9 Graph Algorithms

9 December 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-12-09 16:01`

Learning Outcomes

Unit 9: ***Graph Algorithms***

1. Know basic terminology from graph theory, including types of graphs.
2. Know adjacency matrix and adjacency list representations and their performance characteristica.
3. Know graph-traversal based algorithm, including efficient implementations.
4. Be able to proof correctness of graph-traversal-based algorithms.
5. Know algorithms for maximum flows in networks.
6. Be able to model new algorithmic problems as graph problems.

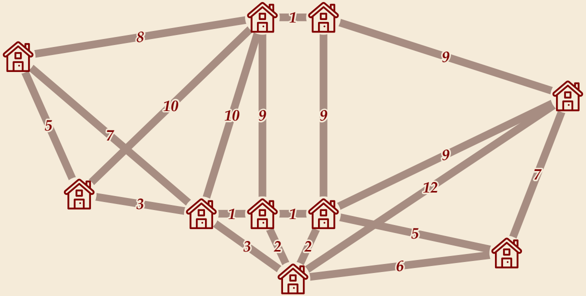
Outline

9 Graph Algorithms

9.1 Introduction & Definitions

Graphs in real life

- a graph is an abstraction of *entities* with their (pairwise) *relationships*
- abundant examples in real life (often called network there)
  - social networks: e. g. persons and their friendships, . . . *Five/Six? degrees of separation*
  - physical networks: cities and highways, roads networks, power grids etc., the Internet, . . .
  - content networks: world wide web, ontologies, . . .
  - . . .



Many More examples, e. g., in Sedgewick & Wayne's videos:

`https://www.coursera.org/learn/algorithms-part2`

Flavors of Graphs

- Since graphs are used to model so many different entities and relations, they come in several variants

Property	Yes	No
edges are one-way	*directed* graph (*digraph*)	*undirected* graph
≤ 1 edge between u and v	*simple* graph	*multigraph* / with *parallel* edges
edges can lead from v to v	with *loops* (Schlinge, Schleife)	(loop-free)
edges have weights	*(edge-) weighted* graph	*unweighted* graph

🙃 any combination of the above can make sense ...

- Synonyms:
  - **vertex** („Knoten“) = node = point = „Ecke”
  - **edge** („Kante“) = arc = line = relation = arrow = „Pfeil“
  - **graph** = network

Graph Theory

- default: unweighted, undirected, loop-free & simple graphs
- *Graph* $G = (V, E)$ with
  - V a finite of *vertices*
  - $E \subseteq [V]^2$ a set of *edges*, which are 2-subsets of V: $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

$e = \{u, v\}$

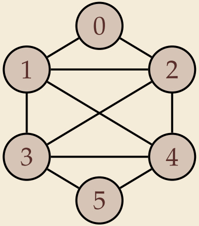
Graph Theory

- default: unweighted, undirected, loop-free & simple graphs
- *Graph* $G = (V, E)$ with
  - V a finite of *vertices*
  - $E \subseteq [V]^2$ a set of *edges*, which are 2-subsets of V: $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

Example

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

Graphical representation



like so . . .

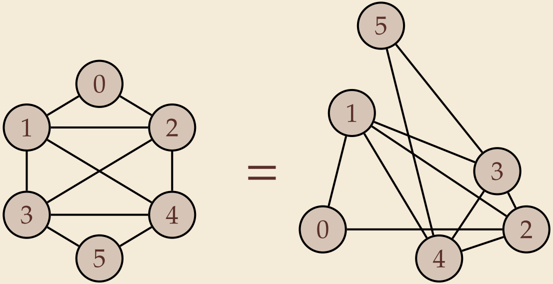
Graph Theory

- default: unweighted, undirected, loop-free & simple graphs
- *Graph* $G = (V, E)$ with
  - V a finite of *vertices*
  - $E \subseteq [V]^2$ a set of *edges*, which are 2-subsets of V: $[V]^2 = \{e : e \subseteq V \wedge |e| = 2\}$

Example

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

Graphical representation



like so . . .   . . . or so

(same graph)

Digraphs

- default digraph: unweighted, loop-free & simple
- *Digraph (directed graph)* $G = (V, E)$ with
  - V a finite of *vertices*
  - $E \subseteq V^2 \setminus \{(v, v) : v \in V\}$ a set of *(directed) edges*,
    $V^2 = V \times V = \{(x, y) : x \in V \wedge y \in V\}$ 2-tuples / ordered pairs over V

