary

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.

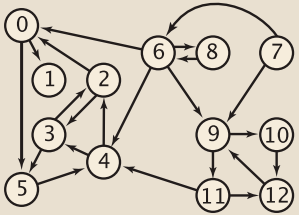
Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

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

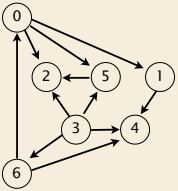
Precedence scheduling

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

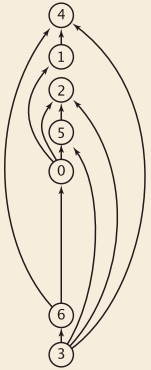
Digraph model. vertex = task; edge = precedence constraint.

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

Depth-first search order

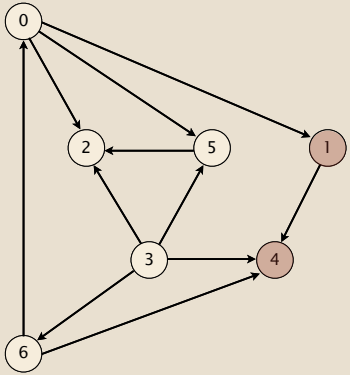
```
public class DepthFirstOrder
{
   private boolean[] marked;
   private Stack<Integer> reversePostorder;

   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;  }
}
```

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

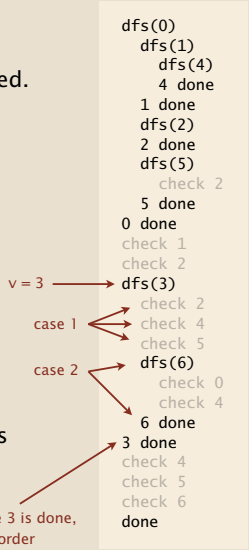
3 6 0 5 2 1 4

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.

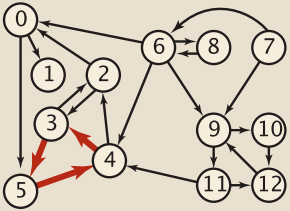


Directed cycle detection

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.



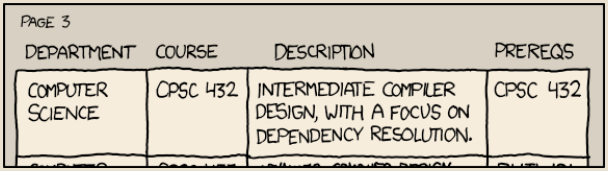
a digraph with a directed cycle

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
{
   ...
}
```

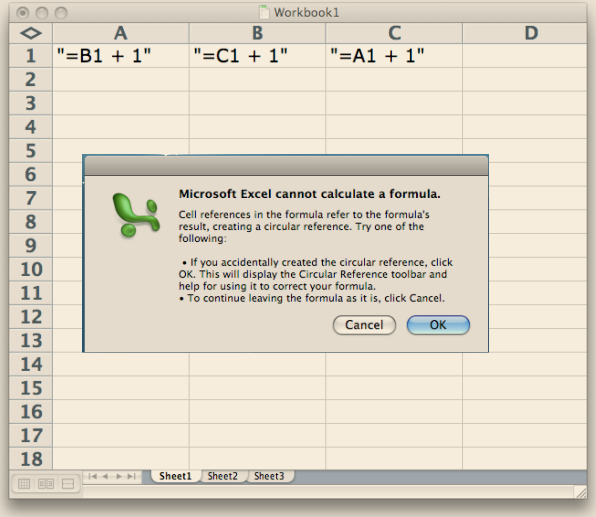
```
public class B extends C
{
   ...
}
```

