9 Graph Algorithms

9 December 2024

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2024-12-08 22:44` *H*

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.

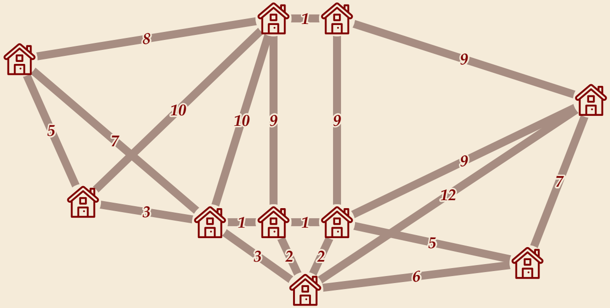
Outline

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*	(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

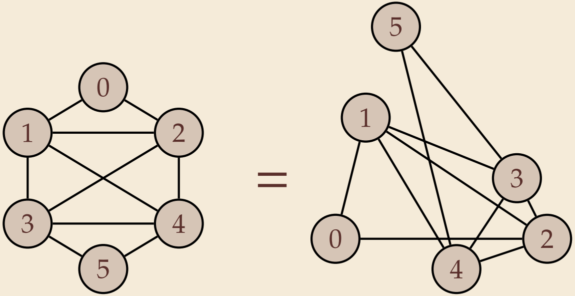
Graph Theory

- 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: $[V]^2 = \{e : e \subseteq V \land |e| = 2\}$

Example

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

Graphical representation



like so . . .

. . . or so

(same graph)

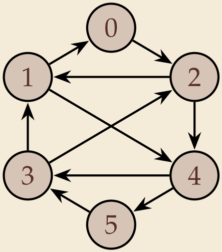
Digraphs

- 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

Example

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

Graphical representation



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 $N_{\mathrm{in}}(v)$, $N_{\mathrm{out}}(v)$
- in-/out-degree $d_{\mathrm{in}}(v)$, $d_{\mathrm{out}}(v)$

- *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
- *strongly connected* for digraphs (*weakly connected* = connected ignoring directions)
- G is *acyclic* iff $\nexists$ cycle (of length $n \geq 1$) in G

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?

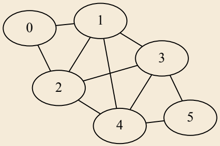
can vary a lot, despite superficial similarity of problems

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

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

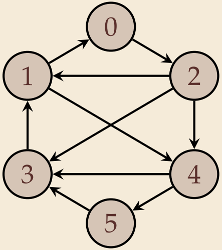
Conventions:

- (di)graph $G = (V, E)$ (omitted if clear from context)
- $n = |V|$, $m = |E|$
- in implementations assume $V = [0..n)$ (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 $A[u,v] = [uv \in E]$
  - 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:



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

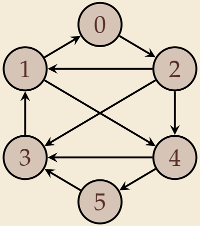
👍 `isAdjacent` in $O(1)$ time

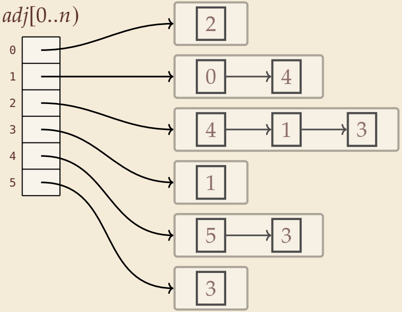
👎 $O(n^2)$ (bits of) space wasteful for sparse graphs

👎 `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





`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

9.3 Graph Traversal

Generic Graph Traversal

- Plethora of graph algorithms can be expressed as a systematic exploration of a graph

  visiting all edges

  - depth-first search, breadth-first search
  - cycle finding
  - topological sorting
  - Hierholzer's algorithm for Euler cycles
  - connected components
  - strong components
  - testing bipartiteness
  - Dijkstra's algorithm
  - Prim's algorithm
  - Lex-BFS for perfect elimination orders of chordal graphs
  - ...