$x \longrightarrow y$

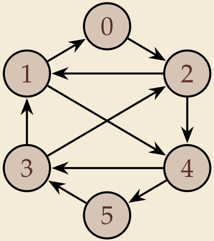
Digraphs

- default digraph: unweighted, loop-free & simple
- *Digraph (directed graph)* $G = (V, E)$ with
  - V a finite of *vertices*
  - $E \subseteq V^2 \setminus \{(v, v) : v \in V\}$ a set of *(directed) edges*,
    $V^2 = V \times V = \{(x, y) \ : \ x \in V \wedge y \in V\}$ 2-tuples / ordered pairs over V

Example

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

Graphical representation



Graph Terminology

Undirected Graphs

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

Directed Graphs (where different)

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

Graph Terminology

Undirected Graphs	**Directed Graphs** (where different)
▸ $V(G)$ set of vertices, $E(G)$ set of edges	
▸ write uv (or vu) for edge $\{u, v\}$	▸ uv for (u, v)
▸ edges *incident* at vertex v: $E(v)$	
▸ u and v are *adjacent* iff $\{u, v\} \in E$,	▸ iff $(u, v) \in E \vee (v, u) \in E$
▸ *neighborhood* $N(v) = \{w \in V : w \text{ adjacent to } v\}$	▸ in-/out-neighbors $N_{\text{in}}(v)$, $N_{\text{out}}(v)$
▸ *degree* $d(v) = \lvert E(v) \rvert$	▸ in-/out-degree $d_{\text{in}}(v)$, $d_{\text{out}}(v)$

Kantenzug

- *walk* w of length n: sequence of vertices $w[0..n]$ with $\forall i \in [0..n) : w[i]w[i+1] \in E$
- *path* p is a (vertex-) simple walk: without duplicate vertices except possibly its endpoints
- *edge-simple* walk/path: no edge used twice
- *cycle* c is a closed path, i. e., $c[0] = c[n]$ (geschlossener Weg, Zykel, Kreis, Zyklus)

Graph Terminology

Undirected Graphs

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

Directed Graphs (where different)

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

- *walk* w of length n: sequence of vertices $w[0..n]$ with $\forall i \in [0..n) : w[i]w[i+1] \in E$
- *path* p is a (vertex-) simple walk: without duplicate vertices except possibly its endpoints
- *edge-simple* walk/path: no edge used twice
- *cycle* c is a closed path, i. e., $c[0] = c[n]$

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

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

Typical graph-processing problems

- **Path**: Is there a path between s and t?
  Shortest path: What is the shortest path (distance) between s and t?
- **Cycle**: Is there a cycle in the graph?
  Euler tour: Is there a cycle that uses each edge exactly once?
  Hamilton(ian) cycle: Is there a cycle that uses each vertex exactly once.
- **Connectivity**: Is there a way to connect all of the vertices?
  MST: What is the best way to connect all of the vertices?
  Biconnectivity: Is there a vertex whose removal disconnects the graph?
- **Planarity**: Can you draw the graph in the plane with no crossing edges?
- **Graph isomorphism**: Are two graphs the same up to renaming vertices?

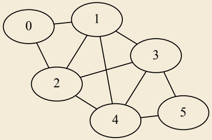
can vary a lot, despite superficial similarity of problems

Challenge: Which of these problems
can be computed in (near) linear time?
in reasonable polynomial time?
are intractable?

Tools to work with graphs

- Convenient GUI to edit & draw graphs: *yEd live*
  `yworks.com/yed-live`
- *graphviz* cmdline utility to draw graphs
  - Simple text format for graphs: `DOT`

```
graph G {
     0 -- 2;     2 -- 4;
     1 -- 0;     2 -- 3;
     1 -- 4;     3 -- 4;
     1 -- 3;     3 -- 5;
     2 -- 1;     4 -- 5;
}
```



`dot -Tpdf graph.dot -Kfdp > graph.pdf`

- graphs are typically not built into programming languages, but libraries exist
  - e. g. part of *Google Guava* for Java
  - they usually allow arbitrary objects as vertices
  - aimed at ease of use

9.2 Graph Representations

Graphs in Computer Memory

- We defined graphs in set-theoretic terms. . .
  but computers can't directly deal with sets efficiently

$\rightsquigarrow$ need to choose a *representation* for graphs.

- which is better depends on the required operations

Graphs in Computer Memory

- We defined graphs in set-theoretic terms...
  but computers can't directly deal with sets efficiently

$\rightsquigarrow$ need to choose a *representation* for graphs.

  - which is better depends on the required operations

Key Operations:

  - `isAdjacent(`u`,`v`)`
    Test whether $uv \in E$
  - `adj(`v`)`
    Adjacency list of v (iterate through (out-) neighbors of v)
  - most others can be computed based on these

Graphs in Computer Memory

- We defined graphs in set-theoretic terms...
  but computers can't directly deal with sets efficiently

$\rightsquigarrow$ need to choose a *representation* for graphs.

