4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
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.
Protein-protein interaction network

Reference: Jeong et al, Nature Review | Genetics
Framingham heart study

Each circle (node) represents one person in the data set. There are 2200 persons in this subcomponent of the social network. Circles with red borders denote women, and circles with blue borders denote men. The size of each circle is proportional to the person’s body-mass index. The interior color of the circles indicates the person’s obesity status: yellow denotes an obese person (body-mass index, ≥30) and green denotes a nonobese person. The colors of the ties between the nodes indicate the relationship between them: purple denotes a friendship or marital tie and orange denotes a familial tie.
The evolution of FCC lobbying coalitions

“The Evolution of FCC Lobbying Coalitions” by Pierre de Vries in JoSS Visualization Symposium 2010
Map of science clickstreams

http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803
10 million Facebook friends

"Visualizing Friendships" by Paul Butler
The Internet as mapped by the Opte Project

http://en.wikipedia.org/wiki/Internet
# Graph applications

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

<table>
<thead>
<tr>
<th>problem</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s–t path</strong></td>
<td><em>Is there a path between s and t?</em></td>
</tr>
<tr>
<td><strong>shortest s–t path</strong></td>
<td><em>What is the shortest path between s and t?</em></td>
</tr>
<tr>
<td><strong>cycle</strong></td>
<td><em>Is there a cycle in the graph?</em></td>
</tr>
<tr>
<td><strong>Euler cycle</strong></td>
<td><em>Is there a cycle that uses each edge exactly once?</em></td>
</tr>
<tr>
<td><strong>Hamilton cycle</strong></td>
<td><em>Is there a cycle that uses each vertex exactly once?</em></td>
</tr>
<tr>
<td><strong>connectivity</strong></td>
<td><em>Is there a path between every pair of vertices?</em></td>
</tr>
<tr>
<td><strong>biconnectivity</strong></td>
<td><em>Is there a vertex whose removal disconnects the graph?</em></td>
</tr>
<tr>
<td><strong>planarity</strong></td>
<td><em>Can the graph be drawn in the plane with no crossing edges?</em></td>
</tr>
<tr>
<td><strong>graph isomorphism</strong></td>
<td><em>Are two graphs isomorphic?</em></td>
</tr>
</tbody>
</table>

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

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.

- Not all graph representations can handle these.
**Graph API**

```java
public class Graph {
    public Graph(int V) {
        // create an empty graph with V vertices
    }

    public Graph(In in) {
        // create a graph from input stream
    }

    public void addEdge(int v, int w) {
        // add an edge v-w
    }

    public Iterable<Integer> adj(int v) {
        // vertices adjacent to v
    }

    public int V() {
        // number of vertices
    }

    public int E() {
        // number of edges
    }
}
```

```java
// 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: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}.$
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?

```java
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. $VE$
E. I don't know.
Graph representation: adjacency lists

Maintain vertex-indexed array of lists.
Which is order of growth of running time of the following code fragment if the graph uses the \textit{adjacency-lists} representation, where $V$ is the number of vertices and $E$ is the number of edges?

\begin{verbatim}
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);
\end{verbatim}

prints each edge exactly once

\begin{itemize}
    \item[A.] $V$
    \item[B.] $E + V$
    \item[C.] $V^2$
    \item[D.] $VE$
    \item[E.] I don't know.
\end{itemize}
In practice. Use adjacency-lists representation.

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

Two graphs (\( V = 50 \))

sparse (\( E = 200 \))

dense (\( E = 1000 \))

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**.

![Graph representations table](image)

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>add edge</th>
<th>edge between v and w?</th>
<th>iterate over vertices adjacent to v?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>( E )</td>
<td>1</td>
<td>( E )</td>
<td>( E )</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>( V^2 )</td>
<td>1( \dagger )</td>
<td>1</td>
<td>( V )</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>( E + V )</td>
<td>1</td>
<td>( \text{degree}(v) )</td>
<td>( \text{degree}(v) )</td>
</tr>
</tbody>
</table>

\( \dagger \) disallows parallel edges

**Real-world graphs tend to be sparse.**
- Huge number of vertices, small average vertex degree

![Sparse graphs](image)
Adjacency-list graph representation: Java implementation

```java
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
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.
Maze exploration: National Building Museum

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 $v$. 

---

**graph G**

---

Input format for Graph constructor (two examples)

```
250 1273
244 246
239 240
238 245
235 238
233 240
232 248
231 248
229 249
228 241
226 231
... 
```

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

tinyG.txt

```
V
E
13
13
```

mediumG.txt
Depth-first search demo

To visit a vertex $v$:

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

vertices reachable from 0

<table>
<thead>
<tr>
<th>$v$</th>
<th>marked[]</th>
<th>edgeTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>9</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>
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.

```java
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 $\text{marked}[]$ to mark vertices.
- Integer array $\text{edgeTo}[]$ to keep track of paths.
  $(\text{edgeTo}[w] = v)$ means that edge $v$–$w$ taken to discover vertex $w$
- Function-call stack for recursion.
Depth-first search: Java implementation

```java
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.
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 $s$.

