Graphs

Part: Structure

Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

Institute of Pervasive Computing

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Definition

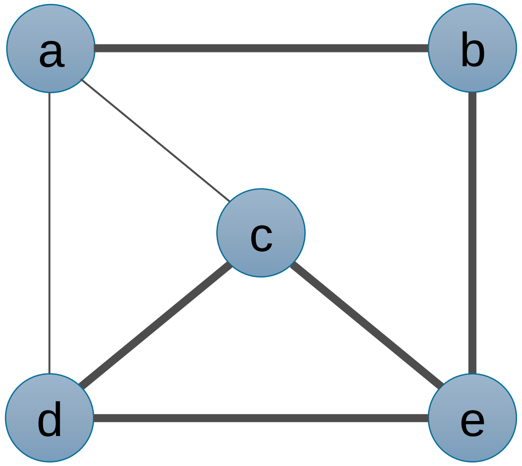
A graph $G = (V, E)$ consists of

V ... a quantity of **vertices**/nodes and
E ... a quantity of **edges**/arcs.

An edge $e = (u,v)$ is a pair of vertices.

Example:

$V = \{ a, b, c, d, e\}$
$E = \{ (a,b), (a,c), (a,d), (b,e), (c,d), (c,e), (d,e)\}$



Definitions of Terms

Two **vertices** are adjacent to each other, if they are **connected by an edge**.

- Example: *a* and *c* are adjacent to each other

An **edge** connecting two adjacent vertices is called **incident to** these **vertices**.

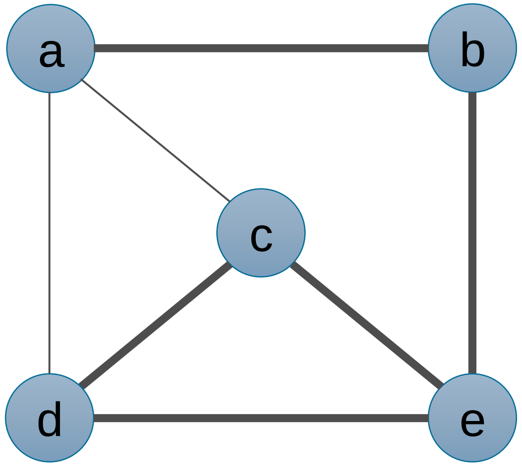
The **degree** of a vertex ***deg(v)*** is defined as the number of vertices adjacent to it.

- We have: $\sum_{v \in V} \deg(v) = 2 * (\#edges)$

- Example: *deg(c) = 3*

A **path** is a **sequence** of **vertices** ($v_1, v_2, ..., v_k$) in which successive vertices v_i and v_{i+1} are adjacent to each other.

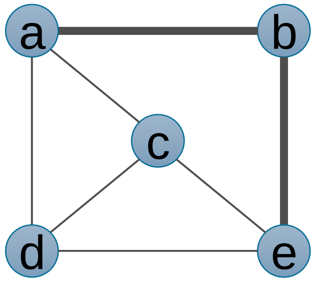
- Example: path (a, b, e, d, c, e)



Definitions of Terms

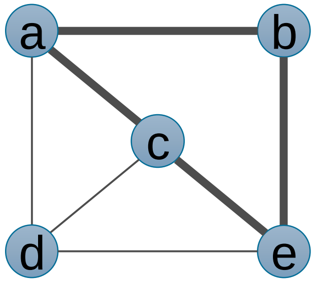
In a **simple path** no vertex occurs more than once.

- Example: a, b, e



A **cyclic path** (short: cycle) is a simple path with the exception, that the **first vertex** in the path is **identical** to the **last vertex** in the path.

- Example: a, b, e, c, a



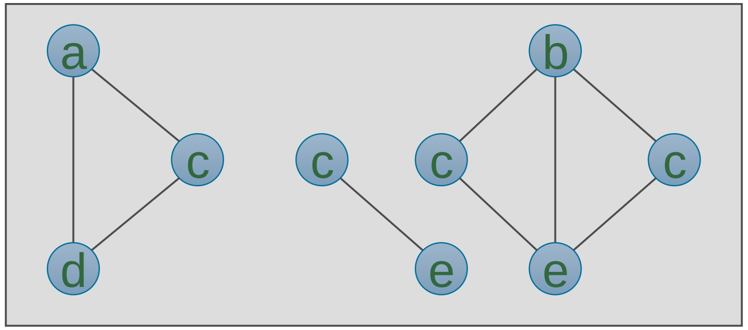
A graph is **connected** if **any two** of its **vertices** are **connected** to each other by a path.

A **subgraph** is a subset of vertices and edges of a graph, which in turn form a graph.

Definitions of Terms

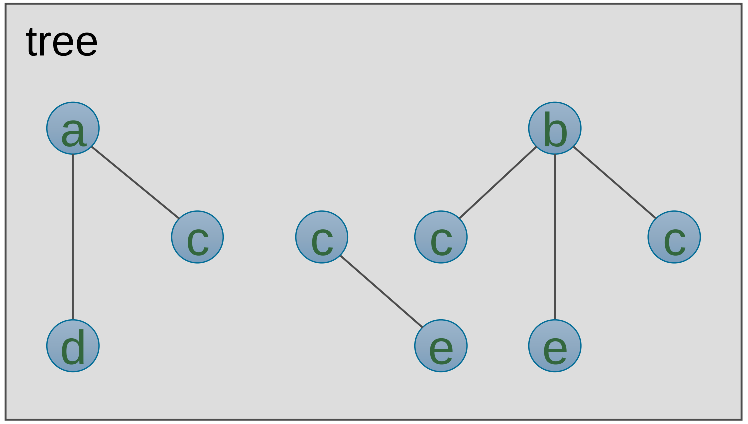
A **connected component** in a graph is a **maximum connected subgraph**.

- Example:
  Graph with 3 connected components



A **tree** is a connected graph **without cycles**.

A **forest** is a **set** of **trees**.



Connectivity

In a **complete graph** **each pair** of vertices is **adjacent** to each other.

Let

n ... number of **vertices** and

m ... number of **edges**

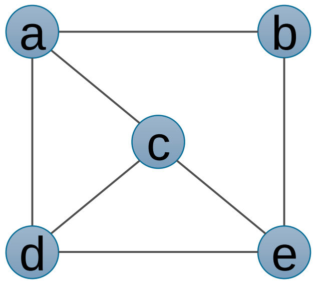
- in a **complete graph** we have: $$m = \frac{1}{2}\sum_{v \in V} \deg(v) = \frac{1}{2}\sum_{v \in V} (n-1) = \frac{n(n-1)}{2}$$
- in a **non-complete** graph we have: $$m < \frac{n(n-1)}{2}$$
- in a **tree** we have: $$m = n - 1$$

If $m < n\text{-}1$, then the graph is **not connected** (consists of more than one connected component).

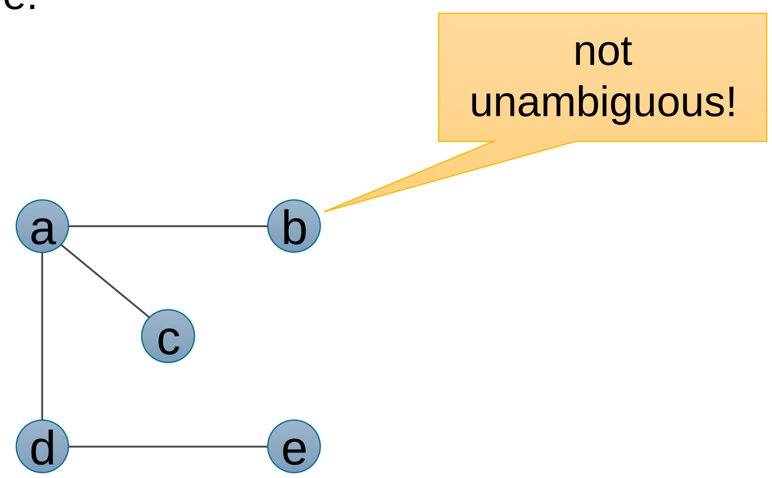
Spanning Tree

A **spanning tree ST** in a graph *G* is a subgraph of *G* for which we have:

- **ST** is a **tree**
- **ST** contains **all vertices** of *G*



G



Spanning tree of *G*

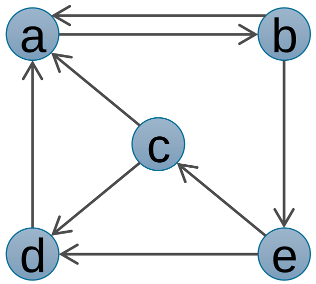
If a **single edge** is **removed**, the graph is **no longer connected**.

Directed Graph

An edge *(v,w)* is defined as **directed**, if it leads *v* to *w*, but not vice versa. Then *(v,w)* is an ordered pair.

- Illustration: as arrow

A graph is directed (**digraph**), if it contains **directed edges**.



A directed graph without cycles is called a **directed acyclic graph** (**DAG**).

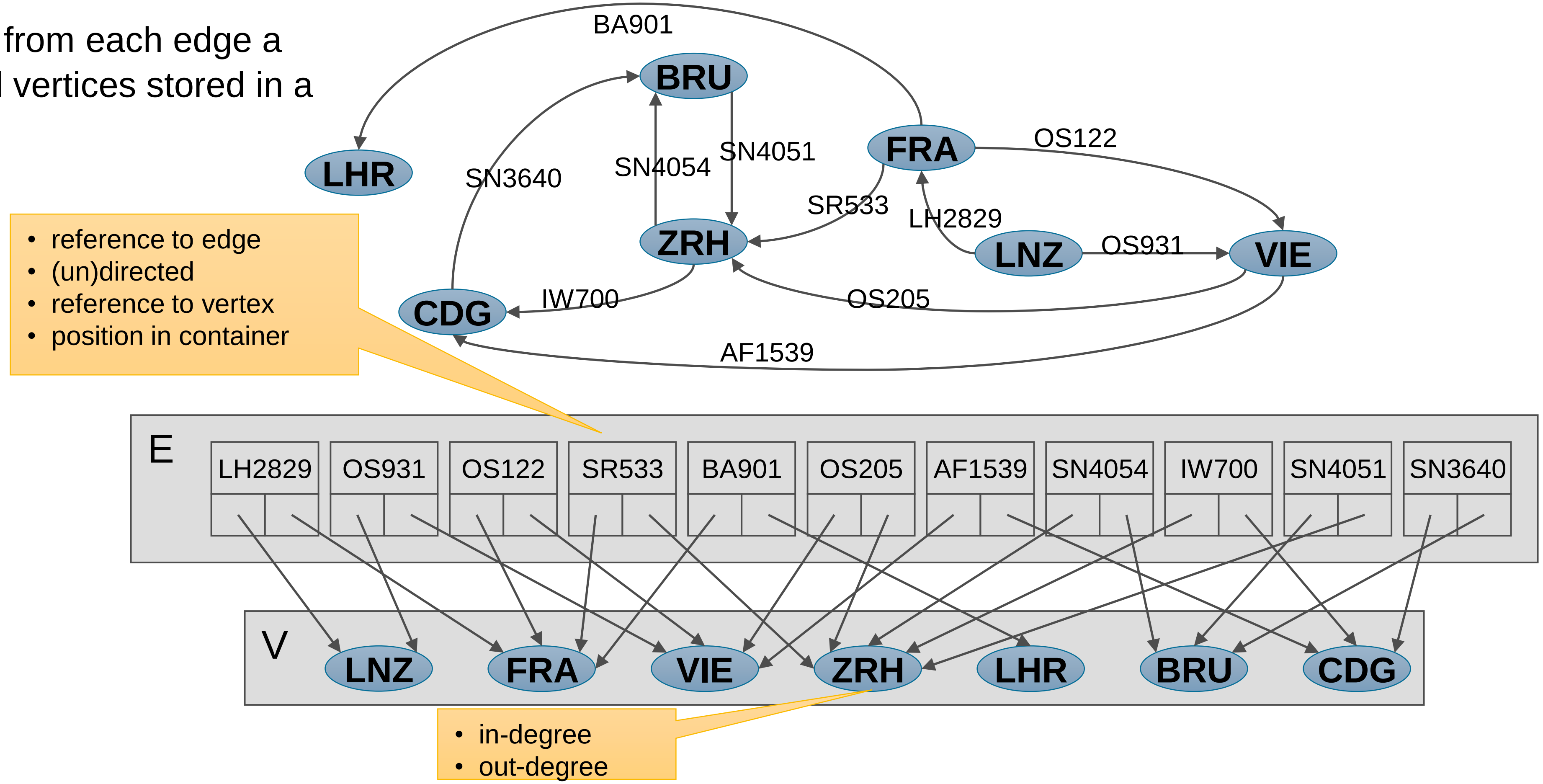
A **digraph** is **strongly connected**, if **each vertex** can be **reached from any other vertex**.