```
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);
}
```

11.9 Starke Zusammenhangskomponenten

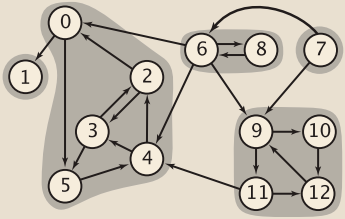
Strongly-connected components

Def. Vertices v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v.

Key property. Strong connectivity is an equivalence relation:

- v is strongly connected to v.
- If v is strongly connected to w, then w is strongly connected to v.
- If v is strongly connected to w and w to x, then v is strongly connected to x.

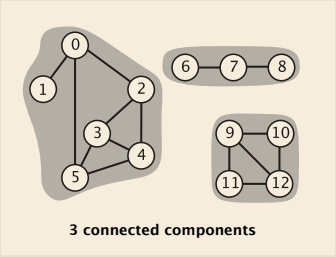
Def. A strong component is a maximal subset of strongly-connected vertices.



5 strongly-connected components

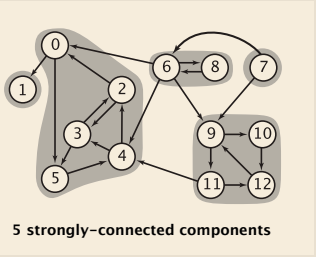
Connected components vs. strongly-connected components

v and w are connected if there is a path between v and w



3 connected components

v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v



5 strongly-connected components

connected component id (easy to compute with DFS)

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

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

constant-time client connectivity query

strongly-connected component id (how to compute?)

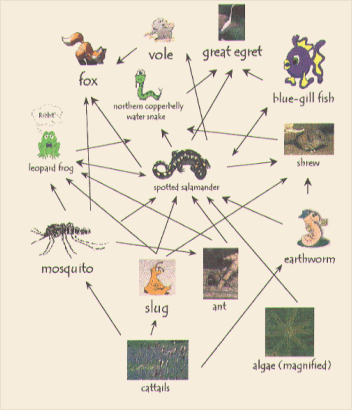
	0	1	2	3	4	5	6	7	8	9	10	11	12
id[]	1	0	1	1	1	1	3	4	3	2	2	2	2

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

constant-time client strong-connectivity query

Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.



http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif

Strong component. Subset of species with common energy flow.

Kosaraju-Sharir algorithm: intuition

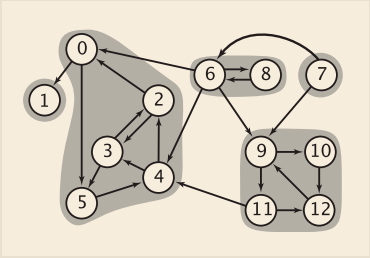
Reverse graph. Strong components in G are same as in G^R.

Kernel DAG. Contract each strong component into a single vertex.

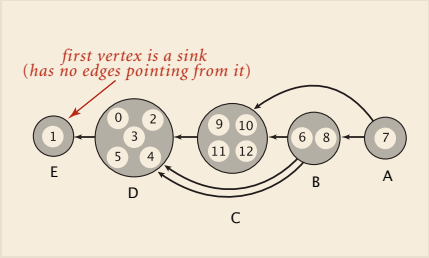
Idea.

how to compute?

- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.



digraph G and its strong components



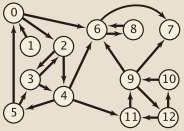
kernel DAG of G (topological order: A B C D E)

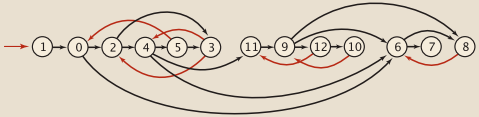
Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G, considering vertices in order given by first DFS.

DFS in reverse digraph G^R





check unmarked vertices in the order
0 1 2 3 4 5 6 7 8 9 10 11 12

reverse postorder for use in second `dfs()`
1 0 2 4 5 3 11 9 12 10 6 7 8

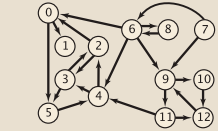
```
dfs(0)
  dfs(6)
    dfs(8)
      check 6
    8 done
    dfs(7)
    7 done
  6 done
  dfs(2)
    dfs(4)
      dfs(11)
        dfs(9)
          dfs(12)
            check 11
            dfs(10)
              check 9
            10 done
          12 done
          check 7
          check 6
  . . .
```

Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

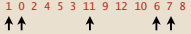
- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G, considering vertices in order given by first DFS.

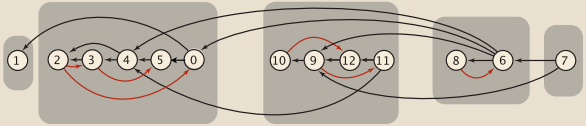
DFS in original digraph G



check unmarked vertices in the order

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






```
dfs(1)
1 done
```

```
dfs(0)
  dfs(5)
    dfs(4)
      dfs(3)
        check 5
        dfs(2)
          check 0
          check 3
        2 done
      3 done
      check 2
    4 done
  5 done
  check 1
0 done
check 2
check 4
check 5
check 3
```

```
dfs(11)
  check 4
  dfs(12)
    dfs(9)
      check 11
      dfs(10)
        check 12
        10 done
    9 done
  12 done
11 done
check 9
check 12
check 10
```

```
dfs(6)
  check 9
  check 4
  dfs(8)
    check 6
  8 done
  check 0
6 done
```

```
dfs(7)
  check 6
  check 9
7 done
check 8
```

Kosaraju-Sharir algorithm

Proposition. Kosaraju-Sharir algorithm computes the strong components of a digraph in time proportional to $E + V$.

Pf.

- Running time: bottleneck is running DFS twice (and computing G^R).
- Correctness: tricky, see textbook (2nd printing).
- Implementation: easy!

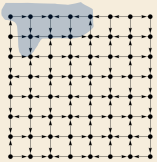
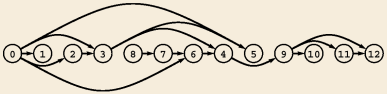
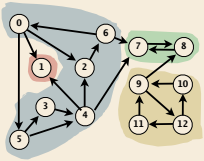
Connected components in an undirected graph (with DFS)

```
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++;
         }
      }
   }

   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);
   }

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

Strong components in a digraph (with two DFSs)

