http://algs4.cs.princeton.edu

4.1 UNDIRECTED GRAPHS

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

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.

Paris
www.ratp.fr
La Défense
Gare de Lyon
Gare d'Austerlitz
Gare Montparnasse
Gare du Nord
Gare de l'Est
Gare St-Lazare
Châtelet
Nation
Bastille
Bercy
Porte de Vincennes
Porte de Clignancourt
Porte de la Chapelle
Porte de Pantin
Porte des Lilas
Porte d'Orléans
Porte de Vanves
Porte de Versailles
Porte de Clichy
Porte de Saint-Ouen
Porte de Saint-Cloud
Porte d'Auteuil
Porte Maillot
Porte Dauphine
Porte de Champerret
Porte de la Villette
Porte de Montreuil
Porte de Bagnolet
Porte d'Ivry
Porte d'Italie
Porte de Choisy
Porte de Bagnolet
Porte de Charenton
Porte Dorée
Porte de Pantin
Porte de Montreuil
Porte de Versailles
Porte de Vanves
Porte de Clichy
Porte de Saint-Ouen
Porte de la Villette
Porte de la Chapelle
Porte de Clignancourt
Porte de Pantin
Porte des Lilas
Porte d'Orléans
Porte d'Ivry
Porte d'Italie
Porte de Choisy
Porte de Bagnolet
Porte de Charenton
Porte Dorée
Porte de Montreuil
Porte de Vincennes
Porte de Saint-Cloud
Porte d'Auteuil
Porte Maillot
Porte Dauphine
Porte de Champerret
Porte de la Villette
Porte de Montreuil
Porte de Bagnolet
Porte de Charenton
Porte Dorée
Porte de Pantin
Porte des Lilas
Porte d'Orléans
Porte de Vanves
Porte de Versailles
Porte de Clichy
Porte de Saint-Ouen

Protein-protein interaction network

Reference: Jeong et al, Nature Review | Genetics

10 million Facebook friends

"Visualizing Friendships" by Paul Butler

Graph applications

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	intersection	street
internet	class C network	connection
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

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 path between every pair of 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	*Are two graphs isomorphic?*

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

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

Graph representation

Graph drawing. Provides intuition about the structure of the graph.

two drawings of the same graph

Caveat. Intuition can be misleading.

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*

```
// 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;
}
```

Graph representation: adjacency matrix

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

Undirected graphs: quiz 1

Which is order of growth of running time of the following code fragment if the graph uses the adjacency-matrix representation, where V is the number of vertices and E is the number of edges?

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

prints each edge exactly once

A. V

B. $E + V$

C. V^2

D. $V E$

E. *I don't know.*

Graph representation: adjacency lists

Maintain vertex-indexed array of lists.

Undirected graphs: quiz 2

Which is order of growth of running time of the following code fragment if the graph uses the adjacency-lists representation, where V is the number of vertices and E is the number of edges?

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

prints each edge exactly once

A. V

B. $E + V$

C. V^2

D. $V E$

E. *I don't know.*

Graph representations

In practice. Use adjacency-lists representation.

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

huge number of vertices,
small average vertex degree

Graph representations

In practice. Use adjacency-lists representation.

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

huge number of vertices,
small average vertex degree

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

† disallows parallel edges

Adjacency-list graph representation: Java implementation

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

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

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

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

adjacency lists
(using Bag data type)

create empty graph
with V vertices

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

iterator for vertices adjacent to v

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges

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 newly discovered intersection and passage.
- Retrace steps when no unmarked options.

Maze exploration: easy

Maze exploration: medium

Maze exploration: challenge for the bored

Depth-first search

Goal. Systematically traverse a graph.

Idea. Mimic maze exploration. ← function-call stack acts as ball of string

DFS (to visit a vertex v)

Mark vertex v.

Recursively visit all unmarked vertices w adjacent to v.

Typical applications.

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design challenge. How to implement?

Depth-first search demo

To visit a vertex v :

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

tinyG.txt

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

graph G

Depth-first search demo

To visit a vertex v :

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

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

vertices reachable from 0

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.
- Recursively visit all unmarked vertices adjacent to v.

Data structures.

- Boolean array `marked[]` to mark vertices.
- Integer array `edgeTo[]` to keep track of paths.

  `(edgeTo[w] == v)` means that edge `v-w` taken to discover vertex `w`
- 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])
         {
            edgeTo[w] = v;
            dfs(G, w);
         }
   }
}
```

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

Los amigos de mis amigos son mis amigos

En esta ciudad vive una serie de personas, y sabemos que algunas de ellas son amigas entre sí. De acuerdo con el refrán que dice "*Los amigos de mis amigos son mis amigos*", sabemos que si A y B son amigos y B y C son amigos, entonces también son amigos A y C.

Tu misión consiste en contar las personas en el grupo de amigos más grande.

Los amigos de mis amigos son mis amigos

Representamos los datos del problema mediante un grafo donde los vértices representan a las personas y dos vértices están conectados mediante una arista si sabemos que son amigos.

Cada componente conexa representa un grupo de amigos. Podemos recorrerla con un recorrido en profundidad que vaya contando el número de vértices que pertenecen a esa componente conexa (su tamaño). Necesitamos recorrer todas las componentes recordando cuál es el tamaño mayor. Esa es la solución.

El recorrido en profundidad tiene una complejidad lineal en la suma del número de vértices y aristas, ya que cada vértice se visita una única vez y cuando se visita se recorren todas las aristas adyacentes a él.

Por tanto, la complejidad de la solución está en $O(N + M)$, donde N es el número de personas y M el número de relaciones de amistad dadas.

Los amigos de mis amigos son mis amigos

