Algorithms and Data Structures
Graphs, Depth-/Breadth-first Search, Graph-Connectivity

Prof. Dr. Rolf Backofen
Bioinformatics Group / Department of Computer Science
Algorithms and Data Structures, January 2019
Graphs

Introduction
Implementation
Application example
Graphs - Overview:

- Besides arrays, lists and trees the most common data structure
  (Trees are a special type of graph)
- Representation of graphs in the computer
- Breadth-first search (BFS)
- Depth-first search (DFS)
- Connected components of a graph
Terminology:

- Each graph $G = (V, E)$ consists of:
  - A set of vertices (nodes) $V = \{v_1, v_2, \ldots\}$
  - A set of edges (arcs) $E = \{e_1, e_2, \ldots\}$
- Each edge connects two vertices $(u, v \in V)$
  - Undirected edge: $e = \{u, v\}$ (set)
  - Directed edge: $e = (u, v)$ (tuple)
- Self-loops are also possible: $e = (u, u)$ or $e = \{u, u\}$
Weighted graph:

- Each edge is marked with a real number named **weight**
- The **weight** is also named **length** or **cost** of the edge depending on the application
**Example:** Road network

- Intersections: vertices
- Roads: edges
- Travel time: costs of the edges

*Figure:* Map of Freiburg © OpenStreetMap
How to represent this graph computationally?

1. **Adjacency matrix** with space consumption $\Theta(|V|^2)$

Figure: Weighted graph with $|V| = 4$, $|E| = 6$

Figure: Adjacency matrix
How to represent this graph computationally?

2. *Adjacency list / fields* with space consumption $\Theta(|V| + |E|)$

Each list item stores the target vertex and the cost of the edge

**Figure:** Weighted graph with $|V| = 4$, $|E| = 6$

**Figure:** Adjacency list
Graph: Arrangement

- Graph is fully defined through the adjacency matrix / list
- The arrangement is not relevant for visualisation of the graph

Figure: Weighted graph with $|V| = 4$, $|E| = 6$

Figure: Same graph ordered by number - outer planar graph
class Graph:
    def __init__(self):
        self.vertices = []
        self.edges = []

    def addVertice(self, vert):
        self.vertices.append(vert)

    def addEdge(self, fromVert, toVert, cost):
        self.edges.append((fromVert, toVert, cost))

    ...
Degree of a vertex: Directed graph: $G = (V, E)$

**Figure:** Vertex with in- / outdegree of 3 / 2

- **Indegree** of a vertex $u$ is the number of edge head ends adjacent to the vertex
  \[
  \text{deg}^+(u) = |\{(v, u) : (v, u) \in E\}| 
  \]

- **Outdegree** of a vertex $u$ is the number of edge tail ends adjacent to the vertex
  \[
  \text{deg}^-(u) = |\{(u, v) : (u, v) \in E\}| 
  \]
Degree of a vertex: Undirected graph: \( G = (V, E) \)

Figure: Vertex with degree of 4

- Degree of a vertex \( u \) is the number of vertices adjacent to the vertex

\[
\text{deg}(u) = |\{\{v, u\} : \{v, u\} \in E\}|\]
Paths in a graph: $G = (V, E)$

- A path of $G$ is a sequence of edges $u_1, u_2, \ldots, u_i \in V$ with
  - Undirected graph: $\{u_1, u_2\}, \{u_2, u_3\}, \ldots, \{u_{i-1}, u_i\} \in E$
  - Directed graph: $(u_1, u_2), (u_2, u_3), \ldots, (u_{i-1}, u_i) \in E$
Paths in a graph: $G = (V, E)$

Figure: Directed path of length 3  
$P = (0, 3, 1, 4)$

Figure: Weighted path with cost 6  
$P = (2, 3, 1)$

- The length of a path is: (also costs of a path)
  - Without weights: number of edges taken
  - With weights: sum of weights of edges taken
Shortest path in a graph: \( G = (V, E) \)

The shortest path between two vertices \( u, v \) is the path \( P = (u, \ldots, v) \) with the shortest length \( d(u, v) \) or lowest costs.

Figure: Shortest path from 0 to 2 with cost / distance \( d(0, 2) = ? \)
Shortest path in a graph: $G = (V,E)$

The shortest path between two vertices $u, v$ is the path $P = (u, \ldots, v)$ with the shortest length $d(u,v)$ or lowest costs.
Diameter of a graph: $G = (V, E)$

$$d = \max_{u, v \in V} d(u, v)$$

The diameter of a graph is the length / the costs of the longest shortest path
Diameter of a graph: \( G = (V, E) \)

\[
d = \max_{u,v \in V} d(u, v)
\]

The diameter of a graph is the length / the costs of the longest shortest path.
Connected components: \( G = (V, E) \)