A **strongly connected component** in a digraph is a subgraph, in which each vertex can be reached from any other vertex.

Data Structures for Storing Graphs

Edge list

Edges are stored **in a list**, from each edge a **reference leads** to the end vertices stored in a separate data structure.



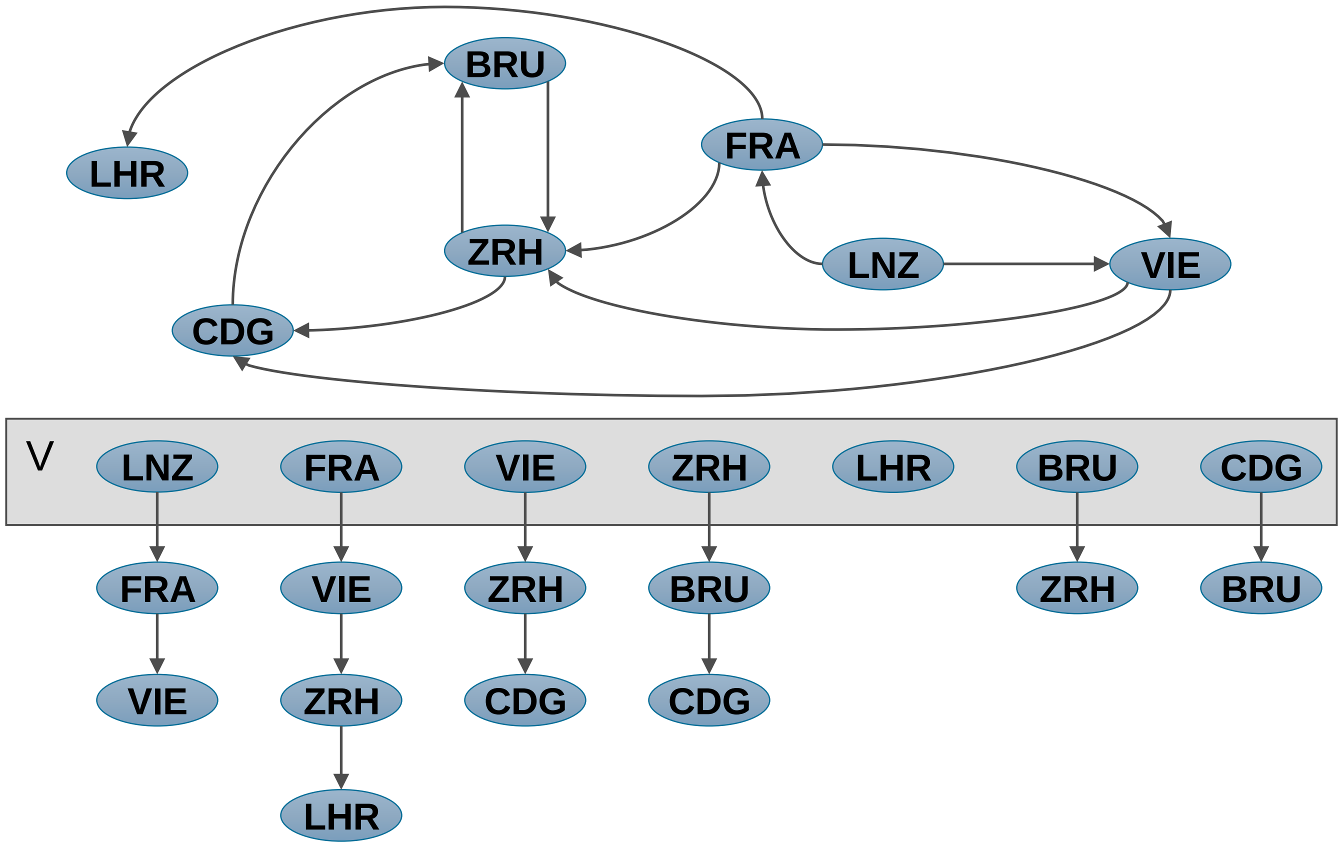
A. Ferscha

Institute of Pervasive Computing

Data Structures for Storing Graphs

Adjacency list (traditional)

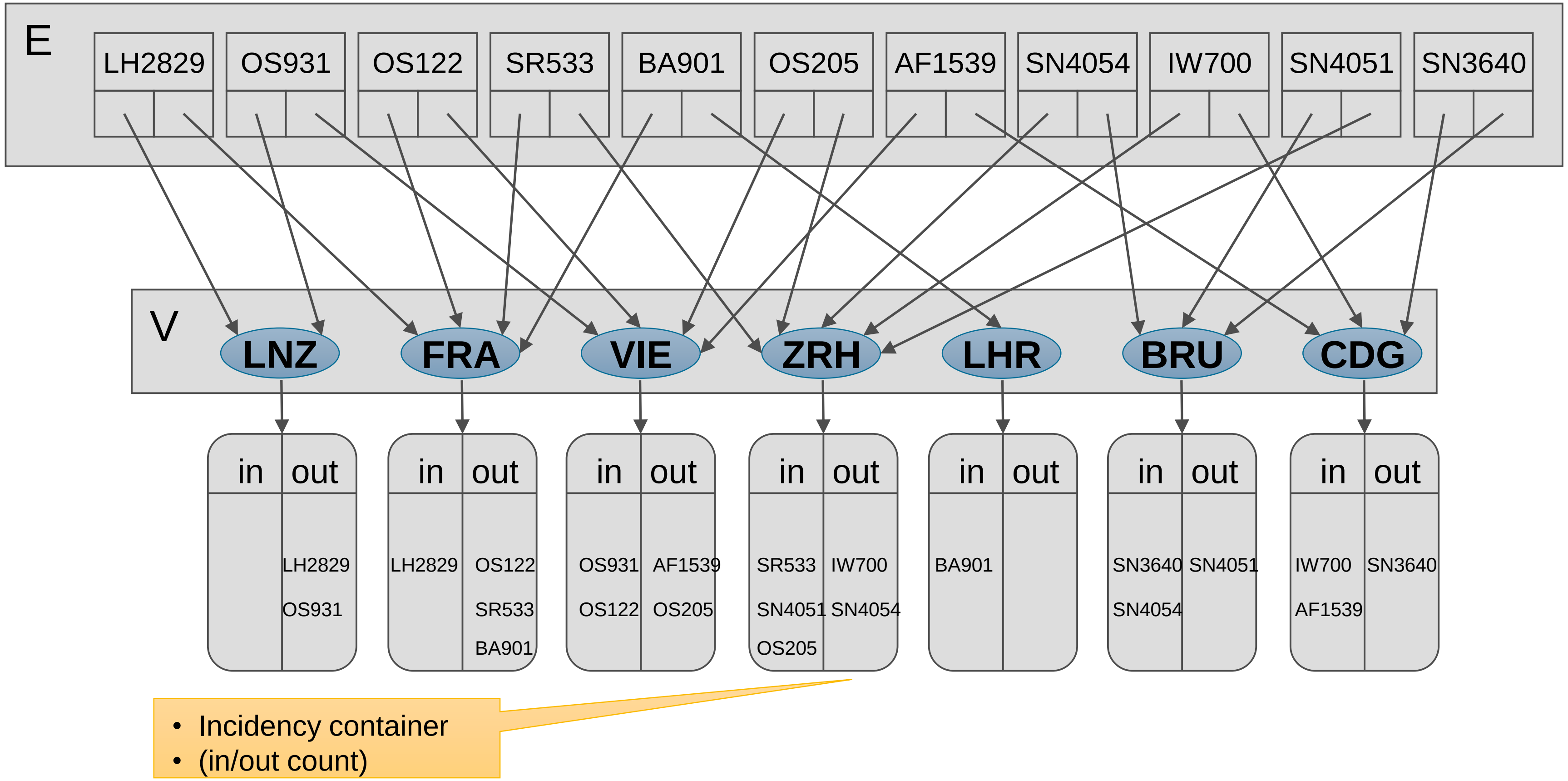
- For each vertex its **adjacent vertices** are stored in a **list**.
- All vertices with adjacency lists are again stored in a list.



Data Structures for Storing Graphs

Adjacency list (modern)

- **Extension** of the **edge list** by a **list of** the **incident edges** for each vertex



Data Structures for Storing Graphs

Adjacency matrix (traditional)

- Matrix *M* where *M[i,j]* is **true** if an **edge** from **vertex *i*** to **vertex *j*** exists, or **false** otherwise.

from \ to	LNZ	FRA	VIE	ZRH	LHR	BRU	CDG
LNZ	F	T	T	F	F	F	F
FRA	F	F	T	T	T	F	F
VIE	F	F	F	T	F	F	T
ZRH	F	F	F	F	F	T	T
LHR	F	F	F	F	F	F	F
BRU	F	F	F	T	F	F	F
CDG	F	F	F	F	F	T	F

Data Structures for Storing Graphs

Adjacency matrix (modern)

- Matrix elements store a reference to an edge object

from \ to	LNZ	FRA	VIE	ZRH	LHR	BRU	CDG
LNZ	F	LH2829	OS931	F	F	F	F
FRA	F	F	OS122	SR533	BA901	F	F
VIE	F	F	F	OS205	F	F	AF1539
ZRH	F	F	F	F	F	SN4054	IW700
LHR	F	F	F	F	F	F	F
BRU	F	F	F	SN4051	F	F	F
CDG	F	F	F	F	F	SN3640	F

Complexity Comparison

Assumption:
- Edges, vertices as doubly linked list
- Container with position information

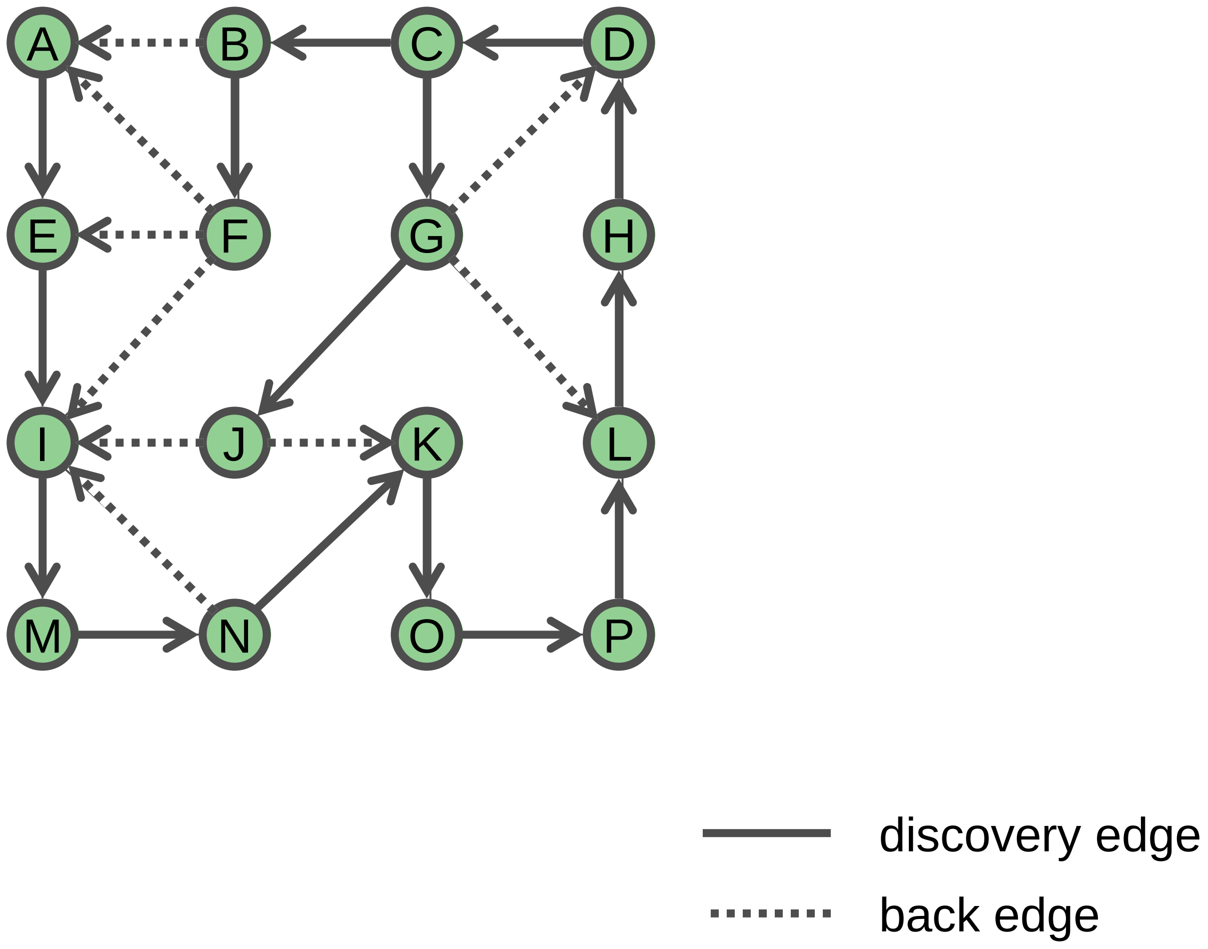
	Complexity		
	Edge list	Adjacency list	Adjacency matrix
Space	**$O(m+n)$**	**$O(m+n)$**	**$O(n^2)$**
Operations:			
`size`, `isEmpty`	$O(1)$	$O(1)$	$O(1)$
`vertices`	$O(n)$	$O(n)$	$O(n)$
`edges`	$O(m)$	$O(m)$	$O(m)$
`endVertices`, `opposite`, `isDirected`	$O(1)$	$O(1)$	$O(1)$
`incidentEdges`	$O(m)$	$O(deg(v))$	$O(n)$
`areAdjacent`	**$O(m)$**	**$O(min(deg(u),deg(v)))$**	**$O(1)$**
`insertVertex`	**$O(1)$**	**$O(1)$**	**$O(n^2)$**
`removeVertex`	$O(m)$	$O(deg(v))$	$O(n^2)$
`insertEdge`, `insertDirectedEdge`, `removeEdge`	$O(1)$	$O(1)$	$O(1)$