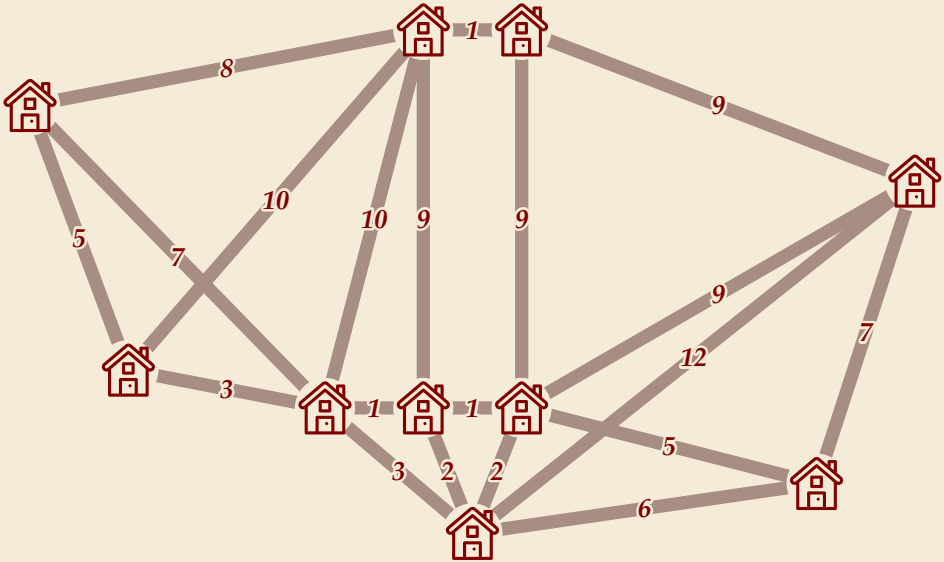
```
public class KosarajuSharirSCC
{
   private boolean marked[];
   private int[] id;
   private int count;

   public KosarajuSharirSCC(Digraph G)
   {
      marked = new boolean[G.V()];
      id = new int[G.V()];
      DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
      for (int v : dfs.reversePostorder())
      {
         if (!marked[v])
         {
            dfs(G, v);
            count++;
         }
      }
   }

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

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

Digraph-processing summary: algorithms of the day

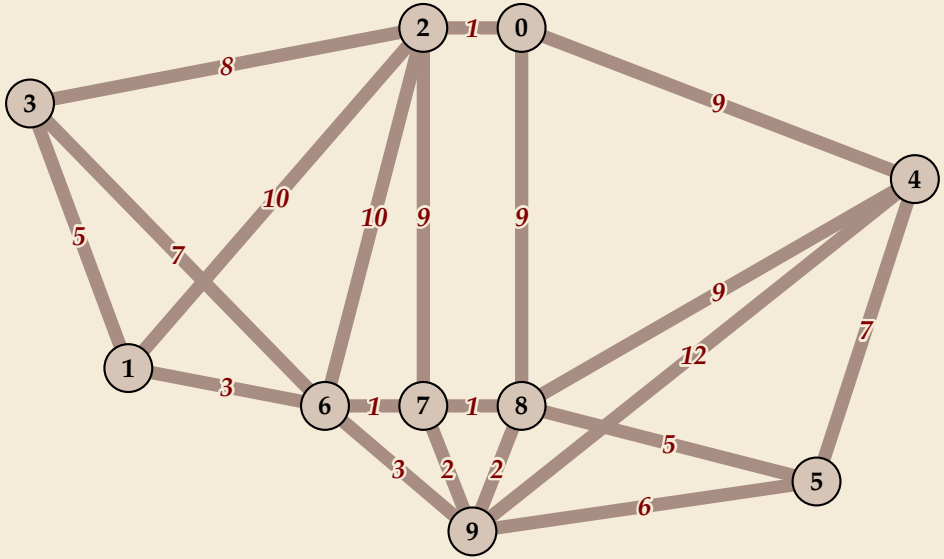
single-source reachability in a digraph		DFS
topological sort in a DAG		DFS
strong components in a digraph		Kosaraju-Sharir DFS (twice)

11.10 Minimale Spannbäume

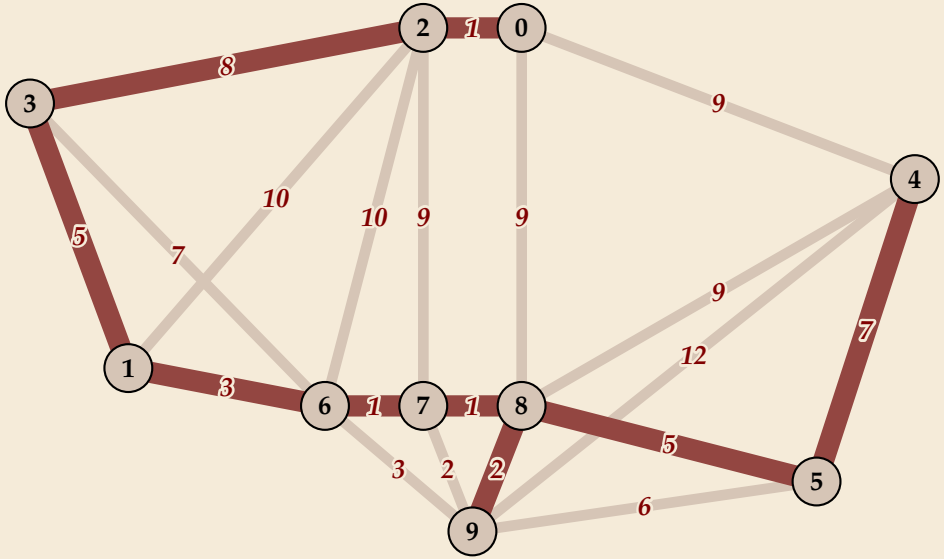
Stromnetze planen



Stromnetze planen



Stromnetze planen

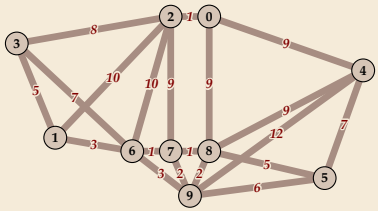


Das Minimale-Spannbaum-Problem

Gegeben: **un**gerichteter, Kanten-**gewichteter**, einfacher, **zusammenhängender** Graph $G = (V, E, c)$

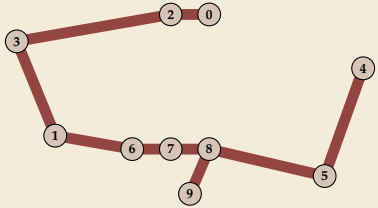
keine Self-Loops, keine parallelen Kanten

Formal: $V = [0..n)$
Kanten $E \subseteq \big\{\{u, v\} : u, v \in V \wedge u \neq v\big\}$
Kosten $c : E \to \mathbb{R}$
für alle $u, v \in V$ existiert Weg $u \rightsquigarrow v$ in (V, E)



Gesucht: ein **Spannbaum** (V, T)
mit **minimalen** Kosten $c(T) := \sum_{e \in T} c(e)$

Formal: $T \subseteq E$
(V, T) ist zusammenhängend und azyklisch
für jeden Spannbaum (V, T') von G ist $c(T') \geq c(T)$.



Applications

MST is fundamental problem with diverse applications.

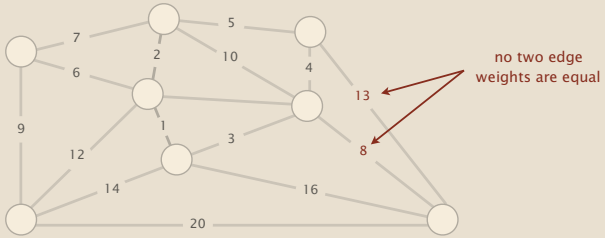
- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, computer, road).

http://www.ics.uci.edu/~eppstein/gina/mst.html

Simplifying assumptions

- Graph is connected.
- Edge weights are distinct.

Consequence. MST exists and is unique.

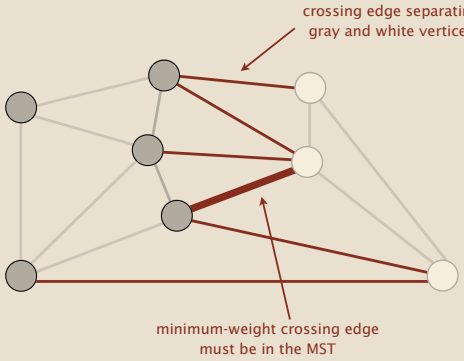


Cut property

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.