**Figure:** Three connected components

- **Undirected graph:**
  - All connected components are a partition of \( V \)
    \[
    V = V_1 \cup \cdots \cup V_k
    \]
  - Two vertices \( u, v \) are in the same connected component if a path between \( u \) and \( v \) exists
Connected components: $G = (V, E)$

- Directed graph:
  - Named strongly connected components
  - Direction of edge has to be regarded
  - Not part of this lecture
Graph Exploration: (Informal definition)

- Let $G = (V, E)$ be a graph and $s \in V$ a start vertex
- We visit each reachable vertex connected to $s$
- **Breadth-first search**: in order of the smallest distance to $s$
- **Depth-first search**: in order of the largest distance to $s$
- Not a problem on its own but is often used as subroutine of other algorithms
  - Searching of connected components
  - Flood fill in drawing programs
Breadth-First Search:

1. We start with all vertices unmarked and mark visited vertices.
2. Mark the start vertex \( s \) (level 0).
3. Mark all unmarked connected vertices (level 1).
4. Mark all unmarked vertices connected to a level 1-vertex (level 2).
5. Iteratively mark reachable vertices for all levels.
6. All connected nodes are now marked and in the same connected component as the start vertex \( s \).
The marked vertices create a “spanning tree” containing all reachable nodes.

**Figure:** spanning tree of a breadth-first search
The marked vertices create a “spanning tree” containing all reachable nodes.

**Figure**: spanning tree of a breadth-first search.
The marked vertices create a “spanning tree” containing all reachable nodes.

**Figure:** spanning tree of a breadth-first search
The marked vertices create a “spanning tree” containing all reachable nodes.

Figure: spanning tree of a breadth-first search
The marked vertices create a “spanning tree” containing all reachable nodes.

Figure: spanning tree of a breadth-first search
The marked vertices create a “spanning tree” containing all reachable nodes.

Figure: spanning tree of a breadth-first search
The marked vertices create a “spanning tree” containing all reachable nodes.

Figure: spanning tree of a breadth-first search
The marked vertices create a “spanning tree” containing all reachable nodes.

*level 0*
*level 1*
*level 2*
*level 3*

**Figure:** spanning tree of a breadth-first search
Graphs
Connected Components - Depth-First Search

Depth-First Search:

1. We start with all vertices unmarked and mark visited vertices
2. Mark the start vertex $s$
3. Pick an unmarked connected vertex and start a recursive depth-first search with the vertex as start vertex (continue on step 2)
4. If no unmarked connected vertex exists go one vertex back and continue recursive search (reduce the recursion level by one)
Depth-first search:

- Search starts with long paths (searching with depth)
- Like breadth-first search marks all connected vertices
- If the graph is acyclic we get a topological sorting
  - Each newly visited vertex gets marked by an increasing number
  - The numbers increase with path length from the start vertex
The marked vertices create a different spanning tree containing all reachable nodes

Figure: spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

- **start-node**
- **path 1**
- **path 2**
- **path 3**

**Figure:** spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

- **start-node**
- path 1
- path 2
- path 3

**Figure**: spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

- start-node
- path 1
- path 2
- path 3

**Figure:** spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

**Figure:** spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

**Figure:** spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

- start-node
- path 1
- path 2
- path 3

**Figure:** spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

- start-node
- path 1
- path 2
- path 3

**Figure:** spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

- start-node
- path 1
- path 2
- path 3

Figure: spanning tree of a depth-first search
The marked vertices create a different spanning tree containing all reachable nodes.

- start-node
- path 1
- path 2
- path 3

Figure: spanning tree of a depth-first search
Graphs

Why is this called Breadth- and Depth-First Search?
Runtime complexity:

- Constant costs for each visited vertex and edge
- We get a runtime complexity of $\Theta(|V'| + |E'|)$
- Let $V'$ and $E'$ be the reachable vertices and edges
- All vertices of $V'$ are in the same connected component as our start vertex $s$
- This can only be improved by a constant factor
Application example

Image processing

- Connected component labeling
- Counting of objects in an image
## Application example