Graph Structure Analysis

Graph Traversal – Depth-First Search (DFS)

```
Algorithm DFS(v);

    Input: A vertex v in a graph
    Output: A labeling of edges as "discovery" and "back" edges

    for each edge e incident to v do

        if edge e is unexplored then
            let w be the other endpoint of e

            if vertex w is unexplored then
                label e as a discovery edge
                recursively call DFS(w)

            else
                label e as a back edge
```

DFS :: Example



Properties of DFS

For an **undirected graph G** in which a **DFS starting** with **vertex *s*** is executed, we have:

- The traversal **visits all vertices** in the connected component (=maximum connected subgraph), which contains *s*.
- The set of **discovery edges** form a **spanning tree** for this connected component.

Complexity of DFS

If $\boldsymbol{n_s}$ is the number of **vertices** in a connected component within *s*
and if $\boldsymbol{m_s}$ is the number of **edges** in the connected component of *s*,

then the complexity of DFS is $O(n_s + m_s)$ on the assumptions:

- Graph is stored so that access to the vertices and edges is O(1)
- Marking and testing of the edges is O(1)
- There is a mechanism that systematically searches the edges of a node without looking at an edge more than once.

DFS :: Finding a Path

Each **vertex** is labeled initially as **UNEXPLORED** and can be relabeled as **VISITED**.

Each **edge** is labeled initially as **UNEXPLORED** and can be relabeled as **DISCOVERY** or **BACK**.

A **stack S** is used to keep track of the **path between** the **start vertex** and the **current vertex**.

When the **destination vertex** *z* is **reached**, the **stack content** is **returned** as the path between *v* and *z*.

Based on Goodrich et al., *Data Structures & Algorithms in Python*

```
Algorithm pathDFS(G,v,z);
    Input: A graph G and the vertices v (start) and z
    (destination) in graph G
    Output: Path as a sequence of vertices

    setLabel(v, VISITED)
    S.push(v)
    if v = z
        return S.elements() and terminate algorithm
    for all e ∊ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                S.push(e)
                pathDFS(G, w, z)
                S.pop(e)
            else
                setLabel(e, BACK)
    S.pop(v)
```

DFS :: Finding a Cycle

Each **vertex** is labeled initially as **UNEXPLORED** and can be relabeled as **VISITED**.

Each **edge** is labeled initially as **UNEXPLORED** and can be relabeled as **DISCOVERY** or **BACK**.

A **stack S** is used to keep track of the **path between** the **start vertex** and the **current vertex**.

A **cycle** is detected when a back edge to an already visited vertex is recognized, which is not a parent.

Based on Goodrich et al., *Data Structures & Algorithms in Python*

```
Algorithm cycleDFS(G,v);
    Input: A vertex v (start) in a graph G
    Output: Path of in the graph representing the cycle

    setLabel(v, VISITED)
    S.push(v)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                cycleDFS(G, w)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements() and terminate
    S.pop(v)
```

Graph Traversal – Breadth-First Search (BFS)

Principle:

- Let ***s*** be the **start** node of **BFS**, set *s* to **level 0**
- In the first step, **visit all** (not yet visited) vertices that are **adjacent** to *s* and set them to **level 1.**
- In the next step **visit for** all vertices on level 1 all **not yet visited adjacent** vertices and set them to **level 2.**
- Repeat this step as long as all vertices have been reached.

Result:

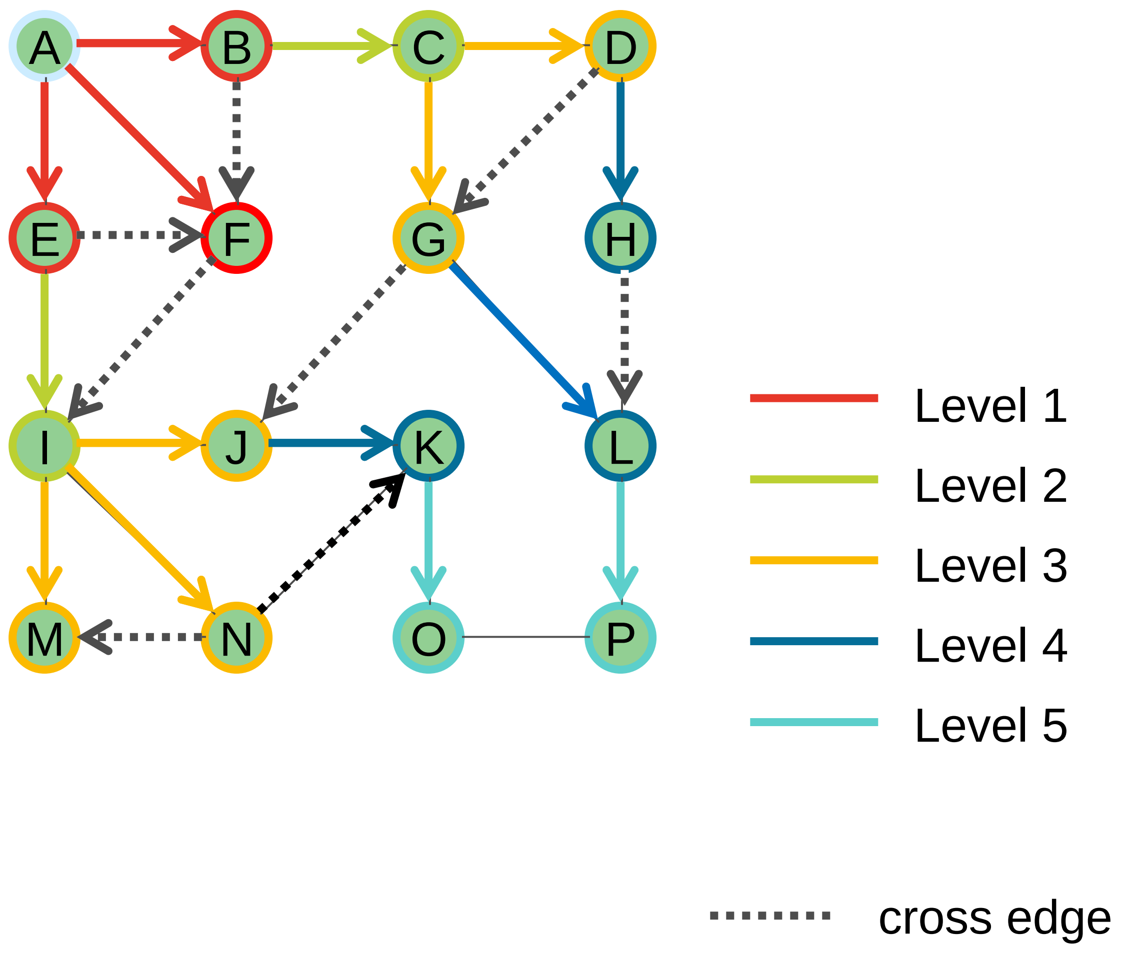
- Traversal of the graph
- **Level** of a vertex *v* shows the **length of the shortest path** from *v* to *s*.

Graph Traversal – Breadth-First Search (BFS)

Pseudo code example:

```
Algorithm BFS(v);

    Input: A vertex s in a graph
    Output: A labeling of edges as "discovery" and "cross" edges

    initialize container L_0 to contain vertex s

    i ← 0
    while L_i is not empty do
        create container L_i+1 to initially be empty
        for each vertex v in L_i do
            for each edge e incident on v do
                if edge e is unexplored then
                    let w be the other endpoint of e
                    if vertex w is unexplored then
                        label e as discovery edge
                        insert w into L_i+1
                    else
                        label e as cross edge
        i ← i+1
```

BFS :: Example



A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Level 1
Level 2
Level 3
Level 4
Level 5
cross edge

Properties of BFS

Let G be an **undirected graph** in which BFS is executed starting with vertex s, then we have:

- The traversal **visits all vertices** in the connected component, which contains s.
- The set of **discovery edges** form a **spanning tree** (BFS tree) for this connected component.
- For each vertex v on **level *i***, the **path** to s **along** the **BFS tree** has **length *i***
  and **any other** path from v to s has **at least length *i***.
- If (u,v) is an edge that is not in the BFS tree, then the **levels** of u and v **differ by 1 at maximum**.

Complexity of BFS (analogue DFS)

If n is the number of **vertices** of G and m the number of **edges**, then the complexity of **BFS** in G is $O(n + m)$

With the **same complexity** the following tasks can be solved:

- Test, if G is **connected**
- Calculation of a **spanning tree** in G
- Calculation of the **connected components** in G
- Calculation of a **cycle** in G (or that G has no cycles)

Graph Traversal – DFS for Directed Graphs

Same algorithms **as** for **undirected graphs**

- The result can also be a set of unconnected DFS trees (forest)

Accessibility in directed graphs

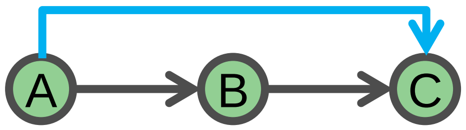
- A **DFS tree** with **root *s*** contains **all vertices** that can be **reached** from *s* via directed edges.