Def. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.



Cut property: correctness proof

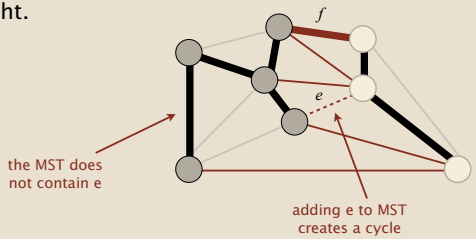
Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.

Def. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.

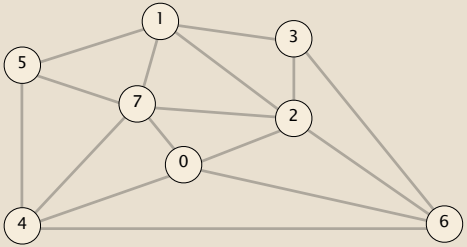
Pf. Suppose min-weight crossing edge e is not in the MST.

- Adding e to the MST creates a cycle.
- Some other edge f in cycle must be a crossing edge.
- Removing f and adding e is also a spanning tree.
- Since weight of e is less than the weight of f, that spanning tree is lower weight.
- Contradiction. ▪



Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

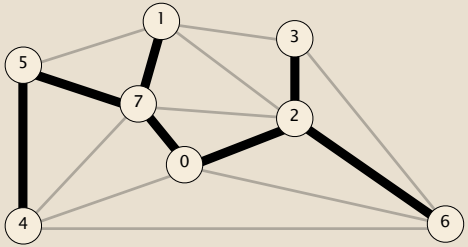


an edge-weighted graph

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Greedy MST algorithm demo

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



MST edges

0-2 5-7 6-2 0-7 2-3 1-7 4-5

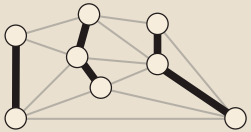
Greedy MST algorithm: correctness proof

Proposition. The greedy algorithm computes the MST.

Pf.

- Any edge colored black is in the MST (via cut property).
- Fewer than $V - 1$ black edges $\Rightarrow$ cut with no black crossing edges. (consider cut whose vertices are any one connected component)

a cut with no black crossing edges



fewer than V-1 edges colored black

Greedy MST algorithm: efficient implementations

Proposition. The greedy algorithm computes the MST.

Efficient implementations. Choose cut? Find min-weight edge?

Ex 1. Kruskal's algorithm. [stay tuned]

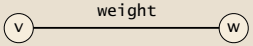
Ex 2. Prim's algorithm. [stay tuned]

Ex 3. Borüvka's algorithm.

Weighted edge API

Edge abstraction needed for weighted edges.

public class Edge implements Comparable<Edge>	
Edge(int v, int w, double weight)	*create a weighted edge v-w*
int either()	*either endpoint*
int other(int v)	*the endpoint that's not v*
int compareTo(Edge that)	*compare this edge to that edge*
double weight()	*the weight*
String toString()	*string representation*



Idiom for processing an edge e: `int v = e.either(), w = e.other(v);`

Weighted edge: Java implementation