### Image processing

**What is object, what is background?**

|   | A   | B   | C   | D   | E   | F   | G   | H   | I   | J   | K   | L   | M   | N   | O   | P   | Q   | R   | S   | T   | U   |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 35| 104 | 80  | 56  | 40  | 16  | 16  | 8   | 16  | 16  | 24  | 32  | 32  | 32  | 32  | 32  | 32  | 32  | 32  | 32  | 32  | 24  | 24  | 16  |
| 36| 80  | 64  | 48  | 32  | 16  | 16  | 16  | 24  | 32  | 40  | 40  | 40  | 40  | 40  | 40  | 40  | 40  | 40  | 40  | 40  | 32  | 24  | 24  | 24  |
| 37| 56  | 48  | 32  | 24  | 8   | 16  | 16  | 32  | 40  | 48  | 48  | 48  | 40  | 40  | 40  | 40  | 40  | 40  | 40  | 40  | 32  | 24  | 24  | 24  |
| 38| 40  | 32  | 24  | 24  | 16  | 32  | 48  | 64  | 72  | 80  | 80  | 72  | 56  | 56  | 48  | 48  | 40  | 40  | 32  | 32  | 32  | 32  | 32  |
| 39| 16  | 16  | 16  | 24  | 24  | 48  | 72  | 88  | 104 | 112 | 112 | 96  | 72  | 64  | 56  | 48  | 40  | 40  | 40  | 40  | 40  | 40  | 40  |
| 40| 16  | 16  | 24  | 40  | 56  | 88  | 120 | 128 | 136 | 144 | 144 | 120 | 96  | 88  | 72  | 56  | 48  | 40  | 40  | 40  | 40  | 40  | 40  |
| 41| 8   | 16  | 24  | 56  | 80  | 120 | 160 | 168 | 168 | 168 | 144 | 120 | 104 | 80  | 64  | 48  | 40  | 40  | 40  | 40  | 40  | 32  | 32  |
| 42| 16  | 32  | 40  | 80  | 112 | 144 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 152 | 128 | 112 | 88  | 64  | 48  | 40  | 32  | 32  | 24  |
| 43| 24  | 40  | 56  | 96  | 136 | 160 | 184 | 184 | 176 | 176 | 176 | 176 | 152 | 152 | 136 | 112 | 88  | 64  | 40  | 32  | 24  | 24  | 16  |
| 44| 40  | 56  | 80  | 112 | 152 | 168 | 184 | 184 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 152 | 136 | 112 | 88  | 64  | 40  | 32  |
| 45| 48  | 72  | 96  | 128 | 160 | 176 | 184 | 184 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 152 | 136 | 112 | 88  | 64  |
| 46| 48  | 72  | 96  | 136 | 168 | 176 | 192 | 192 | 184 | 184 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 |
| 47| 48  | 72  | 96  | 136 | 168 | 176 | 192 | 192 | 184 | 184 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 | 176 |
| 48| 48  | 72  | 96  | 128 | 160 | 184 | 200 | 200 | 192 | 184 | 184 | 160 | 128 | 96  | 64  | 48  | 40  | 32  | 32  | 32  | 32  | 32  | 32  |
| 49| 48  | 72  | 88  | 128 | 160 | 184 | 200 | 200 | 192 | 184 | 184 | 152 | 120 | 88  | 56  | 40  | 32  | 32  | 32  | 32  | 32  | 32  | 32  |
| 50| 48  | 64  | 80  | 112 | 136 | 160 | 176 | 176 | 176 | 176 | 176 | 168 | 160 | 136 | 104 | 80  | 64  | 40  | 32  | 40  | 40  | 56  | 88  |
| 51| 48  | 64  | 72  | 96  | 112 | 128 | 144 | 152 | 152 | 144 | 136 | 112 | 88  | 64  | 40  | 40  | 32  | 48  | 48  | 64  | 112 | 112 |
| 52| 48  | 56  | 64  | 80  | 88  | 104 | 112 | 112 | 120 | 112 | 104 | 88  | 72  | 56  | 32  | 32  | 32  | 32  | 32  | 32  | 32  | 64  | 88  |
| 53| 40  | 48  | 48  | 56  | 64  | 72  | 72  | 80  | 80  | 80  | 72  | 48  | 40  | 24  | 32  | 32  | 32  | 32  | 72  | 104 | 144 | 184 |
| 54| 48  | 48  | 48  | 48  | 48  | 48  | 48  | 56  | 56  | 56  | 56  | 56  | 56  | 48  | 48  | 32  | 32  | 32  | 32  | 32  | 40  | 48  | 88  |

---

*January 2019  Prof. Dr. Rolf Backofen – Bioinformatics - University Freiburg - Germany*
Application example
Image processing

Convert to black and white using threshold:

```python
value = 255 if value > 100 else 0
```
Interpret image as graph:

- Each white pixel is a node
- Edges between adjacent pixels (normally 4 or 8 neighbors)
- Edges are not saved externally, algorithm works directly on array
- Breadth- / depth-first search find all connected components (particles)
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Application example
Image processing

Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 1
- Check neighbors of all new labeled pixels
- Label non-zero pixels as component 1
Find connected components:

- Search pixel-by-pixel for non-zero intensity
- Label found pixel as component 2
Result of connected component labeling:

Figure: Result: particle indices instead of intensities
Further Literature

General

Introduction to Algorithms. 

[MS08] Kurt Mehlhorn and Peter Sanders. 
Algorithms and data structures, 2008. 
Further Literature

■ **Graph Search**

[Wika] Breadth-first search

[Wikb] Depth-first search

■ **Graph Connectivity**

[Wik] Connectivity (graph theory)
https://en.wikipedia.org/wiki/Connectivity_(graph_theory)