```
class MaximaComponenteConexa {
public:
   MaximaComponenteConexa(Grafo const& G) : marked(G.V(), false),
                                            _maximo(0) {
      for (int v = 0; v < G.V(); ++v) {
         if (!marked[v]) { // se recorre una nueva componente conexa
            int tam = 0;
            dfs(G, v, tam);
            _maximo = max(_maximo, tam);
         }
      }
   }

   // tamaño máximo de una componente conexa
   int maximo() const {
      return _maximo;
   }
```

Los amigos de mis amigos son mis amigos

```
private:
   vector<bool> marked;  // marked[v] = se ha visitado el vértice v?
   int _maximo;

   // recorrido en profundidad de la componente de v
   void dfs(Grafo const& G, int v, int & tam) {
      marked[v] = true;
      ++tam;
      for (int w : G.ady(v)) {
         if (!marked[w]) {
            dfs(G, w, tam);
         }
      }
   }
};
```

Los amigos de mis amigos son mis amigos

```
void resuelveCaso() {
   int V, E;

   cin >> V >> E;

   Grafo grafo(V);

   int v, w;
   for (int i = 0; i < E; ++i) {
      cin >> v >> w;
      grafo.ponArista(v-1, w-1);
   }

   MaximaComponenteConexa mcc(grafo);
   cout << mcc.maximo() << '\n';

}
```

Depth-first search application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

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

recolor red blob to blue

Depth-first search application: flood fill

Change color of entire blob of neighboring red pixels to blue.

Build a grid graph.

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

Detección de manchas negras

Dado un *bitmap* de píxeles blancos y negros, queremos saber el número de *manchas negras* que contiene y el tamaño (número de píxeles) de la mancha negra más grande. Dos píxeles negros pertenecen a la misma mancha si se puede pasar de uno a otro atravesando solamente píxeles negros y moviéndonos píxel a píxel solamente en horizontal o vertical.

En esta imagen aparecen 4 manchas y la mancha más grande (marcada con puntos blancos) tiene 10 píxeles.

Detección de manchas negras

```
using Mapa = vector<string>;  // grafo implícito en el mapa

class Manchas {
public:
   Manchas(Mapa const& M) : F(M.size()), C(M[0].size()),
        marked(F, vector<bool>(C, false)), _numero(0), _maximo(0) {
      for (int i = 0; i < F; ++i) {
         for (int j = 0; j < C; ++j) {
            if (M[i][j] == '#' && !marked[i][j]) {
               // se recorre una nueva mancha
               ++_numero;
               int nuevotam = 0;
               dfs(M, i, j, nuevotam);
               _maximo = max(_maximo, nuevotam);
            }
         }
      }
   }

   int numero() const { return _numero; }

   int maximo() const { return _maximo; }
```

Detección de manchas negras

```
private:
  int F, C; // tamaño del mapa
  vector<vector<bool>> marked;  // marked[i][j] = se ha visitado <i,j>?
  int _numero;  // número de manchas
  int _maximo;  // tamaño de la mancha más grande

  bool correcta(int i, int j) const {
     return 0 <= i && i < F && 0 <= j && j < C;
  }
  const vector<pair<int,int>> dirs = {{1,0},{0,1},{-1,0},{0,-1}};

  // recorrido en profundidad de la mancha del vértice <i,j>
  void dfs(Mapa const& M, int i, int j, int & tam) {
     marked[i][j] = true;
     ++tam;
     for (auto d : dirs) {
        int ni = i + d.first, nj = j + d.second;
        if (correcta(ni,nj) && M[ni][nj] == '#' && !marked[ni][nj]) {
           dfs(M, ni, nj, tam);
        }
     }
  }
};
```

Detección de manchas negras

```
bool resuelveCaso() {
   int F, C;
   cin >> F >> C; // número de filas y columnas
   if (!cin)
      return false;

   Mapa mapa(F);

   // leemos la imagen
   for (string & linea : mapa) {
      cin >> linea;
   }

   // y la analizamos
   Manchas manchas(mapa);
   cout << manchas.numero() << ' ' << manchas.maximo() << '\n';
   return true;
}
```

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges

Graph search

Tree traversal. Many ways to explore every vertex in a binary tree.

- Inorder: `A C E H M R S X`
- Preorder: `S E A C R H M X`
- Postorder: `C A M H R E X S`
- Level-order: `S E X A R C H M`

Graph search. Many ways to explore every vertex in a graph.

- Preorder: vertices in order DFS calls `dfs(G, v)`.
- Postorder: vertices in order DFS returns from `dfs(G, v)`.
- Level-order: vertices in increasing order of distance from `s`.

Breadth-first search demo

Repeat until queue is empty:

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

tinyCG.txt

```
V
6
8
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2
```

E

graph G

Breadth-first search demo

Repeat until queue is empty:

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

v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	2	2
4	2	2
5	0	1

done

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 marked neighbors to the queue, and mark them.

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

Búsqueda en anchura

```
class BusquedaAnchura {
public:
   BusquedaAnchura(Grafo const& G, int s)
   : marked(G.V(), false), distTo(G.V()) {
      bfs(G, s);
   }

   // ¿Hay camino del origen a v?
   bool hayCamino(int v) const {
      return marked[v];
   }

   // número de aristas entre s y v
   int distancia(int v) const {
      return distTo[v];
   }
```

Búsqueda en anchura