$\rightsquigarrow$ Formulate generic traversal algorithm

  - first in abstract terms to argue about correctness
  - then again for concrete instance with efficient data structures

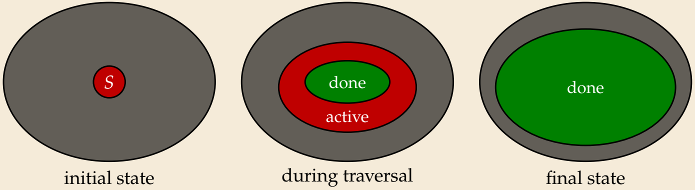
Tricolor Graph Traversal

Tricolor Graph Search:

- maintain vertices in 3 (dynamic) sets
  - **Gray: unseen vertices**
    The traversal has not reached these vertices so far.
  - **Green: done vertices** (a. k. a. visited vertices)
    These vertices have been visited and all their edges have been explored already.
  - **Red: active vertices** (a. k. a. frontier („Rand“) of traversal)
    All others, i. e., vertices that have been reached and some unexplored edges remain; initially some selected start vertices S.

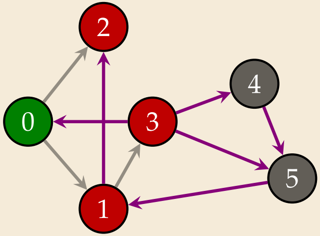
Invariant:
No edges from *done* to *unseen* vertices

- (implicitly) maintain status of each edge
  - **not yet used**
  - **used edge**



Generic Tricolor Graph Traversal – Code

```
procedure genericGraphTraversal(G, S)
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // Color array, all cells initialized to unseen
    for s ∈ S do C[s] := active end for
    unusedEdges := E
    while ∃v : C[v] == active
        v := nextActiveVertex() // Freedom 1: Which frontier vertex?
        if ∄vw ∈ unusedEdges // no more edges from v ⇝ done with v done
            C[v] := done
        else
            w := nextUnusedEdge(v) // Freedom 2: Which of its edges?
            if C[w] == unseen
                C[w] := active
            end if
            unusedEdges.remove(vw)
        end if
    end while
```



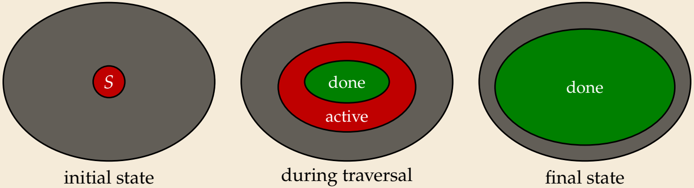
Invariant:
No edges from *done* to *unseen* vertices

- Implementations of nextActiveVertex() and nextUnusedEdge(v) depends on (and defines!) specific traversal-based graph algorithms

Generic Reachability

- Any choices nextActiveVertex() and nextUnusedEdge(v) suffice to find exactly the vertices reachable from S in *done*

- **Invariant:**
  1. No edges from *done* to *unseen* vertices
  2. For every *done* vertex v, there exists a path from $s \in S$ to v.



$\rightsquigarrow$ in final state:

- $v \in done$ $\rightsquigarrow$ path from S $\rightsquigarrow$ reachable from S
- $v \in unseen$ $\rightsquigarrow$ not reachable from $done \supseteq S$ $\rightsquigarrow$ not reachable from S

Data Structures for Frontier

- We need efficient support for
  - test $\exists v : C[v] = \mathit{active}$, nextActiveVertex()
  - test $\exists vw \in \mathit{unusedEdges}$, nextUnusedEdge($v$)
  - $\mathit{unusedEdges}$.remove(vw)
- Typical solution maintains **bag** *frontier* of *pairs* (v, i) where $v \in V$ and i is an **iterator** in `adj`$[v]$
  - $\mathit{unusedEdges}$ represented implicitly: edge used iff previously returned by i