```
public class Edge implements Comparable<Edge>
{
   private final int v, w;
   private final double weight;

   public Edge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int either()
   {  return v;  }

   public int other(int vertex)
   {
      if (vertex == v) return w;
      else return v;
   }

   public int compareTo(Edge that)
   {
      if      (this.weight < that.weight) return -1;
      else if (this.weight > that.weight) return +1;
      else                                return  0;
   }
}
```

constructor

either endpoint

other endpoint

compare edges by weight

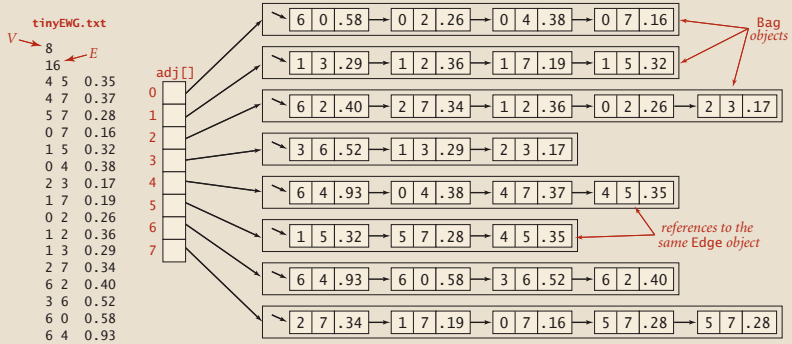
Edge-weighted graph API

public class	EdgeWeightedGraph	
	EdgeWeightedGraph(int V)	*create an empty graph with V vertices*
	EdgeWeightedGraph(In in)	*create a graph from input stream*
void	addEdge(Edge e)	*add weighted edge e to this graph*
Iterable<Edge>	adj(int v)	*edges incident to v*
Iterable<Edge>	edges()	*all edges in this graph*
int	V()	*number of vertices*
int	E()	*number of edges*
String	toString()	*string representation*

Conventions. Allow self-loops and parallel edges.

Edge-weighted graph: adjacency-lists representation

Maintain vertex-indexed array of Edge lists.



Edge-weighted graph: adjacency-lists implementation

```
public class EdgeWeightedGraph
{
   private final int V;
   private final Bag<Edge>[] adj;

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

   public void addEdge(Edge e)
   {
      int v = e.either(), w = e.other(v);
      adj[v].add(e);
      adj[w].add(e);
   }

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

same as Graph, but adjacency lists of Edges instead of integers

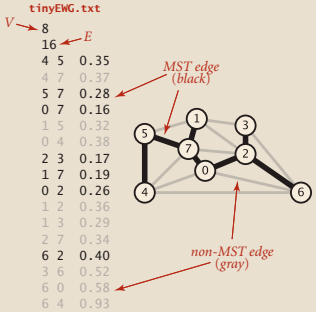
constructor

add edge to both adjacency lists

Minimum spanning tree API

Q. How to represent the MST?

public class MST	
MST(EdgeWeightedGraph G)	*constructor*
Iterable<Edge> edges()	*edges in MST*
double weight()	*weight of MST*

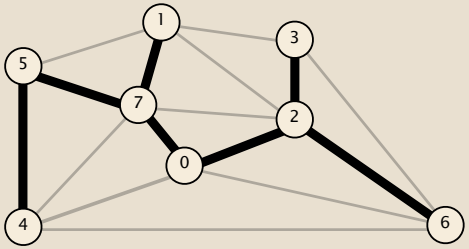



```
% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

Kruskal's algorithm demo

Consider edges in ascending order of weight.

- Add next edge to tree T unless doing so would create a cycle.



a minimum spanning tree

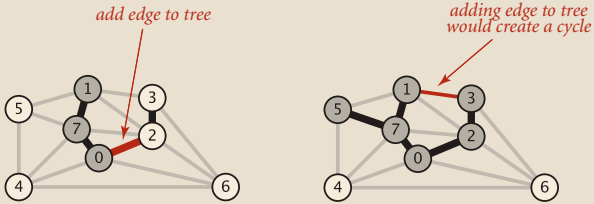
0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's algorithm: implementation challenge

Challenge. Would adding edge v–w to tree T create a cycle? If not, add it.

How difficult?

- $E + V$
- V ← run DFS from v, check if w is reachable (T has at most V – 1 edges)
- $\log V$
- $\log^* V$ ← use the union-find data structure !
- 1

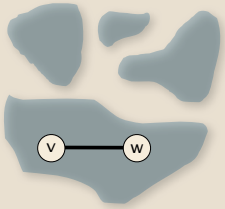


Kruskal's algorithm: implementation challenge

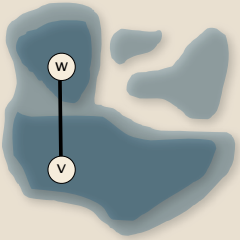
Challenge. Would adding edge v–w to tree T create a cycle? If not, add it.

Efficient solution. Use the union-find data structure.

- Maintain a set for each connected component in T.
- If v and w are in same set, then adding v–w would create a cycle.
- To add v–w to T, merge sets containing v and w.



Case 1: adding v–w creates a cycle



Case 2: add v–w to T and merge sets containing v and w

Kruskal's algorithm: Java implementation