```
private:
  vector<bool> marked; // marked[v] = ¿hay camino de s a v?
  vector<int> distTo;  // distTo[v] = aristas en el camino s-v más corto

  void bfs(Grafo const& G, int s) {
     queue<int> q;
     distTo[s] = 0;
     marked[s] = true;
     q.push(s);
     while (!q.empty()) {
        int v = q.front(); q.pop();
        for (int w : G.ady(v)) {
           if (!marked[w]) {
              distTo[w] = distTo[v] + 1;
              marked[w] = true;
              q.push(w);
           }
        }
     }
  }
};
```

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

http://algs4.cs.princeton.edu

4.1 Undirected Graphs

- *introduction*
- *graph API*
- *depth-first search*
- *breadth-first search*
- *challenges*

Graph-processing challenge 1

Problem. Identify connected components.

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

How difficult?

A. Any programmer could do it.

B. Typical diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

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

Graph-processing challenge 2

Problem. Is a graph bipartite?

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

How difficult?

A. Any programmer could do it.

B. Typical diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

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

{ 0, 3, 4 }

Graph-processing challenge 3

Problem. Find a cycle in a graph (if one exists).

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

0-5-4-6-0

How difficult?

A. Any programmer could do it.

B. Typical diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

Graph-processing challenge 4

Problem. Is there a (general) cycle that uses every edge exactly once?

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

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

How difficult?

A. Any programmer could do it.

B. Typical diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

Graph-processing challenge 5

Problem. Is there a cycle that contains every vertex exactly once?

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

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

How difficult?

A. Any programmer could do it.

B. Typical diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

Graph-processing challenge 6

Problem. Are two graphs identical except for vertex names?

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

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

How difficult?

A. Any programmer could do it.

B. Typical diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows.

0↔4, 1↔3, 2↔2, 3↔6, 4↔5, 5↔0, 6↔1

Graph-processing challenge 7

Problem. Can you draw a graph in the plane with no crossing edges?

try it yourself at http://planarity.net

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

How difficult?

A. Any programmer could do it.

B. Typical diligent algorithms student could do it.

C. Hire an expert.

D. Intractable.

E. No one knows

Graph traversal summary

BFS and DFS enables efficient solution of many (but not all) graph problems.

graph problem	BFS	DFS	time
s–t path	✔	✔	$E + V$
shortest s–t path	✔		$E + V$
cycle	✔	✔	$E + V$
Euler cycle		✔	$E + V$
Hamilton cycle			$2^{1.657\,V}$
bipartiteness (odd cycle)	✔	✔	$E + V$
connected components	✔	✔	$E + V$
biconnected components		✔	$E + V$
planarity		✔	$E + V$
graph isomorphism			$2^{c\sqrt{V \log V}}$

http://algs4.cs.princeton.edu

4.2 Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

Last updated on Mar 24, 2015, 8:05 PM

Directed graphs

Digraph. Set of vertices connected pairwise by directed edges.

Road network

Vertex = intersection; edge = one-way street.

Political blogosphere graph

Vertex = political blog; edge = link.

The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005

Uber taxi graph

Vertex = taxi pickup; edge = taxi ride.

http://blog.uber.com/2012/01/09/uberdata-san-franciscomics/

Combinational circuit

Vertex = logical gate; edge = wire.

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 ?*

http://algs4.cs.princeton.edu

4.2 Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

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 adjacent from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`Digraph`	`reverse()`	*reverse of this digraph*
`String`	`toString()`	*string representation*

Digraph representation: adjacency lists

Maintain vertex-indexed array of lists.

Directed graphs: quiz 1

Which is order of growth of running time of the following code fragment if the digraph uses the adjacency-lists representation?

A. V

B. $E + V$

C. V^2

D. $V E$

E. *I don't know.*

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

prints each edge exactly once

Digraph representations

In practice. Use adjacency-lists representation.

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

huge number of vertices,
small average vertex outdegree

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices adjacent 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 adjacent from v

http://algs4.cs.princeton.edu

4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
- strong components

Reachability

Problem. Find all vertices reachable from s along a directed path.

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 vertex v.

Recursively visit all unmarked vertices w adjacent from v.

Depth-first search demo

To visit a vertex v :

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

a directed graph

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

Depth-first search demo

To visit a vertex v :

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

reachable from vertex 0

v	marked[]	edgeTo[]
0	T	–
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	–
7	F	–
8	F	–
9	F	–
10	F	–
11	F	–
12	F	–

reachable from 0

Depth-first search (in undirected graphs)

Recall code for undirected graphs.

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

   public DepthFirstSearch(Graph G, int s)
   {
      marked = new boolean[G.V()];
      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);
   }

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

true if connected to s

constructor marks vertices connected to s

recursive DFS does the work

client can ask whether any vertex is connected to s

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 adjacent 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 $E + V$.

Directed breadth-first search demo

Repeat until queue is empty:

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

tinyDG2.txt

V → 6

E → 8

5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2

graph G

Directed breadth-first search demo

Repeat until queue is empty:

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

v	edgeTo[]	distTo[]
0	–	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

done

MULTIPLE-SOURCE SHORTEST PATHS

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

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

- Shortest path to 4 is $7 \rightarrow 6 \rightarrow 4$.
- Shortest path to 5 is $7 \rightarrow 6 \rightarrow 0 \rightarrow 5$.
- Shortest path to 12 is $10 \rightarrow 12$.

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

Directed graphs: quiz 2

Suppose that you want to design a web crawler. Which graph search algorithm should you use?

A. Depth-first search

B. Breadth-first search

C. Either A or B

D. Neither A nor B

E. *I don't know.*

Web crawler output

BFS crawl

```
http://www.princeton.edu
http://www.w3.org
http://ogp.me
http://giving.princeton.edu
http://www.princetonartmuseum.org
http://www.goprincetontigers.com
http://library.princeton.edu
http://helpdesk.princeton.edu
http://tigernet.princeton.edu
http://alumni.princeton.edu
http://gradschool.princeton.edu
http://vimeo.com
http://princetonusg.com
http://artmuseum.princeton.edu
http://jobs.princeton.edu
http://odoc.princeton.edu
http://blogs.princeton.edu
http://www.facebook.com
http://twitter.com
http://www.youtube.com
http://deimos.apple.com
http://qeprize.org
http://en.wikipedia.org
...
```

DFS crawl