- which is better depends on the required operations

Key Operations:

- `isAdjacent(`u`,`v`)`
  Test whether $uv \in E$
- `adj(`v`)`
  Adjacency list of v (iterate through (out-) neighbors of v)
- most others can be computed based on these

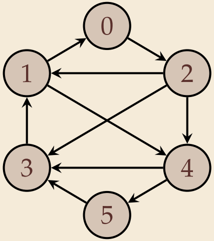
Conventions:

- (di)graph $G = (V, E)$ (omitted if clear from context)
- $n = |V|$, $m = |E|$
- in implementations assume $\boxed{V = [0..n)}$ (if needed, use symbol table to map complex objects to V)

Adjacency Matrix Representation

- adjacency matrix $A \in \{0,1\}^{n \times n}$ of G: matrix with $A[u,v] = [uv \in E]$ $= \begin{cases} 1 & uv \in E \\ 0 & \text{sonst} \end{cases}$
  - works for both directed and undirected graphs (undirected $\rightsquigarrow A = A^T$ symmetric)
  - can use a weight $w(uv)$ or multiplicity in $A[u,v]$ instead of $0/1$
  - can represent loops via $A[v,v]$

Example:

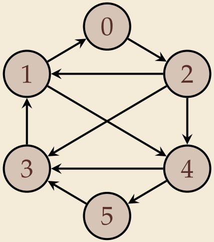


$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Adjacency Matrix Representation

- adjacency matrix $A \in \{0,1\}^{n \times n}$ of G: matrix with $A[u,v] = [uv \in E]$
  - works for both directed and undirected graphs (undirected $\rightsquigarrow A = A^T$ symmetric)
  - can use a weight $w(uv)$ or multiplicity in $A[u,v]$ instead of $0/1$
  - can represent loops via $A[v,v]$

Example:



$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

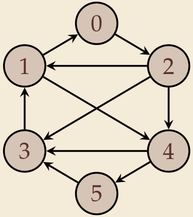
`isAdjacent` in $O(1)$ time

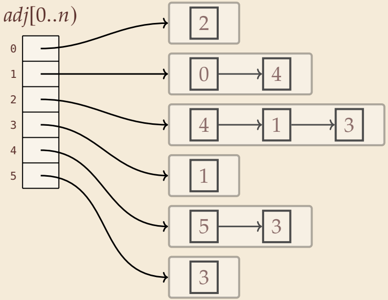
$O(n^2)$ (bits of) space wasteful for sparse graphs

`adj(`v`)` iteration takes $O(n)$ (independent of $d(v)$)

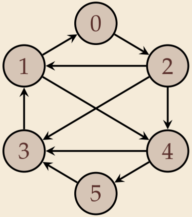
Adjacency List Representation

- Store a linked list of neighbors for each vertex v:
  - $adj[0..n)$ bag of neighbors (as linked list)
  - undirected edge $\{u, v\}$ $\rightsquigarrow$ v in $adj[u]$ and u in $adj[v]$
  - weighted edge uv $\rightsquigarrow$ store pair $(v, w(uv))$ in $adj[u]$
  - multiple edges and loops can be represented

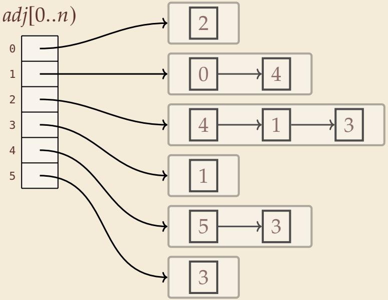




Adjacency List Representation



- Store a linked list of neighbors for each vertex v:
  - $adj[0..n)$ bag of neighbors (as linked list)
  - undirected edge $\{u, v\}$ $\rightsquigarrow$ v in $adj[u]$ and u in $adj[v]$
  - weighted edge uv $\rightsquigarrow$ store pair $(v, w(uv))$ in $adj[u]$
  - multiple edges and loops can be represented



👎 `isAdjacent(`u`,`v`)` takes $\Theta(d(u))$ time (worst case)

👍 `adj(`v`)` iteration $O(1)$ per neighbor

👍 $\Theta(n + m)$ (words of) space for any graph ($\ll \Theta(n^2)$ bits for moderate m)

$\rightsquigarrow$ de-facto standard for graph algorithms

Graph Types and Representations

- Note that adj matrix and lists for undirected graphs effectively are representation of directed graph with directed edges both ways
  - conceptually still important to distinguish!
- multigraphs, loops, edge weights all naturally supported in adj lists
  - good if we allow and use them
  - but requires explicit checks to enforce simple / loopfree / bidirectional!

- we focus on **static graphs**
  dynamically changing graphs much harder to handle