Transitive closure

- The **transitive closure** *G** to a graph *G* is obtained by **inserting** a **directed edge** (*v*,*w*) **if** *v* is **reachable** from *w* (there is a directed path from v to w)

Calculation of the **transitive closure** for graph G:

- Apply DFS on each vertex of G
- Complexity: ***O(n (n+m))***
- Alternative: **Floyd-Warshall Algorithm**



Floyd-Warshall Algorithm

Assumption: Operations areAdjacent and insertDirectedEdge have complexity O(1) (Graph is e.g. as adjacency matrix stored)

Algorithm FloydWarshall(G);
    let v_1 ... v_n be an arbitrary ordering of the vertices of G_0 = G
    for k = 1 to n do
        // consider all possible routing vertices vk
        G_k = G $_{k-1}$
        for each (i, j = 1,..,n) (i != j) (i,j != k) do
            // for each pair of vertices v_i and v_j
            if G_{k-1}.areAdjacent(v_i,v_k) and G_{k-1}.areAdjacent(v_k,v_j) then
                    G_k. insertDirectedEdge(v_i,v_j,null)
        return G_n

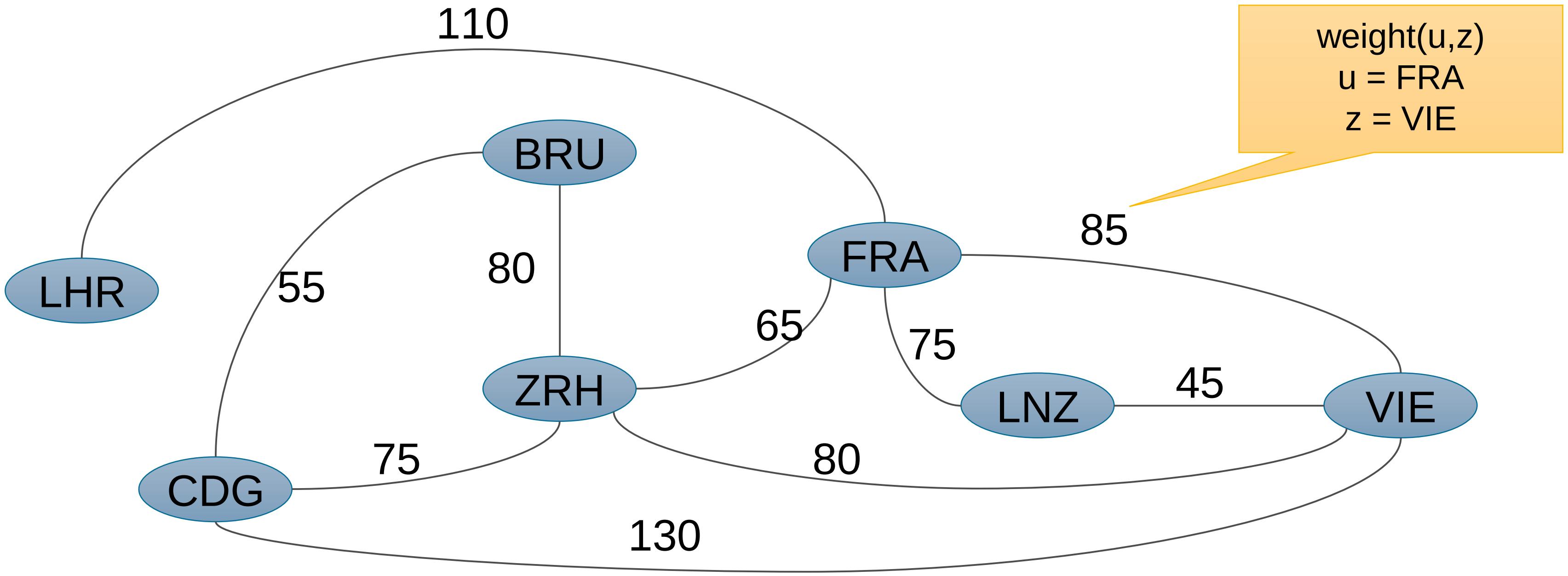
Digraph G_k is the **subdigraph** of the **transitive closure** of ***G***,
which results from the paths to the **intermediate vertices** of set $\{v_1, ..., v_k\}$

Runtime: ***$O(n^3)$***

Weighted Graph

An edge is weighted, if a numerical value is assigned to it.

- Value can represent for example a distance, travel time or **cost**.



Shortest Path – Dijkstra’s Algorithm

BFS finds **path with minimum number of edges**

- Corresponds to the **shortest path** if **all edges** have the same weight

Dijkstra‘s algorithm

- Finds shortest path for **all vertices *z*** to **start vertex *s*** in a graph
  - with undirected edges and
  - with non-negative edge weights
- based on ***greedy method***

Algorithmic idea

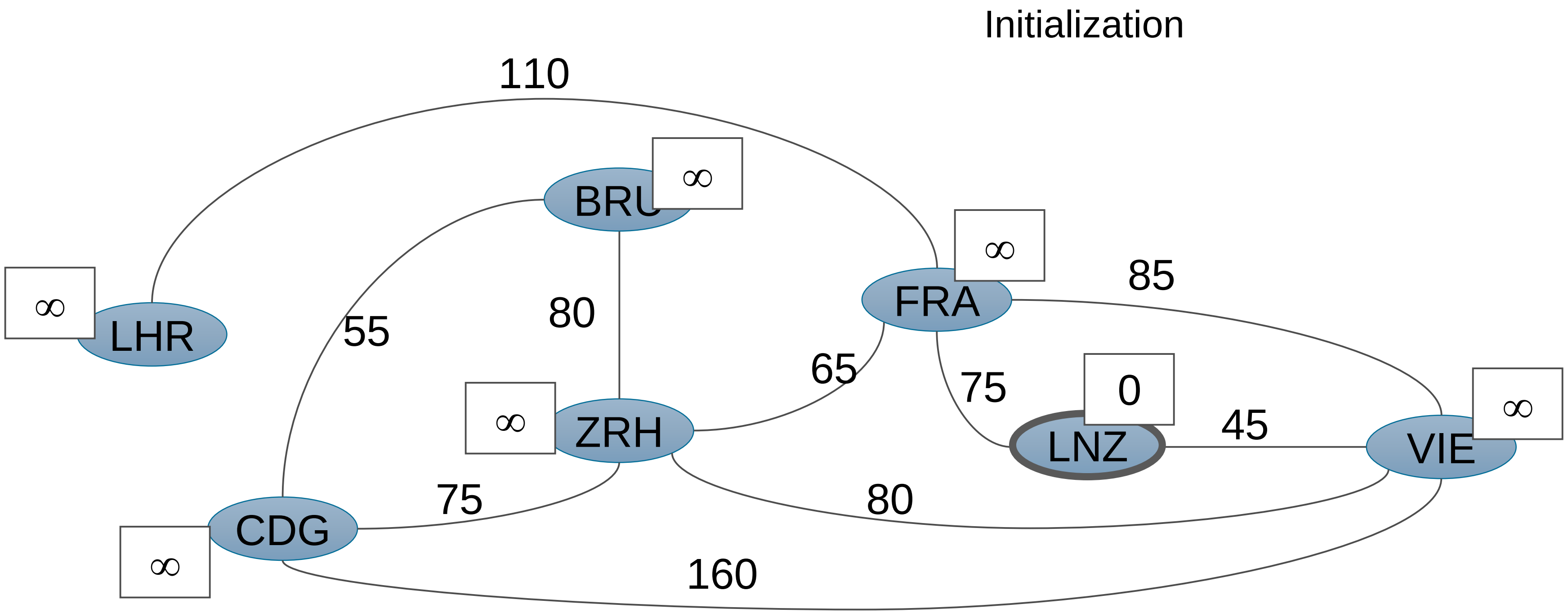
- Set of **nodes** for which a **path** has **already** been **found** is stored in **set *C***.
- *D(z)* denotes the **shortest path** from *v* to *s* found **so far**.
- When a **new node *u*** is visited, the system checks **whether** a route **via this** node to an already visited node *z* has a **shorter distance** than the shortest route found so far, i.e. whether ***D[u] + weight(u,z) < D[z]***
- If yes, then this path is saved via ***b*** as the **new shortest path** to ***z*** and ***D[z]*** is **updated (relaxation)**

Dijkstra's Algorithm

```
Algorithm ShortestPath(G,s);

    Input:    A weighted Graph G and distinguished vertex s of G
    Output:   labels D[u] for each vertex u of G giving the length of the shortest
              path from u to s in G

    initialize D[s] ← 0 and D[u] ← + ∞ for each vertex u ≠ s

    let Q be a priority queue that contains all of the vertices of G
        using the D labels as keys
    while Q ≠ ∅ do
        // pull u into cloud C
        u ← Q.removeMinElement()
        for each vertex z adjazent to u such that z is in Q do
            // perform relaxation operation on edge (u,z)
            if D[u] + w((u,z)) < D[z] then
                D[z] = D[u] + w((u,z))
                change the key value of z in Q to D[z]
    return label D[u] of each vertex u
```

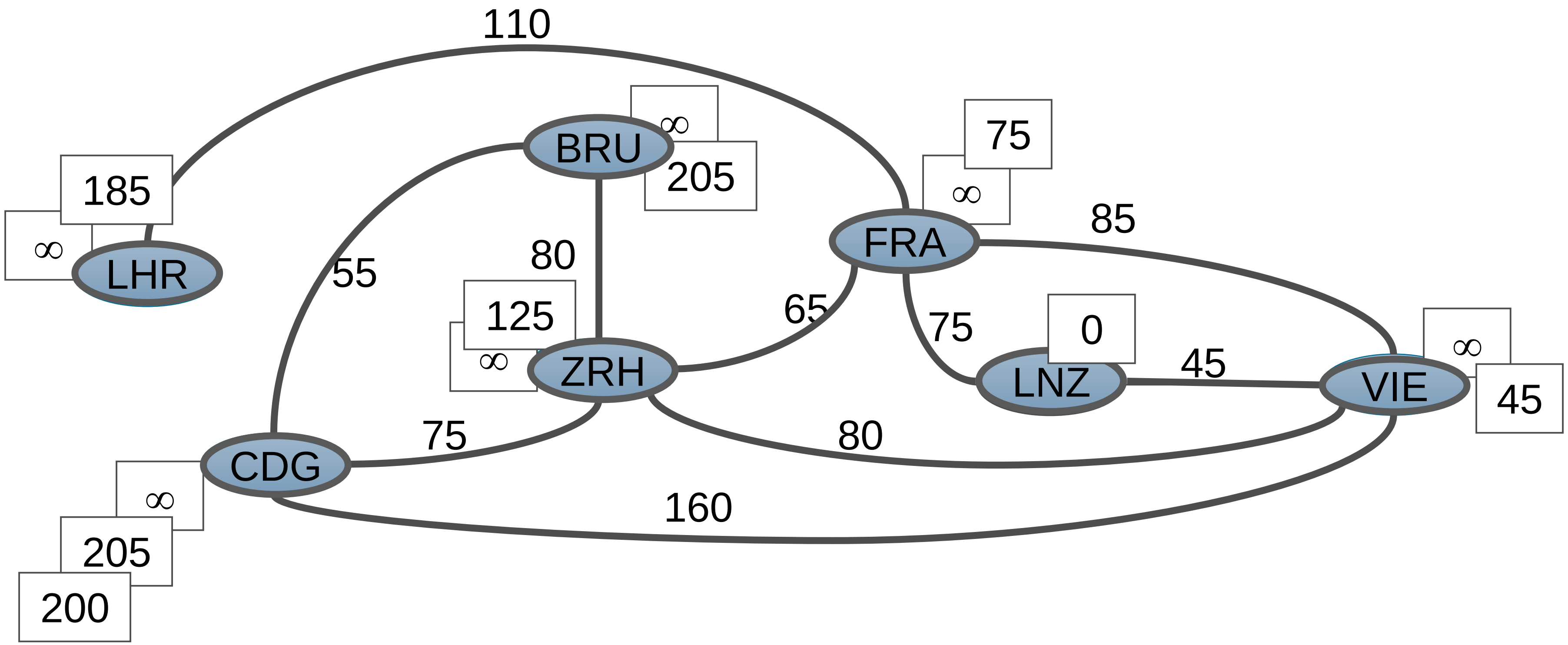
Dijkstra's Algorithm :: Example

Search shortest path starting from LNZ



Dijkstra's Algorithm :: Example

Search shortest path starting from LNZ



Runtime of Dijkstra's Algorithm

Assumption: ***G*** is stored as **adjacency list**

Vertices that are adjacent to *u* can be visited in O(j),
where *j* is the number of vertices that are adjacent to *u*

Variation 1: Priority Queue Q implemented as heap

Initialization of all vertices in Q ***O(log n)***

To be preferred when number of edges is small:

$m < n^2 / \log n$

Within the loop:

- **removeMin** *O(log n)*
- **key updates** *O(log n)* (if locators are used in the heap)
- **Relaxation** *O(degree(u) log n)*

Runtime of the loop

$$\sum_{u \in G} \big(1 + degree(u)\big) \log(n)$$

Therefore total runtime of Dijkstra: ***O((n+m) log n)***

Runtime of Dijkstra's Algorithm

Assumption: ***G*** is stored as **adjacency list**

Vertices that are adjacent to *u* can be visited in O(j),
where *j* is the number of vertices that are adjacent to *u*

Variation 2: Priority Queue as unsorted list

To be preferred if the number of edges is large:

$m > n^2 / \log n$

Within the loop

- **removeMin** $O(n)$
- **key updates** $O(1)$

Therefore total runtime of Dijkstra: $\boldsymbol{O(n^2 + m)}$

Minimum Spanning Tree (MST)