```
http://www.princeton.edu
http://deimos.apple.com
http://www.youtube.com
http://www.google.com
http://news.google.com
http://csi.gstatic.com
http://googlenewsblog.blogspot.com
http://labs.google.com
http://groups.google.com
http://img1.blogblog.com
http://feeds.feedburner.com
http:/buttons.googlesyndication.com
http://fusion.google.com
http://insidesearch.blogspot.com
http://agoogleaday.com
http://static.googleusercontent.com
http://searchresearch1.blogspot.com
http://feedburner.google.com
http://www.dot.ca.gov
http://www.TahoeRoads.com
http://www.LakeTahoeTransit.com
http://www.laketahoe.com
http://ethel.tahoeguide.com
...
```

Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say `www.princeton.edu`.

Solution. [BFS with implicit digraph]

- Choose root web page as source s.
- Maintain a `Queue` of websites to explore.
- Maintain a `SET` of discovered websites.
- Dequeue the next website and enqueue websites to which it links (provided you haven't done so before).

Bare-bones web crawler: Java implementation

```
Queue<String> queue = new Queue<String>();
SET<String> marked = new SET<String>();

String root = "http://www.princeton.edu";
queue.enqueue(root);
marked.add(root);

while (!queue.isEmpty())
{
   String v = queue.dequeue();
   StdOut.println(v);
   In in = new In(v);
   String input = in.readAll();

   String regexp = "http://(\\w+\\.)+(\\w+)";
   Pattern pattern = Pattern.compile(regexp);
   Matcher matcher = pattern.matcher(input);
   while (matcher.find())
   {
      String w = matcher.group();
      if (!marked.contains(w))
      {
         marked.add(w);
         queue.enqueue(w);
      }
   }
}
```

queue of websites to crawl

set of marked websites

start crawling from root website

read in raw html from next website in queue

use regular expression to find all URLs in website of form `http://xxx.yyy.zzz` [crude pattern misses relative URLs]

if unmarked, mark it and put on the queue

http://algs4.cs.princeton.edu

4.2 Directed Graphs

- *introduction*
- *digraph API*
- *digraph search*
- *topological sort*
- *strong components*

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

Topological sort

DAG. Directed acyclic graph.

Topological sort. Redraw DAG so all edges point upwards.

0→5	0→2
0→1	3→6
3→5	3→4
5→2	6→4
6→0	3→2
1→4	

directed edges

DAG

topological order

Solution. DFS. What else?

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

tinyDAG7.txt

```
7
11
 0  5
 0  2
 0  1
 3  6
 3  5
 3  4
 5  2
 6  4
 6  0
 3  2
```

a directed acyclic graph

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

postorder

4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4

done

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 appears before v in postorder
- Case 2: dfs(w) has not yet been called.
  - dfs(w) will get called directly or indirectly by dfs(v)
  - so, dfs(w) will finish before dfs(v)
  - thus, w appears before v in postorder
- Case 3: dfs(w) has already been called, but has not yet returned.
  - function-call stack contains path from w to v
  - edge v→w would complete a cycle
  - contradiction (this case can't happen in a DAG)

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);
}
```

http://algs4.cs.princeton.edu

4.2 Directed Graphs

- introduction
- digraph API
- digraph search
- topological sort
- **strong components**

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

Directed graphs: quiz 3

How many strong components are in a DAG with V vertices and E edges?

A. 0

B. 1

C. V

D. E

E. *I don't know.*

Connected components vs. strongly-connected components

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

3 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

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

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.

Strong component application: software modules

Software module dependency graph.

- Vertex = software module.
- Edge: from module to dependency.

Firefox

Internet Explorer

Strong component. Subset of mutually interacting modules.

Approach 1. Package strong components together.

Approach 2. Use to improve design!

Strong components algorithms: brief history

1960s: Core OR problem.

- Widely studied; some practical algorithms.
- Complexity not understood.

1972: linear-time DFS algorithm (Tarjan).

- Classic algorithm.
- Level of difficulty: Algs4++.
- Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju–Sharir).

- Forgot notes for lecture; developed algorithm in order to teach it!
- Later found in Russian scientific literature (1972).

1990s: more easy linear-time algorithms.

- Gabow: fixed old OR algorithm.
- Cheriyan–Mehlhorn: needed one-pass algorithm for LEDA.

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 demo

Phase 1. Compute reverse postorder in G^R.

Phase 2. Run DFS in G, visiting unmarked vertices in reverse postorder of G^R.

digraph G

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R.

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

reverse digraph G^R

Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in G, visiting unmarked vertices in reverse postorder of G^R.

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

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

done

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];  }
}
```

http://algs4.cs.princeton.edu

1.5 UNION–FIND

- *union–find data type*
- *quick-find*
- *quick-union*
- *improvements*
- *applications*

Last updated on 9/14/17 9:04 PM

http://algs4.cs.princeton.edu

1.5 Union–find

- *union–find data type*
- *quick-find*
- *quick-union*
- *improvements*
- *applications*

Union-find data type

Disjoint sets. A collection of sets; each element in exactly one set.

Find. Return a "canonical" element in the set containing p?

Union. Merge the set containing p with the set containing q.

find(1) = find(4) = find(5) = 4

`{ 0 } { 1, 4, 5 } { 2, 3, 6, 7 }`

8 elements, 3 disjoint sets

union(2, 5)

`{ 0 } { 1, 2, 3, 4, 5, 6, 7 }`

2 disjoint sets

Simplifying assumption. The n elements are named 0, 1, ..., $n - 1$.

Union–find data type (API)

Goal. Design an efficient union–find data type.

- Number of elements n can be huge.
- Number of operations m can be huge.
- Union and find operations can be intermixed.

public class UF	
`UF(int n)`	*initialize union–find data structure with n singleton sets* (0 *to* $n - 1$)
`void union(int p, int q)`	*merge sets containing elements p and q*
`int find(int p)`	*canonical element in set containing p* (0 *to* $n - 1$)

An application: dynamic connectivity

Given n vertices, support two operations:

- Add edge: directly connect two vertices with an edge.
- Connection query: is there a path connecting two vertices?

add edge 4–3

add edge 3–8

add edge 6–5

add edge 9–4

add edge 2–1

are 8 *and* 9 *connected?* ✔

are 5 *and* 7 *connected?* ✕

add edge 5–0

add edge 7–2

add edge 6–1

add edge 1–0

are 5 *and* 7 *connected?* ✔

A larger connectivity example

Q. Is there a path connecting vertices v and w ?

finding a path is a slightly harder problem
(stay tuned for graph algorithms in Chapter 4)

A. Yes.

Modeling the dynamic-connectivity problem

Q. How to model the dynamic-connectivity problem using union–find?

A. Maintain disjoint sets that correspond to connected components.

Connected component. Maximal set of vertices that are mutually connected.

3 connected components