```
public class KruskalMST
{
   private Queue<Edge> mst = new Queue<Edge>();

   public KruskalMST(EdgeWeightedGraph G)
   {
      MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());

      UF uf = new UF(G.V());
      while (!pq.isEmpty() && mst.size() < G.V()-1)
      {
         Edge e = pq.delMin();
         int v = e.either(), w = e.other(v);
         if (!uf.connected(v, w))
         {
            uf.union(v, w);
            mst.enqueue(e);
         }
      }
   }

   public Iterable<Edge> edges()
   {  return mst;  }
}
```

build priority queue (or sort)

greedily add edges to MST

edge v–w does not create cycle

merge sets

add edge to MST

Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Pf.

operation	frequency	time per op
build pq	1	E
delete-min	E	$\log E$
union	V	$\log^* V$ †
connected	E	$\log^* V$ †

† amortized bound using weighted quick union with path compression

recall: log* V ≤ 5 in this universe

Remark. If edges are already sorted, order of growth is $E \log^* V$.

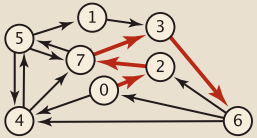
11.11 Kürzeste Wege

Shortest paths in an edge-weighted digraph

Given an edge-weighted digraph, find the shortest path from s to t.

edge-weighted digraph

```
4->5  0.35
5->4  0.35
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```



shortest path from 0 to 6

```
0->2  0.26
2->7  0.34
7->3  0.39
3->6  0.52
```

Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Texture mapping.
- Robot navigation.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

http://en.wikipedia.org/wiki/Seam_carving

Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest path variants

Which vertices?

- Single source: from one vertex s to every other vertex.
- Single sink: from every vertex to one vertex t.
- Source-sink: from one vertex s to another t.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.
- Euclidean weights.
- Arbitrary weights.

Cycles?

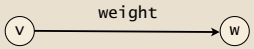
- No directed cycles.
- No "negative cycles."

which variant?

Simplifying assumption. Shortest paths from s to each vertex v exist.

Weighted directed edge API

public class DirectedEdge		
	DirectedEdge(int v, int w, double weight)	*weighted edge v→w*
int	from()	*vertex v*
int	to()	*vertex w*
double	weight()	*weight of this edge*
String	toString()	*string representation*



Idiom for processing an edge `e`: `int v = e.from(), w = e.to();`

Weighted directed edge: implementation in Java

Similar to `Edge` for undirected graphs, but a bit simpler.

```
public class DirectedEdge
{
   private final int v, w;
   private final double weight;

   public DirectedEdge(int v, int w, double weight)
   {
      this.v = v;
      this.w = w;
      this.weight = weight;
   }

   public int from()
   {  return v;  }

   public int to()
   {  return w;  }

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

`from()` and `to()` replace `either()` and `other()`

Edge-weighted digraph API

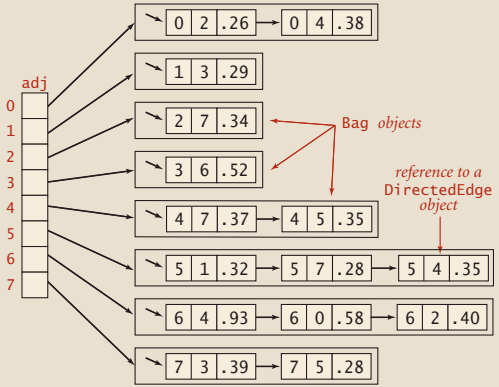
public class	EdgeWeightedDigraph	
	EdgeWeightedDigraph(int V)	*edge-weighted digraph with V vertices*
	EdgeWeightedDigraph(In in)	*edge-weighted digraph from input stream*
void	addEdge(DirectedEdge e)	*add weighted directed edge e*
Iterable<DirectedEdge>	adj(int v)	*edges pointing from v*
int	V()	*number of vertices*
int	E()	*number of edges*
Iterable<DirectedEdge>	edges()	*all edges*
String	toString()	*string representation*

Conventions. Allow self-loops and parallel edges.

Edge-weighted digraph: adjacency-lists representation

tinyEWD.txt

V
8
E
15
4 5 0.35
5 4 0.35
4 7 0.37
5 7 0.28
7 5 0.28
5 1 0.32
0 4 0.38
0 2 0.26
7 3 0.39
1 3 0.29
2 7 0.34
6 2 0.40
3 6 0.52
6 0 0.58
6 4 0.93



Edge-weighted digraph: adjacency-lists implementation in Java

Same as `EdgeWeightedGraph` except replace `Graph` with `Digraph`.