```java
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;
}
```
Problem. Implement flood fill (Photoshop magic wand).
**Challenge.** Implement DFS without recursion.

**One solution.** [see http://algs4.cs.princeton.edu/41undirected/NonrecursiveDFS.java.html]

- Maintain a stack of vertices, initialized with $s$.
- For each vertex, maintain a pointer to current vertex in adjacency list.
- Pop next vertex $v$ off the stack:
  - let $w$ be next unmarked vertex in adjacency list of $v$
  - push $w$ onto stack and mark it

**Alternative solution.** space proportional to $E + V$ (vertex can appear on stack more than once)

- Maintain a stack of vertices, initialized with $s$.
- Pop next vertex $v$ off the stack:
  - if vertex $v$ is marked, continue
  - mark $v$ and add to stack each of its unmarked neighbors
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
**Graph search.** Many ways to explore every vertex in a graph.

- Preorder: vertices in order DFS calls $\text{dfs}(G, v)$.
- Postorder: vertices in order DFS returns from $\text{dfs}(G, v)$.
- Level-order: vertices in increasing order of distance from $s$. 

---

**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: $S\ E\ X\ A\ R\ C\ H\ M$
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.

Graph \(G\) and its adjacency lists:

\[\text{tinyCG.txt}\]

\[V \rightarrow 6\]
\[6 \rightarrow \begin{align*}
0 & \rightarrow 5 \\
0 & \rightarrow 2 \\
2 & \rightarrow 4 \\
2 & \rightarrow 3 \\
1 & \rightarrow 2 \\
0 & \rightarrow 1 \\
3 & \rightarrow 4 \\
3 & \rightarrow 5 \\
0 & \rightarrow 2
\end{align*}\]
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.

```
<table>
<thead>
<tr>
<th>v</th>
<th>edgeTo[]</th>
<th>distTo[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
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 unmarked neighbors to the queue, and mark them.
Breadth-first search: Java implementation

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

    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 $\geq 0$ vertices of distance $k$ from $s$, followed by $\geq 0$ 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**

- $s$
- 0
- 1
- 2
- 3
- 4
- 5

**dist = 0**

- 0
- 2
- 5

**dist = 1**

- 1
- 3

**dist = 2**

- 4
- 3
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 $s = \text{Kevin Bacon}$. 

![Kevin Bacon graph diagram]
Breadth-first search application: Erdős numbers

hand-drawing of part of the Erdős graph by Ron Graham
Erdős-Bacon-Sabbath

Lisa Kudrow  E10 + B2 + S3
Sergey Nikitin & Tatyana Nikitina  E7 + B3 + S4
Jackie Fox  E6 + B3 + S2
Ray Kurzweil  E4 + B2 + S2
Danica McKellar  E4 + B2 + S4
Condoleezza Rice  E6 + B3 + S4
Greg Graffin  E5 + B3 + S3
Richard Vranch  E5 + B2 + S2
Colin Firth  E6 + B1 + S4
Daniel Levitin  E4 + B2 + S2
David Grinspoon  E6 + B3 + S4
Warwick Holt  E5 + B2 + S6
Mayim Bialik  E4 + B2 + S4
David Morgan-Mar  E4 + B5 + S7
Imogen Heap  E8 + B4 + S3
Thomas Edison  E6 + B5 + S6
James Randi  E6 + B2 + S2
Phil Plait  E4 + B3 + S6
Adam Savage  E6 + B2 + S5
Terry Pratchett  E4 + B2 + S3
Lawrence Krauss  E4 + B3 + S6
Patrick Moore  E5 + B3 + S4
Simon Singh  E4 + B2 + S4
Karl Schaffer  E3 + B2 + S6
Buzz Aldrin  E6 + B2 + S3
Brian Cox  E7 + B3 + S2
Tom Lehrer  E4 + B2 + S3
Geoffrey Pullum  E3 + B3 + S4
Fred Rogers  E9 + B2 + S6
Jonathan Feinberg  E5 + B3 + S3
Albert Einstein  E2 + B4 + S5
Carl Sagan  E4 + B2 + S4
Noam Chomsky  E4 + B3 + S4
Stephen Hawking  E4 + B2 + S2
Richard Feynman  E3 + B3 + S4
Natalie Portman  E5 + B2 + S3
Thomas Halliday  E5 + B3 + S3
Adam Rutherford  E6 + B3 + S6
Jeff Baxter  E6 + B2 + S2
Douglas Adams  E10 + B2 + S2
Woody Paul  E7 + B2 + S4
Milo Aukerman  E7 + B3 + S3
Brian May  E5 + B3 + S1

erdosbaconsabbath.com
4.1 Undirected Graphs

- introduction
- graph API
- depth-first search
- breadth-first search
- challenges
Problem. Identify connected components.

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 2

Problem. Is a graph bipartite?

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, 3, 4 \}
Graph-processing challenge 3

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

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?

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 1-2 2-3 2-4 3-4 4-5 4-6
0-1-2-3-4-2-0-6-4-5-0
Graph-processing challenge 5

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

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?

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

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

try it yourself at http://planarity.net
Graph traversal summary

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

<table>
<thead>
<tr>
<th>graph problem</th>
<th>BFS</th>
<th>DFS</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>s–t path</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>shortest s–t path</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>cycle</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>Euler cycle</td>
<td></td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>Hamilton cycle</td>
<td></td>
<td></td>
<td>$2^{1.657V}$</td>
</tr>
<tr>
<td>bipartiteness (odd cycle)</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>connected components</td>
<td>✔</td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>biconnected components</td>
<td></td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>planarity</td>
<td></td>
<td>✔</td>
<td>$E + V$</td>
</tr>
<tr>
<td>graph isomorphism</td>
<td></td>
<td></td>
<td>$2^{c\sqrt{V\log V}}$</td>
</tr>
</tbody>
</table>