```
{ 0 } { 1, 4, 5 } { 2, 3, 6, 7 }
```

3 disjoint sets

Modeling the dynamic-connectivity problem

Q. How to model the dynamic-connectivity problem using union–find?

A. Maintain disjoint sets that correspond to connected components.

- Add edge between vertices v and w.
- Are vertices v and w connected?

add edge 2–5

3 connected components

are vertices 5 and 6 connected?

2 connected components

union(2, 5)

```
{ 0 } { 1, 4, 5 } { 2, 3, 6, 7 }
```

3 disjoint sets

find(5) == find(6) ✔

```
{ 0 } { 1, 2, 3, 4, 5, 6, 7 }
```

2 disjoint sets

Dynamic-connectivity client

- Read number of vertices n from standard input.
- Repeat:
  - read vertex pairs pairs v and w from standard input
  - if v and w are not yet connected, add edge v–w and print pair

```
public static void main(String[] args)
{
   int n = StdIn.readInt();
   UF uf = new UF(n);
   while (!StdIn.isEmpty())
   {
      int v = StdIn.readInt();
      int w = StdIn.readInt();
      if (uf.find(v) != uf.find(w))
      {
         uf.union(v, w);
         StdOut.println(v + " " + w);
      }
   }
}
```

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```

already connected
(don't print these)

http://algs4.cs.princeton.edu

1.5 Union–find

- *union–find data type*
- *quick-find*
- *quick-union*
- *improvements*
- *applications*

Quick-find [eager approach]

Data structure.

- Integer array `id[]` of length `n`.
- Interpretation: `id[p]` is canonical element in the set containing `p`.

	0	1	2	3	4	5	6	7	8	9
`id[]`	0	1	1	8	8	0	0	1	8	8

3 disjoint sets

Q. How to implement `find(p)`?

A. Easy, just return `id[p]`.

Quick-find [eager approach]

Data structure.

- Integer array `id[]` of length `n`.
- Interpretation: `id[p]` is canonical element in the set containing `p`.

union(6, 1)

	0	1	2	3	4	5	6	7	8	9
`id[]`	1	1	1	8	8	1	1	1	8	8

problem: many values can change

Q. How to implement `union(p, q)`?

A. Change all entries whose identifier equals `id[p]` to `id[q]`.

Quick-find demo

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

Quick-find: Java implementation

```
public class QuickFindUF
{
   private int[] id;

   public QuickFindUF(int n)
   {
      id = new int[n];
      for (int i = 0; i < n; i++)
         id[i] = i;
   }

   public int find(int p)
   {  return id[p];  }

   public void union(int p, int q)
   {
      int pid = id[p];
      int qid = id[q];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = qid;
   }
}
```

set id of each element to itself
(n array accesses)

return the id of p
(1 array access)

change all entries with id[p] to id[q]
(n+2 to 2n+2 array accesses)

http://algs4.cs.princeton.edu/15uf/QuickFindUF.java.html

Quick-find is too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find
quick-find	n	n	1

number of array accesses (ignoring leading constant)

Union is too expensive. Processing a sequence of n union operations on n elements takes more than n^2 array accesses.

quadratic

http://algs4.cs.princeton.edu

1.5 Union–find

- *union–find data type*
- *quick-find*
- *quick-union*
- *improvements*
- *applications*

Quick-union [lazy approach]

Data structure.

- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.
- Interpretation: elements in one tree correspond to one set.

0	1	2	3	4	5	6	7	8	9
0	1	9	4	9	6	6	7	8	9

find(i) = 9

{ 0 } { 1 } { 2, 3, 4, 9 } { 5, 6 } { 7 } { 8 }

6 disjoint sets (6 trees)

Q. How to implement `find(p)` operation?

A. Return root of tree containing `p`.

Quick-union quiz

Data structure.

- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.
- Interpretation: elements in one tree correspond to one set.

0	1	2	3	4	5	6	7	8	9
0	1	9	4	9	6	6	7	8	9

How to implement `union(3, 5)` ?

A. Set parent of 3 to 5.

B. Set parent of 9 to 5.

C. Set parent of 9 to 6.

D. Set parents of 2, 3, 4, and 9 each to 6.

Quick-union [lazy approach]

Data structure.

- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.
- Interpretation: elements in one tree correspond to one set.

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	9

Q. How to implement `union(p, q)`?

A. Set parent of `p`'s root to parent of `q`'s root.

Quick-union [lazy approach]

Data structure.

- Integer array `parent[]` of length `n`, where `parent[i]` is parent of `i` in tree.
- Interpretation: elements in one tree correspond to one set.

	0	1	2	3	4	5	6	7	8	9
union(3, 5)	0	1	9	4	9	6	6	7	8	6

only one value changes

Q. How to implement `union(p, q)`?

A. Set parent of `p`'s root to parent of `q`'s root.

Quick-union demo

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

Quick-union: Java implementation