```
public class EdgeWeightedDigraph
{
   private final int V;
   private final Bag<DirectedEdge>[] adj;

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

   public void addEdge(DirectedEdge e)
   {
      int v = e.from();
      adj[v].add(e);
   }

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

add edge e = v→w to only v's adjacency list

Single-source shortest paths API

Goal. Find the shortest path from s to every other vertex.

public class SP		
	SP(EdgeWeightedDigraph G, int s)	*shortest paths from s in graph G*
double	distTo(int v)	*length of shortest path from s to v*
Iterable <DirectedEdge>	pathTo(int v)	*shortest path from s to v*
boolean	hasPathTo(int v)	*is there a path from s to v?*

```
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
   StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
   for (DirectedEdge e : sp.pathTo(v))
      StdOut.print(e + "   ");
   StdOut.println();
}
```

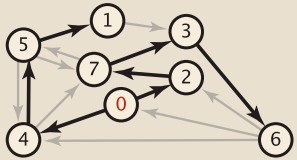
Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A shortest-paths tree (SPT) solution exists. Why?

Consequence. Can represent the SPT with two vertex-indexed arrays:

- `distTo[v]` is length of shortest path from s to v.
- `edgeTo[v]` is last edge on shortest path from s to v.



shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1 0.32	1.05
2	0->2 0.26	0.26
3	7->3 0.37	0.97
4	0->4 0.38	0.38
5	4->5 0.35	0.73
6	3->6 0.52	1.49
7	2->7 0.34	0.60

parent-link representation

Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A shortest-paths tree (SPT) solution exists. Why?

Consequence. Can represent the SPT with two vertex-indexed arrays:

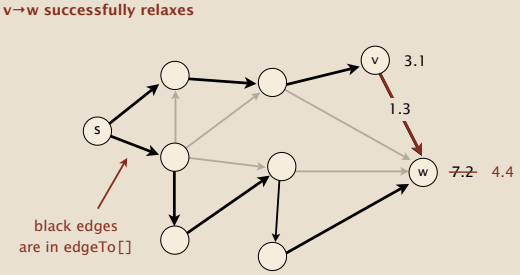
- `distTo[v]` is length of shortest path from s to v.
- `edgeTo[v]` is last edge on shortest path from s to v.

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

public Iterable<DirectedEdge> pathTo(int v)
{
   Stack<DirectedEdge> path = new Stack<DirectedEdge>();
   for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
      path.push(e);
   return path;
}
```

Edge relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest known path from `s` to `v`.
- `distTo[w]` is length of shortest known path from `s` to `w`.
- `edgeTo[w]` is last edge on shortest known path from `s` to `w`.
- If `e` = `v`→`w` gives shorter path to `w` through `v`,
  update both `distTo[w]` and `edgeTo[w]`.



Edge relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest known path from `s` to `v`.
- `distTo[w]` is length of shortest known path from `s` to `w`.
- `edgeTo[w]` is last edge on shortest known path from `s` to `w`.
- If `e` = `v`→`w` gives shorter path to `w` through `v`, update both `distTo[w]` and `edgeTo[w]`.

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
   }
}
```

Shortest-paths optimality conditions

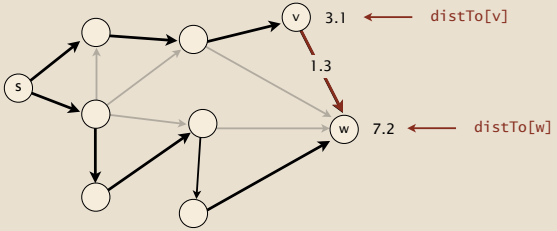
Proposition. Let G be an edge-weighted digraph.

Then `distTo[]` are the shortest path distances from `s` iff:

- `distTo[s]` = 0.
- For each vertex `v`, `distTo[v]` is the length of some path from `s` to `v`.
- For each edge `e` = `v`→`w`, `distTo[w]` ≤ `distTo[v]` + `e.weight()`.

Pf. ⇐ [necessary]

- Suppose that `distTo[w]` > `distTo[v]` + `e.weight()` for some edge `e` = `v`→`w`.
- Then, `e` gives a path from `s` to `w` (through `v`) of length less than `distTo[w]`.



Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.

Then `distTo[]` are the shortest path distances from `s` iff:

- `distTo[s]` = 0.
- For each vertex `v`, `distTo[v]` is the length of some path from `s` to `v`.
- For each edge `e` = `v`→`w`, `distTo[w]` ≤ `distTo[v]` + `e.weight()`.

Pf. ⇒ [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k = w$ is a shortest path from s to w.
- Then,

  `distTo[`v_1`]` ≤ `distTo[`v_0`]` + e_1`.weight()`
  `distTo[`v_2`]` ≤ `distTo[`v_1`]` + e_2`.weight()`
  ...
  `distTo[`v_k`]` ≤ `distTo[`v_{k-1}`]` + e_k`.weight()`

  e_i = i^{th} edge on shortest path from s to w

- Add inequalities; simplify; and substitute `distTo[`v_0`]` = `distTo[s]` = 0:

  `distTo[w]` = `distTo[`v_k`]` ≤ e_1`.weight()` + e_2`.weight()` + … + e_k`.weight()`

  weight of shortest path from s to w

- Thus, `distTo[w]` is the weight of shortest path to w. ▪

  weight of some path from s to w

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Proposition. Generic algorithm computes SPT (if it exists) from `s`.