MST is a spanning tree with minimum total edge weight

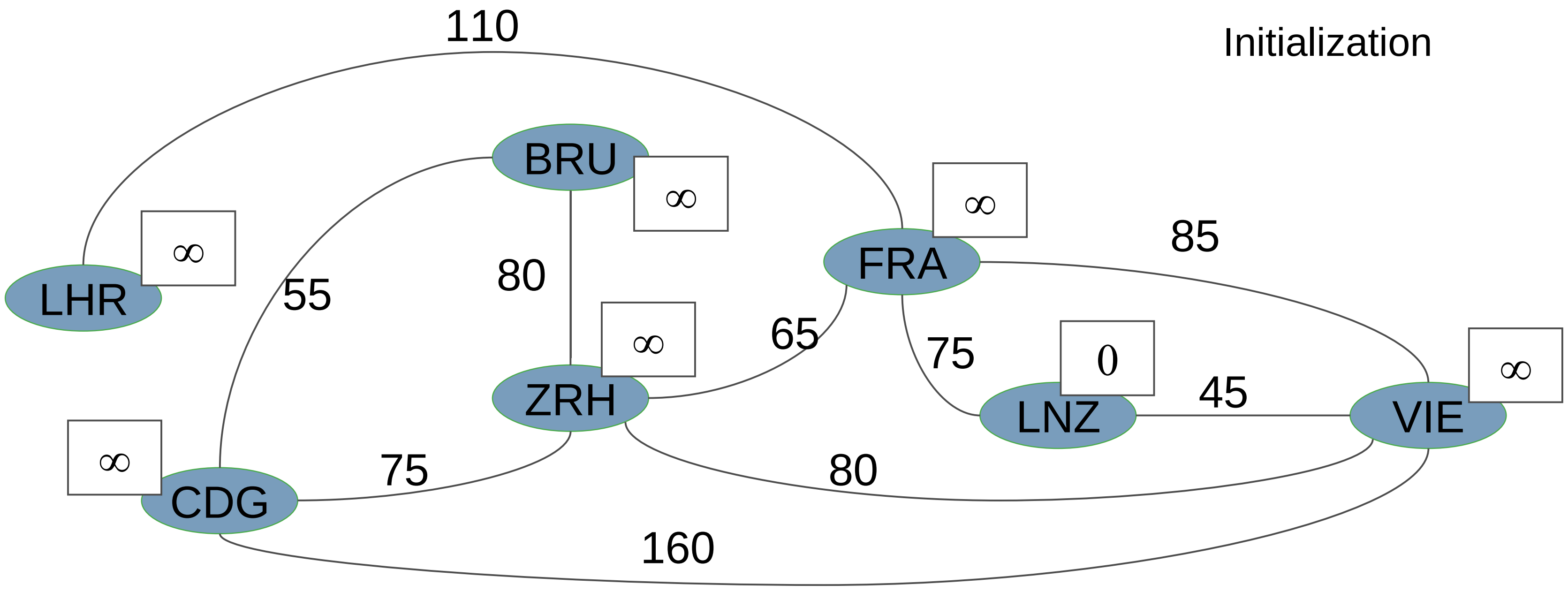
- **Prim-Jarnik** algorithm (similar to Dijkstra)
  - **Start with (any)** start vertex ***v***
  - Tree is constructed **vertex by vertex**
  - **Insert** edge (*v*,*u*) and vertex *u*, which are:
    - Vertex ***v*** is **already in the tree**
    - Vertex ***u*** is **not** yet **in the tree**
    - The weight of (*v*,*u*) is the **minimum** of the weights **of all possible candidates** *u.*
  - For each vertex *D[u]* is stored
    - Indicates for a vertex that has not yet been visited, the minimum of the weights of all edges with which *u* could be connected to the tree.
    - Advantage: shortened search for minimum

Prim-Jarnik Algorithm

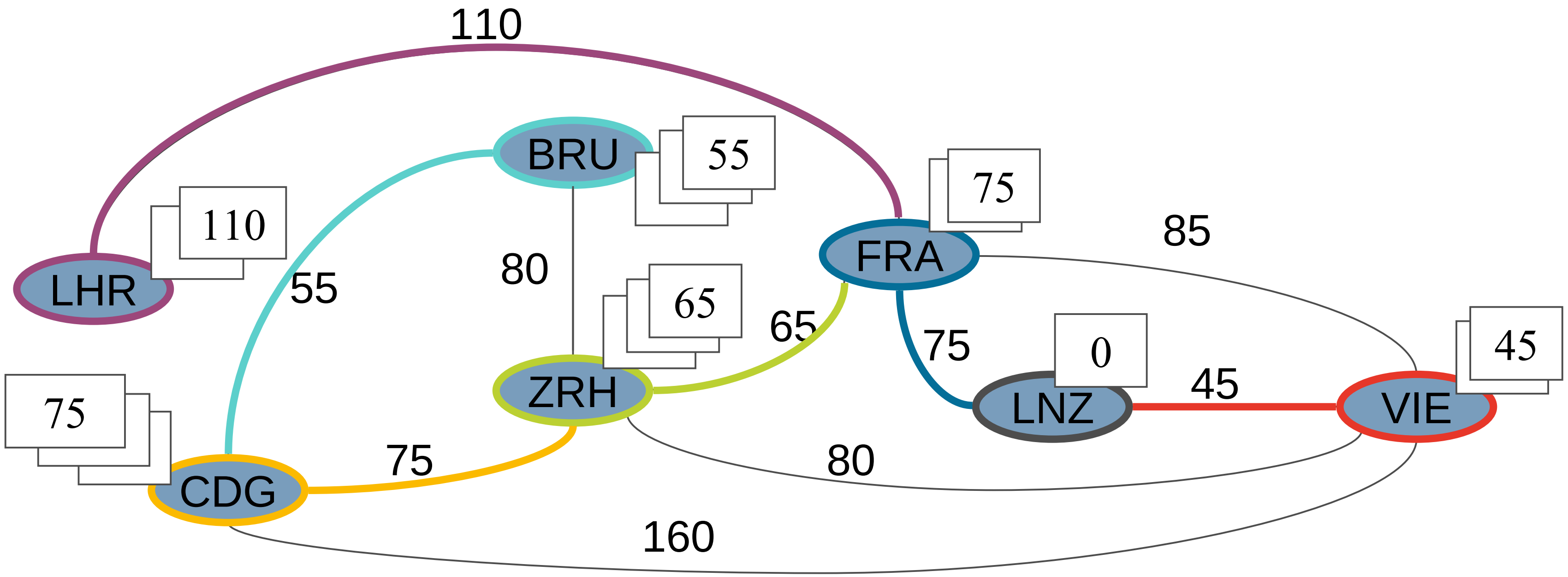
```
Algorithm PrimJarnik(G):

    Input:    A weighted graph G.
    Output:   A minimum spanning tree T for G.

pick any vertex v of G                // grow the tree starting with vertex v
T ← {v}
D[v] ← 0
E[v] ← ∅
for each vertex u ≠ v do              D[u] ← +∞
let Q be a priority queue that contains vertices, using the D labels as keys
while Q ≠ ∅ do                        // pull u into the cloud C
    u← Q.removeMinElement()
    add vertex u and edge E[u] to T
    for each vertex z adjacent to u do
    if z is in Q                      // perform the relaxation operation on edge (u, z)
        if weight(u, z) < D[z] then
            D[z] ← weight(u, z)
            E[z] ← (u, z)
            change the key of z in Q to D[z]
return tree T
```

runtime
O((n+m) log n)

Prim-Jarnik Algorithm :: Example



Prim-Jarnik Algorithm :: Example



PQ: LNZ VIE FRA ZRH CDG BRU LHR

Kruskal’s Algorithm

Put one edge after the other into the **MST** under the following conditions:

- Select the **edge** with the **lowest weight**
- An edge is only **inserted**, **if no cycle will result from the insertion**

Data structure:

- Algorithm **manages** a **set** of **trees** (forest)
- An edge is **accepted** if it **connects vertices from different trees**

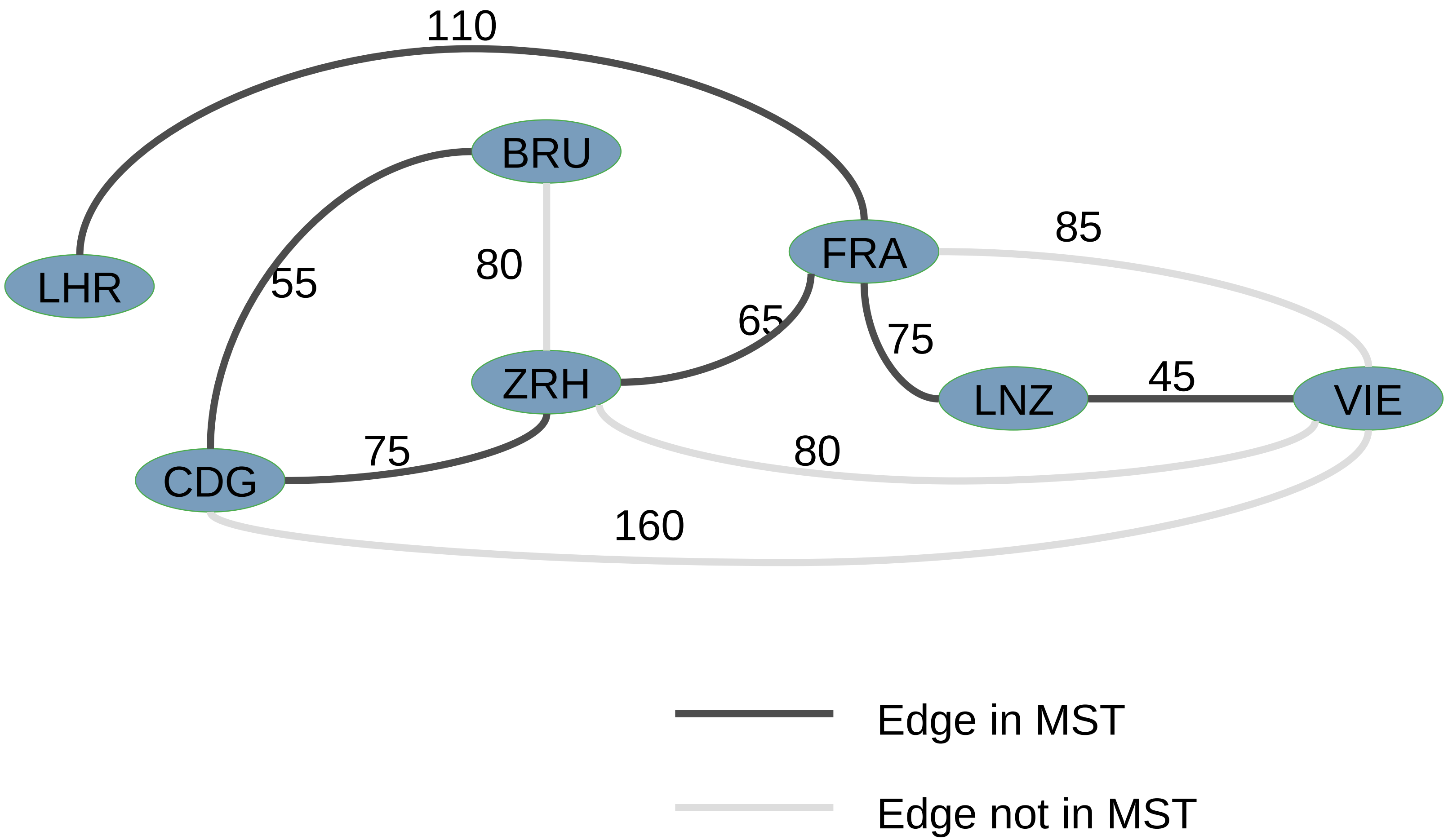
Therefore the data structure must manage disjunctive subsets and support the following operations:

- **find(u)** returns the set, which *u* contains
- **union(u,v)** merges the sets, which *u* and *v* contain

Kruskal’s Algorithm

```
Algorithm Kruskal(G):

    Input:    A weighted graph G.
    Output:   A minimum spanning tree T for G.

let P be a partition of the vertices of G, where each
vertex forms a separate set

let Q be a priority queue storing the edges of G, sorted
by their weights

T ←∅
while Q ≠ ∅ do
    (u,v) ← Q.removeMinElement()
    if P.find(u) ≠ P.find(v) then
        add edge (u,v) to T
        P.union(u,v)

return T
```

Kruskal’s Algorithm :: Example



Institute of Pervasive Computing

Distance and Centrality

Distance between two vertices *u,v* in a Graph *G* (V,E)

- **shortest path between them**
- G **unweighted**: *d(u,v)* **number of hops** of the shortest path
- G **weighted**: *d(u,v)* **sum of the weights** of the shortest path

Eccentricity

- **Maximum (shortest) distance** from the vertex **to any other** vertex in a connected graph