```
public class QuickUnionUF
{
   private int[] parent;

   public QuickUnionUF(int n)
   {
      parent = new int[n];
      for (int i = 0; i < n; i++)
          parent[i] = i;
   }

   public int find(int p)
   {
      while (p != parent[p])
          p = parent[p];
      return p;
   }

   public void union(int p, int q)
   {
      int rootP = find(p);
      int rootQ = find(q);
      parent[rootP] = rootQ;
   }
}
```

set parent of each element to itself
(n array accesses)

chase parent pointers until reach root
(depth of p array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

http://algs4.cs.princeton.edu/15uf/QuickUnionUF.java.html

Quick-union is also too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n

← worst case

number of array accesses (ignoring leading constant)

Quick-find defect.

- Union too expensive (more than n array accesses).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be more than n array accesses).

worst-case input

union(0, 1)

union(0, 2)

union(0, 3)

union(0, 4)

http://algs4.cs.princeton.edu

1.5 Union–find

- *union–find data type*
- *quick-find*
- *quick-union*
- *improvements*
- *applications*

Weighted quick-union quiz

When merging two trees, which strategy is most effective?

A. Link the root of the smaller tree to the root of the larger tree.

B. Link the root of the larger tree to the root of the smaller tree.

C. Link the root of the shorter tree to the root of the taller tree.

D. Link the root of the taller tree to the root of the shorter tree.

shorter and larger tree
(height = 2, size = 14)

taller and smaller tree
(height = 5, size = 9)

Improvement 1: weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of elements).
- Always link root of smaller tree to root of larger tree.

reasonable alternative:
union by height/rank

Weighted quick-union quiz

Suppose that the `parent[]` array during weighted quick-union is:

	0	1	2	3	4	5	6	7	8	9
parent[]	0	0	0	0	0	0	7	8	8	8

Which `parent[]` entry changes during `union(2, 6)`?

A. `parent[0]`

B. `parent[2]`

C. `parent[6]`

D. `parent[8]`

Weighted quick-union demo

	0	1	2	3	4	5	6	7	8	9
parent[]	0	1	2	3	4	5	6	7	8	9

Quick-union vs. weighted quick-union: larger example

quick-union

average distance to root: 5.11

weighted

average distance to root: 1.52

Quick-union and weighted quick-union (100 sites, 88 `union()` operations)

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `size[i]` to count number of elements in the tree rooted at `i`, initially `1`.

- Find: identical to quick-union.
- Union: link root of smaller tree to root of larger tree; update `size[]`.

```
public void union(int p, int q)
{
   int root1 = find(p);
   int root2 = find(q);
   if (root1 == root2) return;

   if (size[root1] >= size[root2])
   { int temp = root1; root1 = root2; root2 = temp; }

   parent[root1] = root2;
   size[root2] += size[root1];

}
```

`root1` is root of smaller tree

link root of smaller tree to root of larger tree

http://algs4.cs.princeton.edu/15uf/WeightedQuickUnionUF.java.html

Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p.
- Union: takes constant time, given two roots.

Proposition. Depth of any node x is at most $\lg n$. ← in computer science, lg means base-2 logarithm

n = 10

depth(x) = 3 ≤ lg n

Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p.
- Union: takes constant time, given two roots.

Proposition. Depth of any node x is at most $\lg n$. ← in computer science, lg means base-2 logarithm

Pf. What causes the depth of element x to increase?

Increases by 1 when root of tree T_1 containing x is linked to root of tree T_2.

- The size of the tree containing x at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing x can double at most $\lg n$ times. Why?

1
2
4
8
16
⋮
n

lg n

Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p.
- Union: takes constant time, given two roots.

Proposition. Depth of any node x is at most $\lg n$.

algorithm	initialize	union	find
quick-find	n	n	1
quick-union	n	n	n
weighted quick-union	n	$\log n$	$\log n$

log mean logarithm, for some constant base

number of array accesses (ignoring leading constant)

Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.

Path compression: Java implementation

Two-pass implementation: add second loop to `root()` to set the `id[]` of each examined node to the root.

Simpler one-pass variant: Make every other node in path point to its grandparent (thereby halving path length).

```
private int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression: amortized analysis

Proposition. [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of M union–find ops on N objects makes $\leq c(N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M\,\alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

iterate log function

Linear-time algorithm for M union-find ops on N objects?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

Amazing fact. [Fredman-Saks] No linear-time algorithm exists.

in "cell-probe" model of computation

Summary

Bottom line. Weighted quick union (with path compression) makes it possible to solve problems that could not otherwise be addressed.

algorithm	worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

M union-find operations on a set of N objects

Ex. [10^9 unions and finds with 10^9 objects]

- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

http://algs4.cs.princeton.edu

4.3 Minimum Spanning Trees

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- *Kruskal's algorithm*
- *Prim's algorithm*
- *context*

Last updated on Mar 30, 2015, 6:21 AM

http://algs4.cs.princeton.edu

4.3 Minimum Spanning Trees

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- *Kruskal's algorithm*
- *Prim's algorithm*
- *context*

Spanning tree

Def. A spanning tree of an undirected graph G is a subgraph T that is:

- A tree: connected and acyclic.
- Spanning: includes all of the vertices.

graph G

spanning tree T

Spanning tree

Def. A spanning tree of an undirected graph G is a subgraph T that is:

- A tree: connected and acyclic.
- Spanning: includes all of the vertices.

not connected

Spanning tree

Def. A spanning tree of an undirected graph G is a subgraph T that is:

- A tree: connected and acyclic.
- Spanning: includes all of the vertices.

not a tree (cyclic)

Spanning tree

Def. A spanning tree of an undirected graph G is a subgraph T that is:

- A tree: connected and acyclic.
- Spanning: includes all of the vertices.

not spanning

Quiz 1: spanning trees

Let T be a spanning tree of a connected graph G with V vertices.
Which of the following statements are true?