Pf sketch.

- The entry `distTo[v]` is always the length of a simple path from `s` to `v`.
- Each successful relaxation decreases `distTo[v]` for some `v`.
- The entry `distTo[v]` can decrease at most a finite number of times. ▪

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

- **Relax any edge.**

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

Ex 2. Topological sort algorithm (no directed cycles).

Ex 3. Bellman-Ford algorithm (no negative cycles).

Edsger W. Dijkstra: select quotes

" Do only what only you can do. "

" In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind. "

" The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence. "

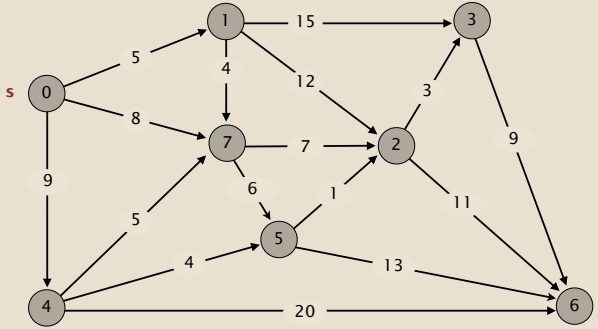
" It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration. "

" APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums. "

Edsger W. Dijkstra
Turing award 1972

Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from `s` (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges pointing from that vertex.



an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Indexed priority queue

Associate an index between 0 and $N - 1$ with each key in a priority queue.

- Supports insert and delete-the-minimum.
- Supports decrease-key given the index of the key.

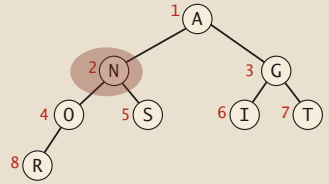
public class IndexMinPQ<Key extends Comparable<Key>>	
IndexMinPQ(int N)	*create indexed priority queue with indices 0, 1, …, N – 1*
void insert(int i, Key key)	*associate key with index i*
void decreaseKey(int i, Key key)	*decrease the key associated with index i*
boolean contains(int i)	*is i an index on the priority queue?*
int delMin()	*remove a minimal key and return its associated index*
boolean isEmpty()	*is the priority queue empty?*
int size()	*number of keys in the priority queue*

Indexed priority queue implementation

Binary heap implementation. [see Section 2.4 of textbook]

- Start with same code as `MinPQ`.
- Maintain parallel arrays `keys[]`, `pq[]`, and `qp[]` so that:
  - `keys[i]` is the priority of `i`
  - `pq[i]` is the index of the key in heap position `i`
  - `qp[i]` is the heap position of the key with index `i`
- Use `swim(qp[i])` to implement `decreaseKey(i, key)`.

i	0	1	2	3	4	5	6	7	8
keys[i]	A	S	O	R	T	I	N	G	-
pq[i]	-	0	6	7	2	1	5	4	3
qp[i]	1	5	4	8	7	6	2	3	-



Dijkstra's algorithm: Java implementation

```
public class DijkstraSP
{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

   public DijkstraSP(EdgeWeightedDigraph G, int s)
   {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());

      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;

      pq.insert(s, 0.0);
      while (!pq.isEmpty())
      {
          int v = pq.delMin();
          for (DirectedEdge e : G.adj(v))
             relax(e);
      }
   }
}
```

relax vertices in order of distance from s

Dijkstra's algorithm: Java implementation

```
private void relax(DirectedEdge e)
{
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight())
   {
       distTo[w] = distTo[v] + e.weight();
       edgeTo[w] = e;
       if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
       else                pq.insert     (w, distTo[w]);
   }
}
```

update PQ

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap	1 †	$\log V$ †	1 †	$E + V \log V$

† amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Es gibt noch einiges mehr zu Graphen!

Es gibt jede Menge weiterer Fragestellungen auf Graphen

- Kürzeste Wege mit negativen Kantengewichten
- Flüsse in Netzwerken (*Wie viel Verkehr von A nach B schafft mein Straßennetz?*)
- Matchings (*Wie viele Paare können gleichzeitig glücklich sein?*)
- Bridges (*Schwachstellen in Netzwerken finden*)
- Euler-Pfade (*genome assembly; Haus des Nikolaus*)
- Graph Coloring (*Mapping von Variablen auf CPU-Register*)
- dynamische Graphen, temporale Graphen

⇝ ***Effiziente Algorithmen*** 😉

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* $d(v) = |E(v)|$

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 $d_{\text{in}}(v)$, $d_{\text{out}}(v)$

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* $d(v) = |E(v)|$

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 $d_{\text{in}}(v)$, $d_{\text{out}}(v)$

- *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., $c[0] = c[n]$

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* $d(v) = |E(v)|$

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 $d_{\text{in}}(v)$, $d_{\text{out}}(v)$

- *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

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