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

Diameter

- is the **maximum (shortest) distance between two vertices** of a connected graph

$$\Delta = \max_{v \in V} \sigma(v)$$

Distance and Centrality

Radius

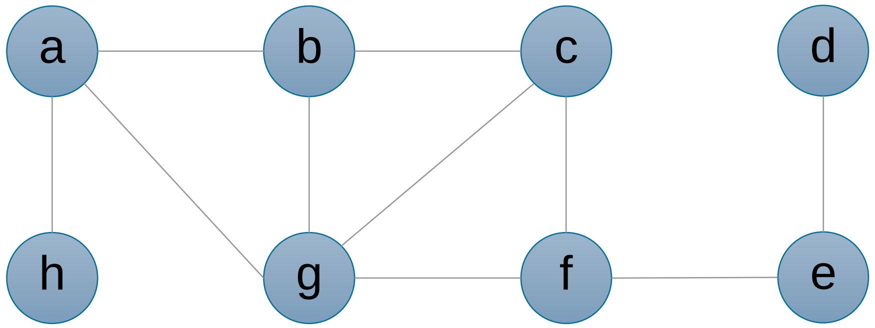
Minimum eccentricity among all vertices

$$rad(G) = \min_{v \in V} \sigma(G)$$

Relation: **diameter-radius**

$$rad(G) \leq diam(G) \leq 2rad(G)$$

Distance and Centrality :: Example



$\Delta(G)$

- h-a-g-f-e-d = 5

eccentricities a, b, c, d, e, f, g, h

- 4,4,3,5,4,3,3,5

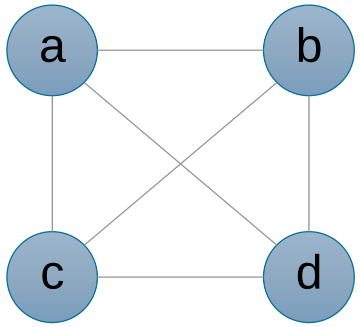
radius:

- 3 (min eccentricities)

Distance and Centrality

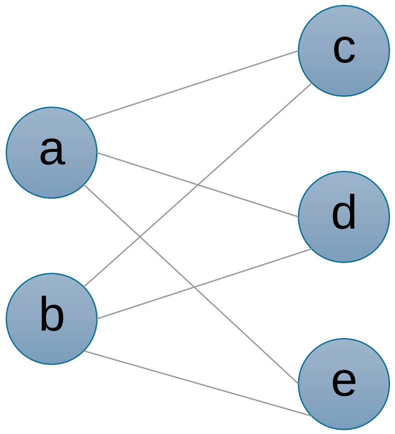
Given a **complete** graph K_n where $n \geq 2$:

- the **diameter** and **radius** is **1**



Given a **complete bipartite** graph $K_{m,n}$ where either m or $n \geq 2$

- the **diameter** and **radius** is **2**



Distance and Centrality

Average distance of a vertex

- *The **average distance** $d_v(av)$ **of a vertex v** is defined as the arithmetic mean of the distance **of v to all other vertices**:*

$$d_v(av) = \frac{1}{n-1} \sum_{u \in V} \delta(u, v)$$

Average distance of a graph

- *The **average distance** $d_v(av)$ **of a graph G** is defined as the arithmetic mean of the distance **among all vertices**:*

$$d_G(av) = \frac{1}{n} \sum_{v \in V} d_v(av)$$

Distance and Centrality

Average distance of a graph cont'd.

substituting in equations above:

$$d_G(av) = \frac{1}{n} \sum_{v \in V} d_v(av) = \frac{1}{n} \sum_{v \in V} (\frac{1}{n-1} \sum_{u \in V} \delta(u,v))$$

Interpretation

- **low average distance** => short paths between most of the vertices
- **high average distance** => general difficult to reach one vertex from another

Short Distance Graphs

d … vertex degree

$\delta_{i,j}$ … (minimum) distance (between vertex *i* and vertex *j*)

$\Delta = max(\delta_{i,j}\ 0 \leq i,j \leq (M-1))$ … diameter

$\frac{\sum_{\delta=1}^{\Delta} \delta N_\delta}{M-1}$ … average distance

N_δ … number of distance δ vertices

Moore Bound

$$\bar{M} = \begin{cases} 2\Delta + 1 \ if \ d = 2 \\ \frac{d(d-1)^\Delta - 2}{d-2} \ if \ d > 2 \end{cases}$$

d=4

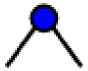
$$\bar{\Delta} = \frac{log(\frac{\bar{M}+1}{2})}{log 3}$$

Short Distance Graphs

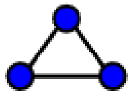
Moore Graph: a regular graph of **degree *d*** and **diameter *Δ*** (also ***k***) whose number of vertices

equals the **upper bound** $= \begin{cases} 2\Delta + 1 \;\; if\; d = 2 \\ \frac{d(d-1)^{\Delta}-2}{d-2} \;\; if\; d > 2 \end{cases}$

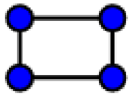
(construction: given ***d*** and ***k***, what is the graph with maximum ***N****?)*



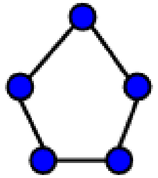
d = 2



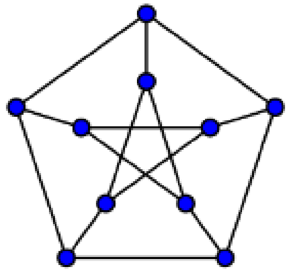
d = 2
k = 1
N = 3



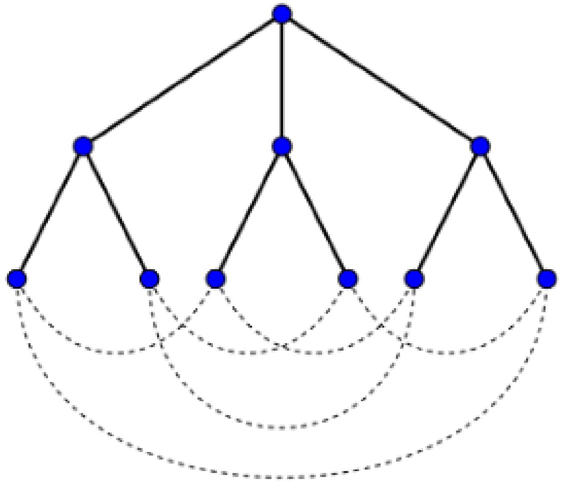
d = 2
k = 2
N = 4



d = 2
k = 2
N = 5
Moore!



d = 3
k = 2
N = 10
Moore!



Petersen graph

Short Distance Graphs

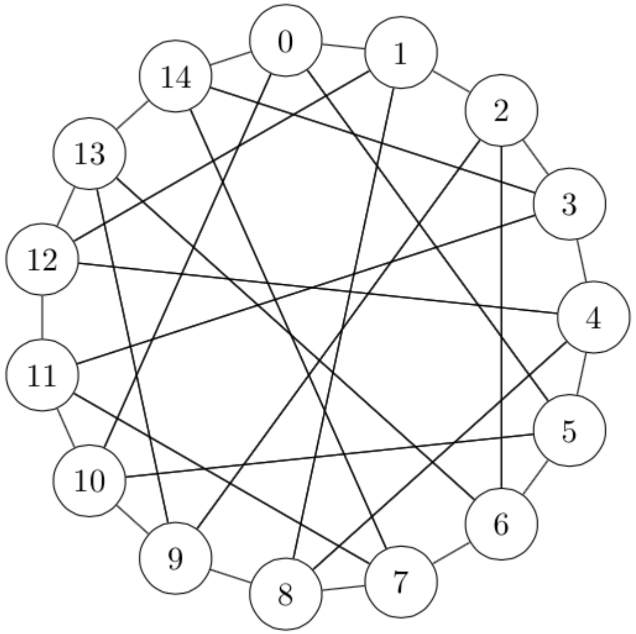
Moore Graph: a regular graph of **degree *d*** and **diameter *Δ*** (also ***k***) whose number of vertices

equals the **upper bound** $= \begin{cases} 2\Delta + 1 \;\; if \; d = 2 \\ \frac{d(d-1)^{\Delta}-2}{d-2} \;\; if \; d > 2 \end{cases}$

Petersen graph

Short Distance Graphs

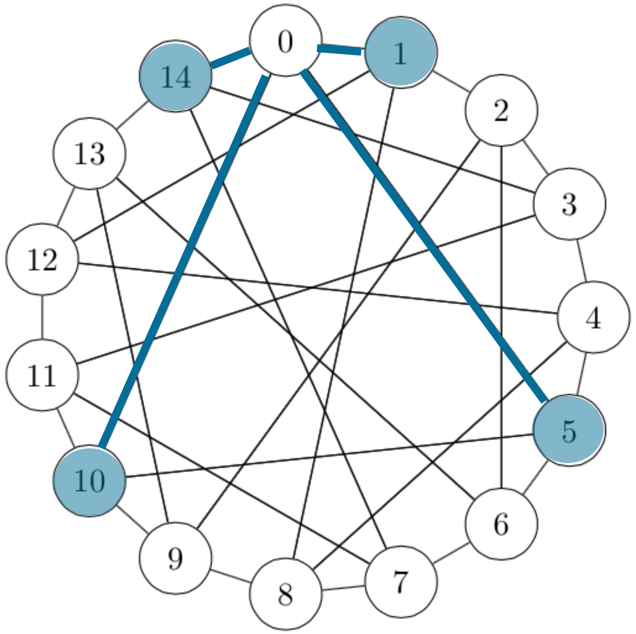
Moore Bound



$$N = 15 \quad d = 4 \quad \Delta = 2 \quad Aver.Dist. = 1.71$$

Short Distance Graphs

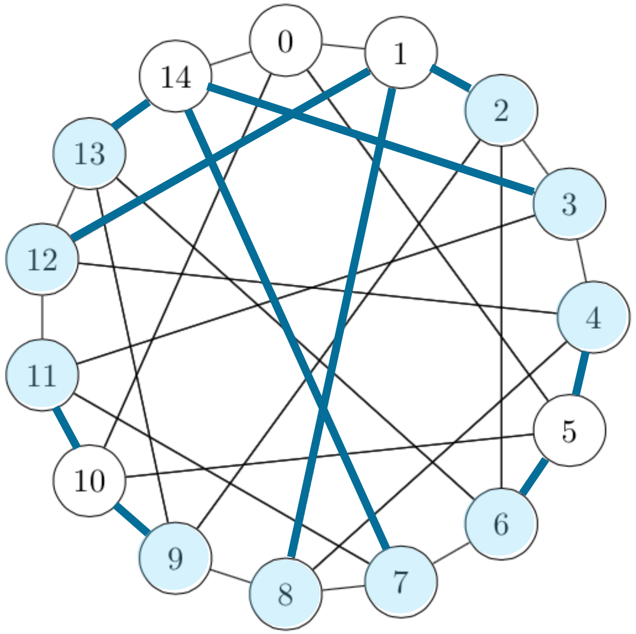
Moore Bound



$$N = 15 \quad d = 4 \quad \Delta = 2 \quad Aver.Dist. = 1.71$$

Short Distance Graphs

Moore Bound



$$N = 15 \quad d = 4 \quad \Delta = 2 \quad Aver.Dist. = 1.71$$

Short Distance Graphs

Generalized Chordal Rings

- A graph *G* is a **generalized chordal ring** if nodes can be labelled with **integers modulo *M*** and there exists a **divisor *p*** of *M* such that **node *i* is joined** to **node *j*** if **node *i + p*** is **joined** to **node *j + p***.
- *G* is a **chordal ring** if **all nodes (*i*,*i*+*1*)** appear