A. T contains exactly $V - 1$ edges.

B. Removing any edge from T disconnects it.

C. Adding any edge to T creates a cycle.

D. All of the above.

E. *I don't know.*

spanning tree T of graph G

Minimum spanning tree problem

Input. Connected, undirected graph G with positive edge weights.

edge-weighted digraph G

Minimum spanning tree problem

Input. Connected, undirected graph G with positive edge weights.

Output. A spanning tree of minimum weight.

minimum spanning tree T
(weight = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7)

Brute force. Try all spanning trees? (Impractical.)

Minimum spanning trees: quiz 1

Let G be a connected edge-weighted graph with V vertices and E edges. How many edges are in a MST of G?

A. $V - 1$

B. V

C. $E - 1$

D. E

E. *I don't know.*

Network design

MST of bicycle routes in North Seattle

http://www.flickr.com/photos/ewedistrict/21980840

Models of nature

MST of random graph

http://algo.inria.fr/broutin/gallery.html

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

For simplicity, we assume

- The graph is connected. $\Rightarrow$ MST exists.
- The edge weights are distinct. $\Rightarrow$ MST 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.

Minimum spanning trees: quiz 2

Which is the min weight edge crossing the cut $\{ 2, 3, 5, 6 \}$?

A. 0–7 (0.16)

B. 2–3 (0.17)

C. 0–2 (0.26)

D. 5–7 (0.28)

E. *I don't know.*

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

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 has lower weight.
- Contradiction. ▪

Removing two simplifying assumptions

Q. What if edge weights are not all distinct?

A. Greedy MST algorithm correct even if equal weights are present!
(our correctness proof fails, but that can be fixed)

1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50

1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50

Q. What if graph is not connected?

A. Compute minimum spanning forest = MST of each component.

can independently compute MSTs of components

4	5	0.61
4	6	0.62
5	6	0.88
1	5	0.11
2	3	0.35
0	3	0.6
1	6	0.10
0	2	0.22

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*

http://algs4.cs.princeton.edu

4.3 Minimum Spanning Trees

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- *Kruskal's algorithm*
- *Prim's algorithm*
- *context*

Kruskal's algorithm demo

Consider edges in ascending order of weight.

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

an edge-weighted graph

graph edges sorted by weight

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: correctness proof

Proposition. [Kruskal 1956] Kruskal's algorithm computes the MST.

Pf. Kruskal's algorithm is a special case of the greedy MST algorithm.

- Suppose Kruskal's algorithm colors the edge $e = v\text{–}w$ black.
- Cut = set of vertices connected to v in tree T.
- No crossing edge is black.
- No crossing edge has lower weight. Why?

Kruskal's algorithm: implementation challenge

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

How difficult to implement?

A. $E + V$

B. V

C. $\log V$

D. $\log^* V$

E. 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 connected components

add edge e 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	$\log^* V$ †

often called fewer than E times

† amortized bound using weighted quick union with path compression

http://algs4.cs.princeton.edu

4.3 Minimum Spanning Trees

- *introduction*
- *greedy algorithm*
- *edge-weighted graph API*
- *Kruskal's algorithm*
- *Prim's algorithm*
- *context*

Prim's algorithm demo

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V - 1$ edges.

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

Prim's algorithm: proof of correctness

Proposition. [Jarník 1930, Dijkstra 1957, Prim 1959]
Prim's algorithm computes the MST.

Pf. Prim's algorithm is a special case of the greedy MST algorithm.

- Suppose edge e = min weight edge connecting a vertex on the tree to a vertex not on the tree.
- Cut = set of vertices connected on tree.
- No crossing edge is black.
- No crossing edge has lower weight.

Prim's algorithm: implementation challenge

Challenge. Find the min weight edge with exactly one endpoint in T.

How difficult?

A. E

B. V

C. $\log E$

D. 1

E. *I don't know.*

1-7 is min weight edge with exactly one endpoint in T

```
1-7 0.19
0-2 0.26
5-7 0.28
2-7 0.34
4-7 0.37
0-4 0.38
6-0 0.58
```

Prim's algorithm: lazy implementation

Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge $e = v–w$ to add to T.
- Disregard if both endpoints v and w are marked (both in T).
- Otherwise, let w be the unmarked vertex (not in T):
  - add e to T and mark w
  - add to PQ any edge incident to w (assuming other endpoint not in T)

Prim's algorithm: lazy implementation demo

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V - 1$ edges.

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

Prim's algorithm: lazy implementation

```
public class LazyPrimMST
{
   private boolean[] marked;   // MST vertices
   private Queue<Edge> mst;    // MST edges
   private MinPQ<Edge> pq;     // PQ of edges

    public LazyPrimMST(WeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        visit(G, 0);

        while (!pq.isEmpty() && mst.size() < G.V() - 1)
        {
           Edge e = pq.delMin();
           int v = e.either(), w = e.other(v);
           if (marked[v] && marked[w]) continue;
           mst.enqueue(e);
           if (!marked[v]) visit(G, v);
           if (!marked[w]) visit(G, w);
        }
   }
}
```