    $\rightsquigarrow$ don't need $\mathit{unusedEdges}$.remove(vw)
  - Implement $\exists v : C[v] = \mathit{active}$ via *frontier*.isEmpty()
  - Implement $\exists vw \in \mathit{unusedEdges}$ via i.hasNext() assuming $(v, i) \in \mathit{frontier}$
  - Implement nextUnusedEdge(v) via i.next() assuming $(v, i) \in \mathit{frontier}$

$\rightsquigarrow$ all operations apart from nextActiveVertex() in $O(1)$ time

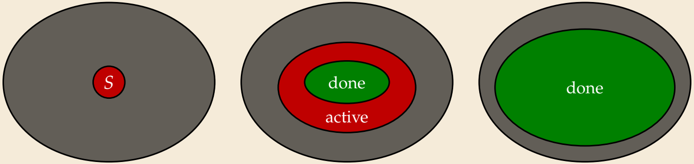
$\rightsquigarrow$ *frontier* requires $O(n)$ extra space

Breadth-First Search

- Maintain *frontier* in a **queue** (FIFO: first in, first out)

- **Invariant:**
  1. No edges from done to unseen vertices

     fewest edges
  2. All *done* vertices are reached via a **shortest path** from S
  3. *frontier* stores active vertices **sorted** by distance from S



initial state | during traversal | final state

⇝ in final state, we reach all reachable vertices via shortest paths

- To preserve that knowledge, we collect extra information during traversal
  - *parent*$[v]$ stores predecessor on path from S via which v was reached
  - *distFromS*$[v]$ stores the length of this path

Breadth-First Search – Code

```
procedure bfs(G, S)
    // (di)graph G = (V, E) and start vertices S ⊆ V
    C[0..n) := unseen // New array initialized to all unseen
    frontier := new Queue;
    parent[0..n) := NOT_VISITED;  distFromS[0..n) := ∞
    for s ∈ S
        parent[s] := NONE;  distFromS[s] := 0
        C[s] := active;  frontier.enqueue((s, G.adj[s].iterator()))
    end for
    while ¬frontier.isEmpty()
        (v, i) := frontier.peek()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.dequeue()
        else
            w := i.next() // Advance i in adj[v]
            if C[w] == unseen
                parent[w] := v;  distFromS[w] := distFromS[v] + 1
                C[w] := active;  frontier.enqueue((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

- *parent* stores a *shortest-path tree/forest*
- can retrieve shortest path to v from some vertex $s \in S$ (backwards) by following *parent*$[v]$ iteratively
- running time $\Theta(n + m)$
- extra space $\Theta(n)$

Depth-First Search

- Maintain *frontier* in a **stack** (LIFO: last in, first out)

```
procedure dfs(G, s)
    // (di)graph G = (V, E) and start vertex s ∈ V
    C[0..n) := unseen;  frontier := new Stack;
    parent[0..n) := NOT_VISITED;
    parent[s] := NONE;
    C[s] := active;  frontier.push((s, G.adj[s].iterator()))
    while ¬frontier.isEmpty()
        (v, i) := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.pop();  postorderVisit(v)
        else
            w := i.next() // Advance i in adj[v]
            visitEdge(vw)
            if C[w] == unseen
                parent[w] := v;
                preorderVisit(w)
                C[w] := active;  frontier.push((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

- *parent* stores a DFS tree
- pre-/postorderVisit hooks to implement further operations
  - preorder: visit v when made *active*
  - postorder: visit v when marked *done*
  - visitEdge: do something for every edge
- running time $\Theta(n + m)$
- extra space $\Theta(n)$