GCR Construction

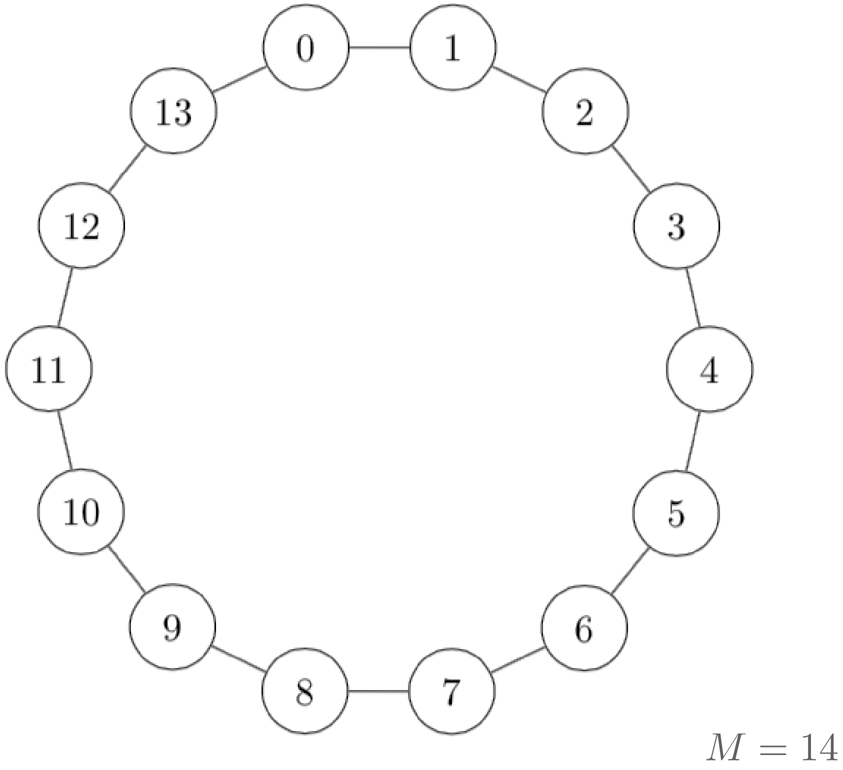
Obtained **from a ring** by **adding chords** (= additional link between nodes) by **variations** over:

- **Number *c*** of chords
- **Chord length *w*** = number of edges between two nodes (on the ring)
- **Period *p*** = number of nodes having the **same chord length** (same **connection pattern**)

$$0 \leq \text{number of chords} \leq \text{vertex degree} - 2$$
$$2 \leq \text{chord length} \leq M - 2$$
$$1 \leq \text{period} \leq M \text{ } div \text{ } 2$$
$$M \text{ } mod \text{ period} = 0$$

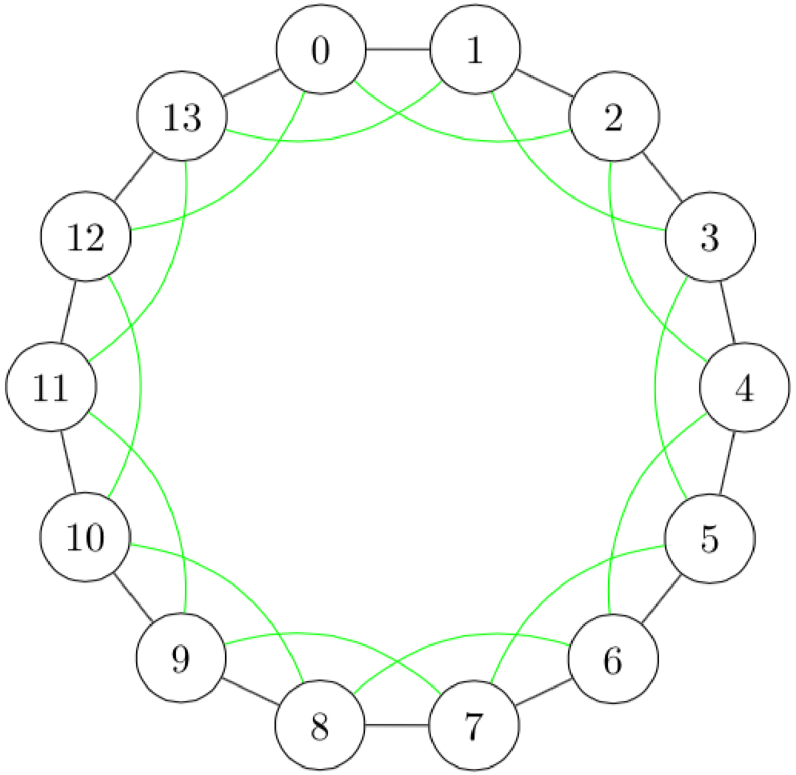
Short Distance Graphs

Example:



Short Distance Graphs

Example:



$P = 1$

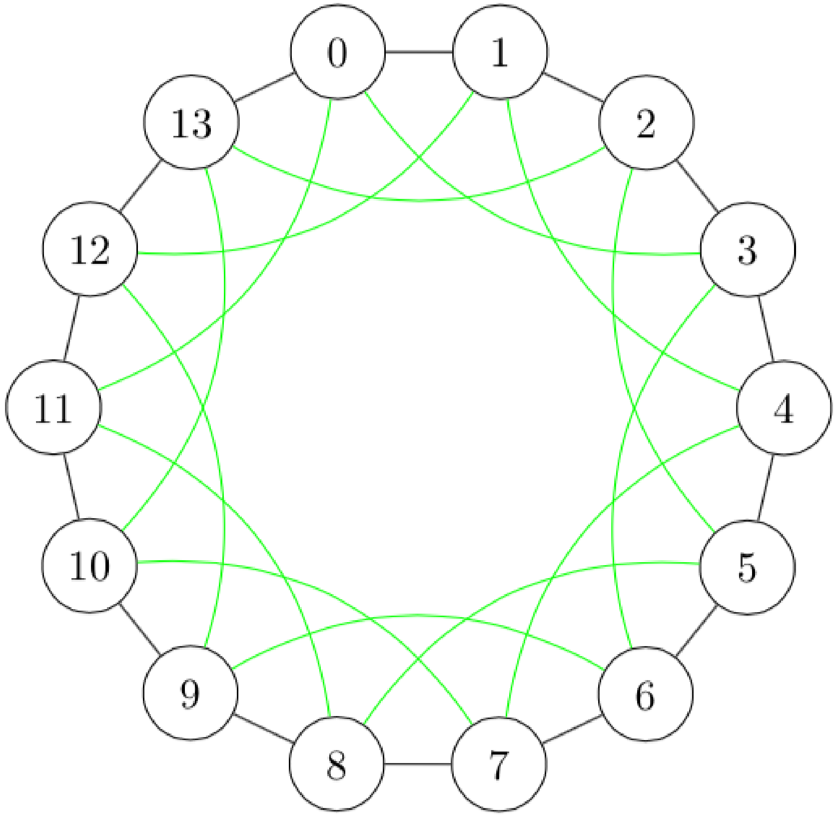
$W = 2$

$Diameter$

$\Delta = 4$

Short Distance Graphs

Example:



$P = 1$

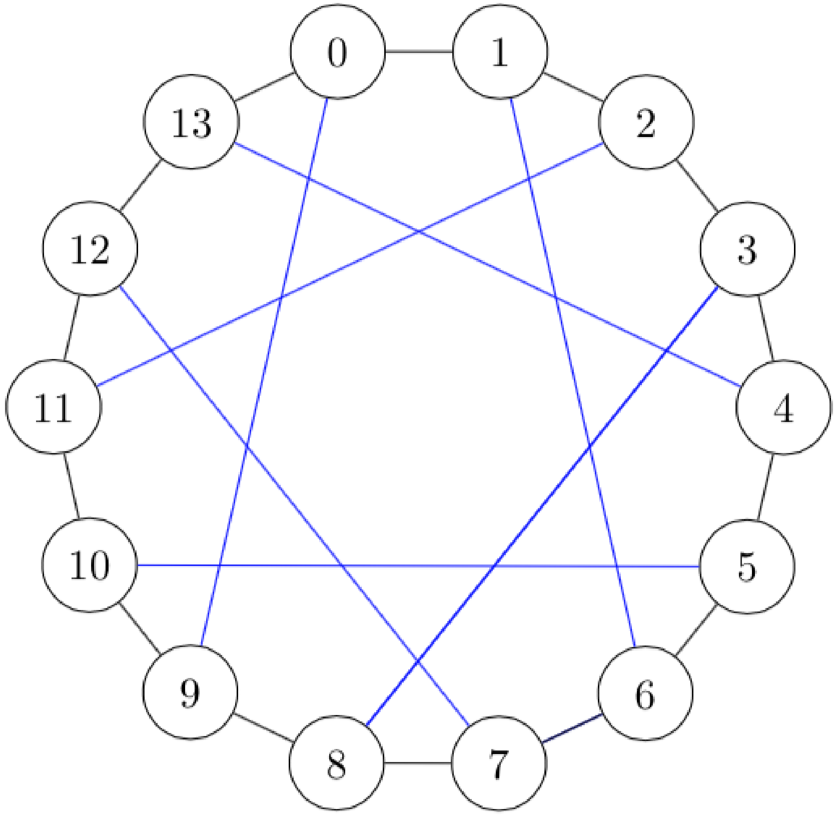
$W = 3$

→ *Diameter*

$\Delta = 3$

Short Distance Graphs

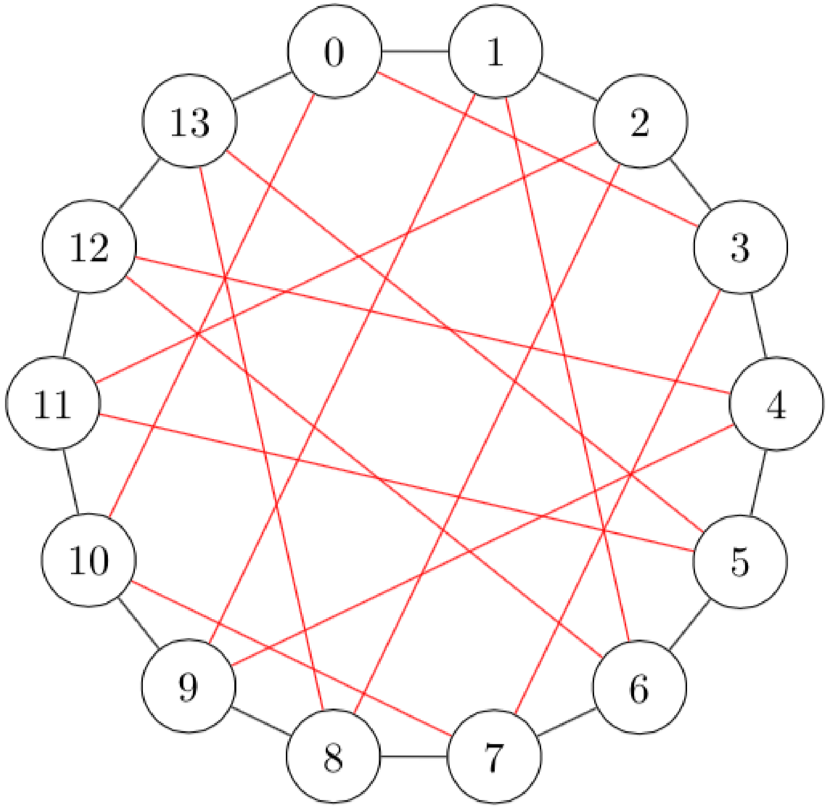
Example:



$$P = 1$$
$$W = 5, 9$$
$\rightarrow$ $Diameter$
$$\Delta = 3$$

Short Distance Graphs

Example:



$P = 7$

$W_1 = 3, 10$

$W_2 = 5, 8$

$W_3 = 6, 9$

$W_4 = 4, 11$

$W_5 = 5, 8$

$W_6 = 6, 8$

$W_7 = 6, 9$

→ *Diameter*

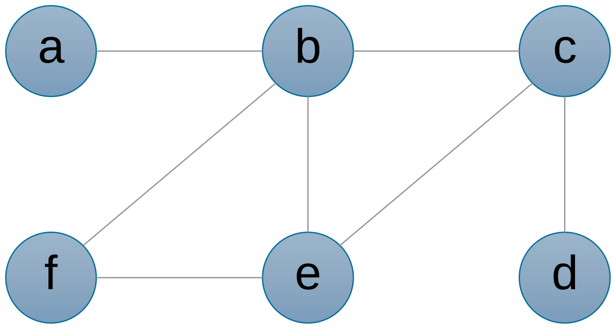
$\Delta = 2$

Distance and Centrality

(Degree) Centrality

Identify the **importance** of specific nodes or edges in the network as a significantly more important node or an edge may be **joining** two **distinctive** parts of the network.

$$C_D(i) = \sum_{j \in V} a_{ij}$$



The equation $C_D = A \times [1]$ where rows and columns are ordered a,…,f :

$$\begin{bmatrix} 1 \\ 4 \\ 3 \\ 1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\rightarrow \Theta(n^2)$$

Distance and Centrality

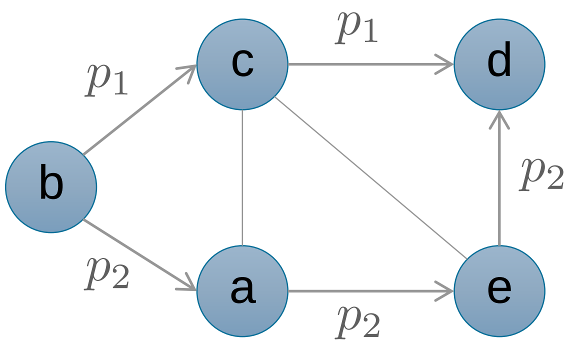
k-path centrality

of a vertex ***v***, ***k*-*C*$_D$(*v*)** is the **number** of **paths** of **length *k*** or less emanating from vertex *v*.