assume G is connected

repeatedly delete the min weight edge e = v–w from PQ

ignore if both endpoints in T

add edge e to tree

add either v or w to tree

Prim's algorithm: lazy implementation

```
private void visit(WeightedGraph G, int v)
{
   marked[v] = true;
   for (Edge e : G.adj(v))
      if (!marked[e.other(v)])
         pq.insert(e);
}

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

add v to T

for each edge e = v–w, add to PQ if w not already in T

Lazy Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

minor defect

Pf.

operation	frequency	binary heap
delete min	E	$\log E$
insert	E	$\log E$

Prim's algorithm: eager implementation

Challenge. Find min weight edge with exactly one endpoint in T.

Observation. For each vertex v, need only lightest edge connecting v to T.

- MST includes at most one edge connecting v to T. Why?
- If MST includes such an edge, it must take lightest such edge. Why?

Prim's algorithm: eager implementation demo

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V - 1$ edges.

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

Prim's algorithm: eager implementation demo

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V - 1$ edges.

v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-5	0.35
6	6-2	0.40

MST edges

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

Prim's algorithm: eager implementation

Challenge. Find min weight edge with exactly one endpoint in T.

PQ has at most one entry per vertex

Eager solution. Maintain a PQ of vertices connected by an edge to T, where priority of vertex v = weight of lightest edge connecting v to T.

- Delete min vertex v and add its associated edge $e = v–w$ to T.
- Update PQ by considering all edges $e = v–x$ incident to v
  - ignore if x is already in T
  - add x to PQ if not already on it
  - decrease priority of x if $v–x$ becomes lightest edge connecting x to T

0		
1	1-7	0.19
2	0-2	0.26
3	1-3	0.29
4	0-4	0.38
5	5-7	0.28
6	6-0	0.58
7	0-7	0.16

red: on PQ

black: on MST

ALGORITHM 4.7 Prim's MST algorithm (eager version)

```
public class PrimMST
{
   private Edge[] edgeTo;          // shortest edge from tree vertex
   private double[] distTo;        // distTo[w] = edgeTo[w].weight()
   private boolean[] marked;       // true if v on tree
   private IndexMinPQ<Double> pq;  // eligible crossing edges

   public PrimMST(EdgeWeightedGraph G)
   {
      edgeTo = new Edge[G.V()];
      distTo = new double[G.V()];
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      pq = new IndexMinPQ<Double>(G.V());

      distTo[0] = 0.0;
      pq.insert(0, 0.0);              // Initialize pq with 0, weight 0.
      while (!pq.isEmpty())
         visit(G, pq.delMin());       // Add closest vertex to tree.
   }
```

```
private void visit(EdgeWeightedGraph G, int v)
{  // Add v to tree; update data structures.
   marked[v] = true;
   for (Edge e : G.adj(v))
   {
      int w = e.other(v);
      if (marked[w]) continue;     // v-w is ineligible.
      if (e.weight() < distTo[w])
      {  // Edge e is new best connection from tree to w.
         edgeTo[w] = e;
         distTo[w] = e.weight();
         if (pq.contains(w)) pq.change(w, distTo[w]);
         else                pq.insert(w, distTo[w]);
      }
   }
}
```

Prim'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.

http://algs4.cs.princeton.edu

4.4 Shortest Paths

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

Last updated on Apr 1, 2015, 7:10 AM

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
```

0.26 + 0.34 + 0.39 + 0.52 = 1.51

Google maps

Map
Satellite
Hybrid
Medical Center At Princeton
Princeton Cemetery
Princeton
Princeton University-Main Campus
Palmer Stadium-Princeton University
Springdale Golf Club
Nassau St
Washington Rd
Harrison St N
Bayard Ln
Mountain Ave
Elm Rd
Stockton St
206
27
1000 ft
200 m
©2005 Google - Map data ©2005 NAVTEQ™ - Terms of Use

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. Each vertex is reachable from s.

http://algs4.cs.princeton.edu

4.4 Shortest Paths

- *APIs*
- *shortest-paths properties*
- *Dijkstra's algorithm*
- *edge-weighted DAGs*
- *negative weights*

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 adjacent 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
    15 ← E
    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?*

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]`.

v→w successfully relaxes

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

```
distTo[v1]   ≤  distTo[v0]   + e1.weight()
distTo[v2]   ≤  distTo[v1]   + e2.weight()
...
distTo[vk]   ≤  distTo[vk-1] + ek.weight()
```

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

- Add inequalities; simplify; and substitute `distTo[v0]` = `distTo[s]` = 0:

$$\texttt{distTo[w]} = \texttt{distTo[v}_k\texttt{]} \le \underbrace{\texttt{e}_1\texttt{.weight()} + \texttt{e}_2\texttt{.weight()} + \ldots + \texttt{e}_k\texttt{.weight()}}_{\text{weight of shortest path from s to w}}$$

weight of some path from s to w

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

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 adjacent from that vertex.

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

an edge-weighted digraph

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 adjacent from that vertex.

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Dijkstra's algorithm: correctness proof 1

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Each edge `e` = `v`→`w` is relaxed exactly once (when vertex `v` is relaxed), leaving `distTo[w]` ≤ `distTo[v] + e.weight()`.
- Inequality holds until algorithm terminates because:
  - `distTo[w]` cannot increase ← `distTo[]` values are monotone decreasing
  - `distTo[v]` will not change ← we choose lowest `distTo[]` value at each step (and edge weights are nonnegative)

when relaxing v

if u has not yet been relaxed,

then distTo[u] ≥ distTo[v]

- Thus, upon termination, shortest-paths optimality conditions hold. ■

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

Shortest paths: quiz 2

What is the order of growth of the running time of Dijkstra's algorithm when using a binary heap for the priority queue?

A. $V + E$

B. $V \log E$

C. $E \log V$

D. $E \log E$

E. *I don't know.*

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.

Computing a spanning tree in a graph

Dijkstra's algorithm seem familiar?

- Prim's algorithm is essentially the same algorithm.
- Both are in a family of algorithms that compute a spanning tree.

Main distinction: rule used to choose next vertex for the tree.

- Prim: Closest vertex to the tree (via an undirected edge).
- Dijkstra: Closest vertex to the source (via a directed path).

Note: DFS and BFS are also in this family of algorithms.