Connected Components

- In an undirected graph, find all *connected components*.
  - **Given:** simple undirected $G = (V, E)$
  - **Goal:** assign component ids $CC[0..n)$, s.t. $CC[v] = CC[u]$ iff $\exists$ path from v to u

```
procedure connectedComponents(G):
    // undirected graph G = (V, E) with V = [0..n)
    C[0..n) := unseen
    CC[0..n) := NONE
    id := 0
    for v := 0, ..., n − 1
        if C[v] == unseen
            dfs(G, v)
            id := id + 1
        end if
    end for
```

```
procedure dfs(G, s)
    // Do not reinitialize C[0..n) to unseen but reuse global C
    frontier := new Stack;
    parent[s] := NONE;
    C[s] := active;  frontier.push((s, G.adj[s].iterator()))
    while ¬frontier.isEmpty()
        (v, i) := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.pop();  postorderVisit(v)
        else
            w := i.next() // Advance i in adj[v]
            visitEdge(vw)
            if C[w] == unseen
                parent[w] := v;
                preorderVisit(w)
                C[w] := active;
                frontier.push((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

- In preorderVisit(v):
  $CC[v] := id$

DFS Postorder & Topological Sort

- Example application of generic DFS: topological sort
  - $R[0..n)$ is topological order of G if $\forall (u,v) \in E : R[u] < R[v]$
- **Topological Sorting**
  - **Given:** simple digraph $G = (V, E)$
  - **Goal:** topological order of vertices $R[0..n)$
    or `NOT_POSSIBLE` (if G contains a directed cycle)
- **DFS Postorder**: The DFS postorder from $s \in V$ is a numbering $P[0..n)$ of V such that
  $P[v] = r$ iff exactly r vertices reached state *done* before v in `dfs`(s).

Lemma 9.1

directed acyclic graph

Let G be a simple, connected DAG and $R[0..n)$ a *reverse DFS postorder* of G, i. e., $R[v] = n - 1 - P[v]$ for a DFS postorder $P[0..n)$. Then R is a topological order of G. ◀

Invariant:

1. If $v \in$ *done* and $(v, w) \in E$ then $w \in$ *done* and $R[v] < R[w]$.

9.4 Advanced Graph-Traversal Algorithms

DFS Postorder Implementation

```
procedure dfs(G, s)
    // (di)graph G = (V, E) and start vertex s ∈ V
    C[0..n) := unseen;  frontier := new Stack;
    parent[0..n) := NOT_VISITED;
    parent[s] := NONE;
    C[s] := active;  frontier.push((s, G.adj[s].iterator()))
    while ¬frontier.isEmpty()
        (v, i) := frontier.top()
        if ¬i.hasNext() // v has no unused edge
            C[v] := done;  frontier.pop();  postorderVisit(v)
        else
            w := i.next() // Advance i in adj[v]
            visitEdge(vw)
            if C[w] == unseen
                parent[w] := v;
                preorderVisit(w)
                C[w] := active;  frontier.push((w, G.adj[w].iterator()))
            end if
        end if
    end while
```

- In postorderVisit(v):
  $P[v] := r; \quad r := r + 1$
- In visitEdge(vw):
  If $w \in \textit{frontier}$, found cycle
  - To check that efficiently, store which vertices are in stack (easy modification)

Dijkstra's Algorithm & Prim's Algorithm

- On edge-weighted, we can use the tricolor traversal with a *priority queue* for *frontier*
- Dijkstra's Algorithm for shortest paths from s in digraphs with weakly positive edge weights
  - priority of vertex v = length of shortest path known so far from s to v
- Prim's Algorithm for finding a minimum spanning tree
  - priority of vertex v = weight of cheapest edge connecting v to current tree

$\rightsquigarrow$ Detailed discussion in Unit 11

Euler Cycles

Strong Components

Kosaraju-Sharir’s Algorithm

9.5 Network flows

Networks and Flows

Reductions

9.6 The Ford-Fulkerson Method

Residual Networks

Augmenting Paths