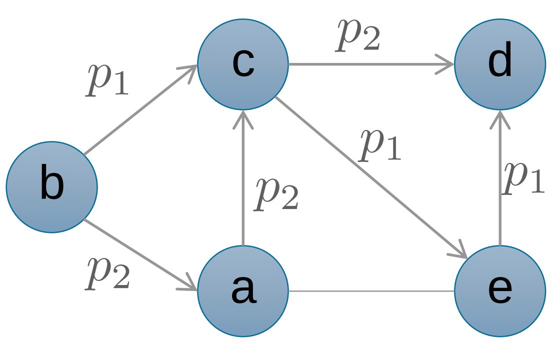
When *k*=*1*, ***k*-path centrality** is **equal** to **degree centrality**.

Edge disjoint paths (with common source/sink) do **not** have any **common edges**

Vertex disjoint paths do **not** have any **common vertices**



Vertex disjoint paths



Edge disjoint paths
(messages from source to sink
never have to use the same link)

Distance and Centrality

Edge disjoint k-path centrality

- of a vertex ***v*** is the **number** of **edge disjoint paths** of **length** ***k*** or **less** that **start** or **end** at vertex *v*. The ***Ford-Fulkerson Theorem*** states that the number of edge disjoint paths between two nodes *u* and *v* of a graph is equal to the **minimum number of vertices** that must be **removed to disconnect *u*** and ***v***.

Vertex disjoint k-path centrality

- of a node ***v*** is the **number of vertex disjoint paths** of **length** ***k*** or **less** that start or end at vertex v. ***Menger*** showed that the number of vertex disjoint paths between two nodes *u* and *v* **equals** the number of **nodes** that must be **removed** to **disconnect *u*** and ***v***.

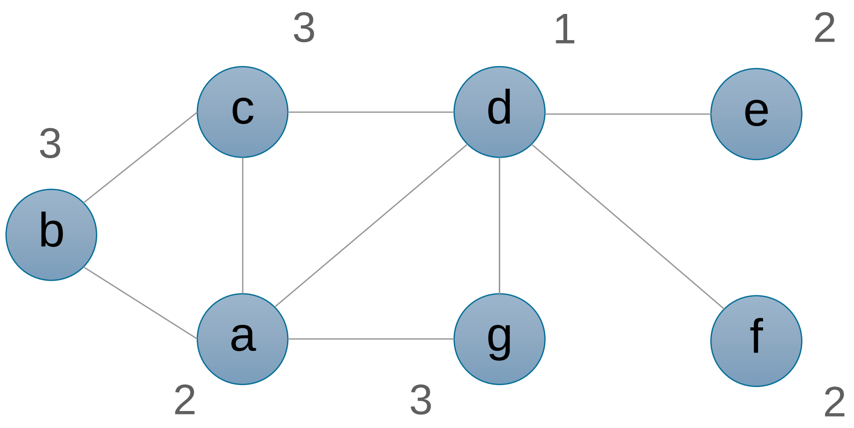
→ Major drawback with degree centrality: only **considers local information**.
(a node having few important neighbors may be more influential globally than a node which has many less important neighbors.)

Distance and Centrality

k-hop degree (k-hop-C_D)

- **number of neighbors** a node has in its **k-hop neighborhood**.
  For *k=1*, k-hop-C_D equals C_D.

k-rank **of a node** as its position in the descended sorted **degree list** of neighbors in its **k-hop neighborhood**.



$$\begin{bmatrix} a = 4 \\ b = 2 \\ c = 3 \\ d = 5 \\ e = 1 \\ f = 1 \\ g = 2 \end{bmatrix}$$

1-hop-degree

a: a,b,c,d,g → d,a,c,{b,g} → a=2
b: a,b,c → a,b,c →b=2
c: a,b,c,d → d,a,c,b → c=3
d: a,c,d,e,f,g → d,a,c,g,e,f → d=1
e: d,e → d,e → e=2
f: d,f →d,f → f=2
g: a,d,g → d,a,g → g=3

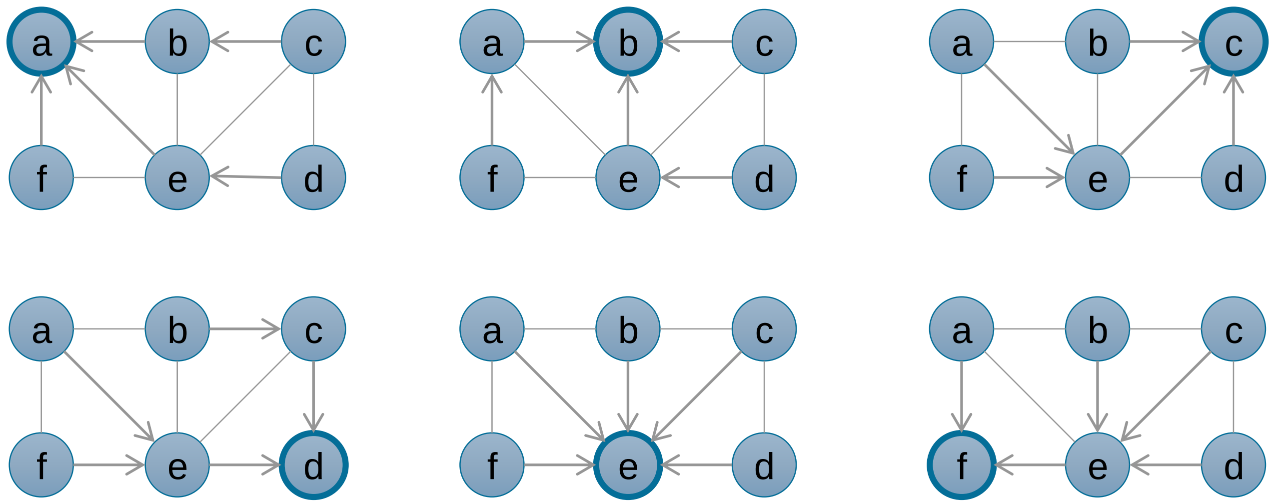
One-rank centrality example

Distance and Centrality

Closeness Centrality

- based on **global** rather than local **knowledge**
- defined as the **reciprocal** of the **total distance** from this vertex **to all other** vertices in a graph G(V,E)

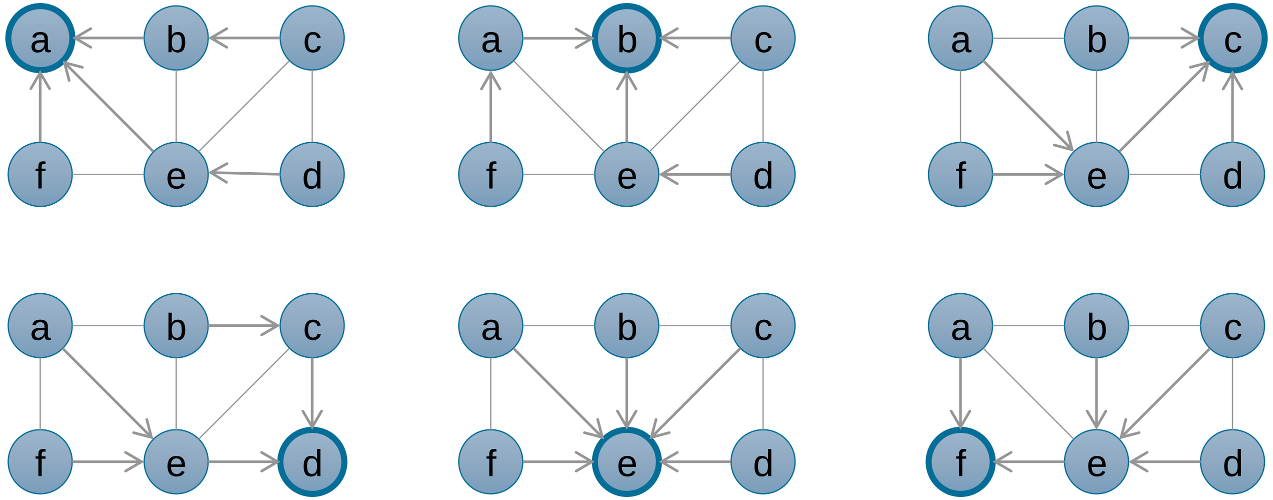
$$C_c(i) = \frac{1}{\sum_{j \in V} \delta(i,j)}$$



BFS trees for vertex distance calculation

Distance and Centrality

Closeness Centrality



$$\begin{bmatrix}7\\7\\7\\8\\5\\8\end{bmatrix} = \begin{bmatrix}0&1&2&2&1&1\\1&0&1&2&1&2\\2&1&0&1&1&2\\2&2&1&0&1&2\\1&1&1&1&0&1\\1&2&2&2&1&0\end{bmatrix} \times \begin{bmatrix}1\\1\\1\\1\\1\\1\end{bmatrix} \rightarrow$$

$$C_c(a) = \frac{1}{7}$$

$$C_c(b) = \frac{1}{7}$$

$$C_c(c) = \frac{1}{7}$$

$$C_c(d) = \frac{1}{8}$$

$$\boxed{C_c(e) = \frac{1}{5}}$$ Highest closeness centrality

$$C_c(f) = \frac{1}{8}$$

Distance and Centrality

Stress Centrality

total number of **all pairs shortest** paths that pass **through** a **vertex *v***

$$C_S(v) = \sum_{s \neq t \neq v} \sigma_{st(v)}$$

$where\ \sigma_{st(v)}\ is\ the\ number\ of\ shortest\ paths\ between\ vertices\ s\ and\ t.$

It is an estimation of the **stress** that a vertex in a network bears, assuming all communication will be carried along the shortest path.

Distance and Centrality

Betweenness Centrality

If the paths for **shortest paths between** nodes of a network pass through some vertices **more often** than others, then these vertices are significantly **more important** than others for communication purposes.

$$C_B(v) = \sum_{s \neq t \neq v} \frac{\sigma_{st(v)}}{\sigma_{st}}$$

Simple procedure to find C_B is to **calculate all shortest paths** in the graph G and **count** the number of paths that pass through v by excluding the paths that start or end at v.

$$where\ \sigma_{st(v)}\ is\ the\ total\ number\ of\ shortest\ paths\ between\ vertices\ s\ and\ t\ that\ pass\ through\ vertex\ v$$

Perform modified BFS on each node → $\Theta(n(n+m))$

Distance and Centrality

Betweenness Centrality :: Newman's Algorithm

```
forall v in V
      b_v ← 1
      find BFS paths from v to all other nodes
end

forall s in V
      starting from the farthest nodes, move from u towards s along paths
      using vertex v.
      b_v ← b_v+b_u

      If v has more than one predecessor, then b_v is devided equally
      between them.
end
```

Distance and Centrality

Eigenvalue Centrality

- a **vertex** in a network may be **considered important** if it has **important neighbors**.
- Given a graph *G(V,E)* with an adjacency matrix *A*, the **score** of a vertex *i* can be defined as **proportional to the sum of all its neighbors' scores**:

$$x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j = \frac{1}{\lambda} \sum_{j \in V} a_{ij} x_j$$

$$where\ N(i)\ is\ the\ set\ of\ neighbors\ of\ i\ and\ \lambda\ is\ a\ constant$$

- The **score** of a node can simply be its **degree**. We are **adding** the **degrees of** the **neighbors** of a vertex *i* to find its **new score**.

$\vec{x} = \frac{1}{\lambda} A \vec{x}$ which is the eigenvalue equation $A\vec{x} = \lambda \vec{x}$

- To find the **eigenvalue centrality** of a graph *G*, we need to find the **eigenvalues** of the **adjacency matrix *A***. We select the largest eigenvalue and its associated **eigenvector**. This eigenvector contains the **eigenvalue centralities** of all vertices.

Graphs

Part: Structure

Algorithms and Data Structures 2, 340300
Lecture – 2023W
Univ.-Prof. Dr. Alois Ferscha, teaching@pervasive.jku.at

Institute of Pervasive Computing

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at